Application Building with Spec 2.0

K. De Hondt, S. Ducasse with S. Jordan Montaño and E. Lorenzano

June 28, 2024

Keepers of the lighthouse
Édition : BoD - Books on Demand,
12/14 rond-point des Champs-Élysées,75008 Paris
Impression : Books on Demand GmbH, Norderstedt, Allemagne
ISBN: 9782322182015
Dépôt légal : 3/2022
Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

CHAPTER 1

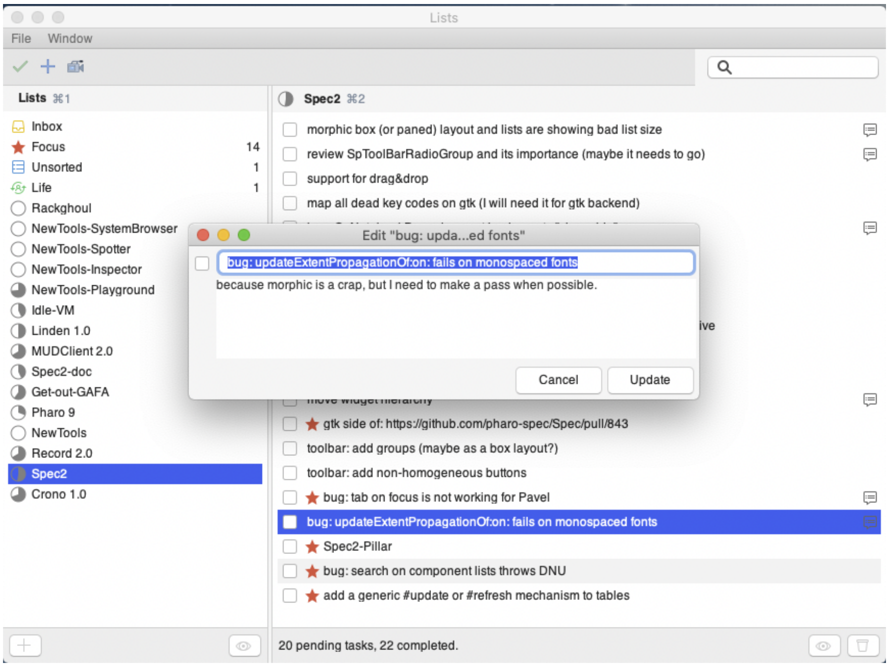
Introduction

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., Iceberg the git manager, Change Sorter, Critics Browser, and the Pharo debugger. An important architectural decision is that Spec supports multiple backends (at the time of writing this book, GTK and Morphic are available).

1.1 Reuse of logic

The fundamental principle behind Spec is the reuse of user interface logic and its visual composition. User interfaces are built by reusing and composing existing user interfaces, and 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 a 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 part of an even bigger floor design.

To allow such reuse, Spec was influenced by VisualWorks' and Dolphin Smalltalk's Model View Presenter (MVP) pattern. Spec recognizes the need for a Presenter class. A presenter represents the glue between a domain and widgets as well as the logic of interaction between the widgets composing the application.



Inbox
Focus 14
Unsorted 1
Life 1
Rackghoul
NewTools-SystemBrowser
NewTools-Spotter
NewTools-Inspector
NewTools-Playground
Idle-VM
Linden 1.0
MUDClient 2.0
Spec2-doc
Get-out-GAFA
Pharo 9
NewTools
Record 2.0
Spec2
Crono 1.0

morphic box (or paned) layout and lists are showing bad list size
review SpToolBarRadioGroup and its importance (maybe it needs to go)
support for drag&drop
map all dead key codes on gtk (I will need it for gtk backend)
bug: updateExtentPropagationOf:on: fails on monospaced fonts
because morphic is a crap, but I need to make a pass when possible.
move widget hierarchy
gtk side of: https://github.com/pharo-spec/Spec/pull/843
toolbar: add groups (maybe as a box layout?)
toolbar: add non-homogeneous buttons
bug: tab on focus is not working for Pavel
bug: updateExtentPropagationOf:on: fails on monospaced fonts
Spec2-Pillar
bug: search on component lists throws DNU
add a generic #update or #refresh mechanism to tables
20 pending tasks, 22 completed.

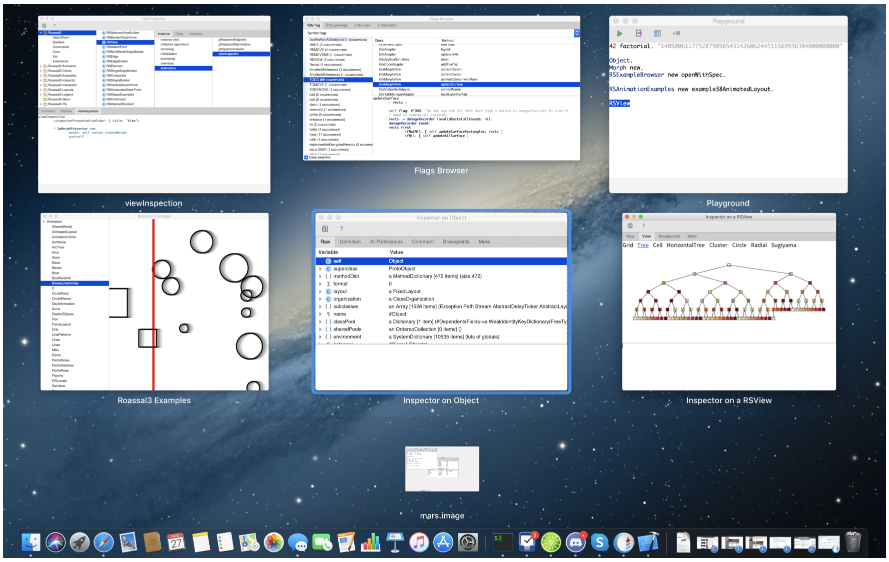
Figure 1-1 Spec supports multiple backends Morphic and GTK3.0.: Here we see GTK. %width=100

In Spec 1.0, this role was filled by the class `ComposableModel` and now, in Spec 2.0, the class is called `SpPresenter`. A presenter manages the *logic UI and the link between widgets and domain objects*. Fundamentally, when writing Spec code, developers do *not* come into contact with UI widgets. Instead, they program a Presenter that holds the UI logic (interactions, layout, ...) and talks to domain objects. When the UI is opened, this presenter instantiates the appropriate widgets. This being said, for developers, this distinction is not apparent and it feels as if the widgets are being programmed directly.

Spec is the standard GUI framework in Pharo and differs from Pharo's other GUI frameworks such as Morphic. It is restricted in that it only allows one to build user interfaces for applications that have typical GUI widgets such as buttons, lists, etc. It cannot be used as a general drawing framework, but you can integrate a canvas inside a Spec component.

For example, you can embed a Roassal visualization (see Figure **??**), or you can extend Spec itself with additional native components.

Another example of integration is the NovaStelo project of Prof. E. Ito as shown in Figure **??**. It shows that Spec can be used for the overall structure of the ap-



```
42 factorial. "1405006117752879898543142606244511569936384000000000"

Object.
Morph new.
RSExampleBrowser new openWithSpec.

RSAnimationExamples new example38AnimatedLayout.

RSView
```

Figure 1-2 Roassal and Spec integration. %width=100&anchor=SpecRoassal

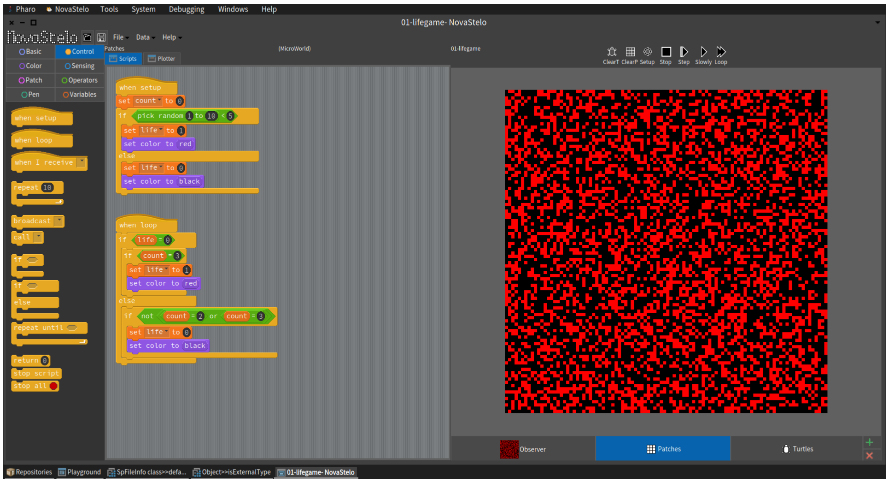
plication and embed specific elements.

1.2 Spec 2.0

Since Spec 2.0, different widget sets can be used to render your applications. At the time of writing this book, Spec can be rendered using either Morphic or GTK as a backend. Spec 2.0 represents a large iteration over Spec 1.0. Many enhancements have been introduced: the way user interface layouts are expressed, the API has been revisited, new widgets are supported, and integration with other projects, such as `Commander`, has been added.

Pharo's objective is to use Spec to build all its own GUIs. This ensures strong support of Spec over time and improves the standardization of Pharo's interfaces as well as their portability to new graphical systems. Using Spec 2.0 provides backend independence and logic reuse. This means that a UI written in Spec will be rendered on backends other than GTK and Morphic. As new backends become available, all applications written in Spec will be able to use them.

While this book uses previous Spec documentation as a foundation, the text has been almost completely rewritten with an aim toward higher quality. We hope that it will be of use to developers who write UIs in Pharo. This book focuses on Pharo 12. Earlier versions of Pharo come equipped with different versions of Spec, which may cause some code samples from this book to break.



```
when setup
set count to 0
if pick random 1 to 10 < 5
  set life to 1
  set color to red
else
  set life to 0
  set color to black
```

```
when loop
if life = 0
  if count = 3
    set life to 1
    set color to red
else
  if not count = 2 or count = 3
    set life to 0
    set color to black
```

Figure 1-3 An integration of Morphic Native Widgets and Spec. %width=100&anchor=NovaStelo

Nevertheless, the fundamental principles of UI development in Spec are the same.

1.3 Code

The code of all the examples in this book is stored at https://github.com/SquareBracketAssociates/CodeOfSpec20Book.

You can load the code by evaluating this code snippet:

```
Metacello new
  baseline: 'CodeOfSpec20Book';
  repository: 'github://SquareBracketAssociates/CodeOfSpec20Book/src';
  load
```

1.4 Acknowledgements

Even though due to the lack of manpower the fundraising campaign was not used, the authors would like to express their warm gratitude to the following people for their financial support: Masashi Fujita, Roch-Alexandre Nominé, Eiichiro Ito, sumim, Hilaire Fernandes, Dominique Dartois, Philippe Mougin, Pavel Krivanek, Michael L. Davis, Ewan Dawson, Luc Fabresse, David Bajger, Jörg Frank, Petter Egesund, Pierre Bulens, Tomohiro Oda, Sebastian Heidbrink, Alexandre Bergel, Jonas Skučas, and Mark Schwenk.

We want to thank I. Thomas for her chapter on the inspector, and R. De Villemeur for the chapter on Athens integration.

Finally, Stéphane Ducasse wants to thank Johan Fabry for his co-authoring of the first book on Spec 1.0. Without that first book, this one would not exist.

If you supported us and you are not on this list, please contact us or do a pull request.

Part I

All Spec in One Example

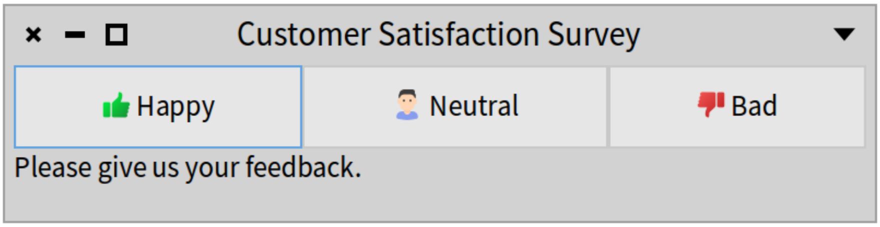
CHAPTER 2

A 10 min small example

We will construct a small but complete user interface. This will allow you to build basic user interfaces.

After completing this chapter you may read Chapter 7 about the reuse of Spec presenters, which is the key behind the power of Spec. With these two chapters, you should be able to construct Spec user interfaces as intended. You could use the rest of this book 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



Please give us your feedback.

Figure 2-1 A screenshot of the customer satisfaction survey UI. % width=50&anchor=figCustomersBasic

We construct a simple customer satisfaction survey UI, which allows a user to give feedback about a service by clicking on one of three buttons. This feedback should be recorded and processed, but that is outside of the scope of this example. Figure **??** shows a screenshot of the UI.

2.2 Create the class of the UI

All user interfaces in Spec are subclasses of `SpPresenter`, so the first step in creating the UI is subclassing that class:

```
SpPresenter << #CustomerSatisfactionPresenter
  slots: { #buttonHappy . #buttonNeutral . #buttonBad . #result};
  package: 'CodeOfSpec20Book'
```

The instance variables of the class hold the *presenters* the UI contains, the so-called *subpresenters*. In this case, we have three buttons and a text to show the result of the survey.

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

2.3 Instantiate and configure subpresenters

A subclass of `SpPresenter` has the responsibility to define the `initializeP-resenters` method, which instantiates and configures the presenters used in the user interface. We will discuss it piece by piece. Note that since this method may be a bit long we will split it into pieces that represent their intent.

Presenter creation

```
CustomerSatisfactionPresenter >> initializePresenters

  result := self newLabel.
  buttonHappy := self newButton.
  buttonNeutral := self newButton.
  buttonBad := self newButton.
```

`SpPresenter` defines messages for the creation of standard presenters: `new-Button`, `newCheckBox`, `newDropList`, ... All of these are defined in the `script-ing - widgets` protocol of the `SpTPresenterBuilder` trait. They are short-cuts to create presenters.

The following method shows how `newButton` is defined.

```
SpPresenter >> newButton

  ^ self instantiate: SpButtonPresenter
```

Note that the naming may be a bit confusing since we write `newButton` while it will create a button *presenter* and not a button *widget*, which Spec will take care by itself. Spec provides `newButton` because it is easier to use than `newButton-Presenter`.

Do not call `new` to instantiate a presenter that is part of your UI. An alternative way to instantiate presenters is to use the message `instantiate:` with a presenter class as an argument. For example `result := self instantiate: SpLabelPresenter`. This allows one to instantiate standard and non-standard presenters.

Presenter configuration

The next step is configuring the buttons of our UI. The message `label:` sets the button label and the message `icon:` specifies the icon that will be displayed near the label.

```
CustomerSatisfactionPresenter >> initializePresenters

  ... continued ...
  result 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)
```

The method `iconNamed:` of `SpPresenter` uses an icon provider to fetch the icon with the given name. You can browse the Spec icon provider by looking at `SpPharoThemeIconProvider`, which is a subclass of `SpIconProvider`. Each application is able to define its own icon provider by defining a subclass of `SpIconProvider`.

Presenter interaction logic

Now we define what will happen when the user presses a button. We define this in a separate method called `connectPresenters`:

```
CustomerSatisfactionPresenter >> connectPresenters

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

We use the message `action:` to specify the action that is performed when the button is clicked. In this case, we change the content of the result text to inform the user 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, some message should be sent to a subpresenter.

To summarize:

- Specialize `initializePresenters` to define and configure the presenters that are the elements of your UI.
- Specialize `connectPresenters` to connect those presenters together and specify their interaction.

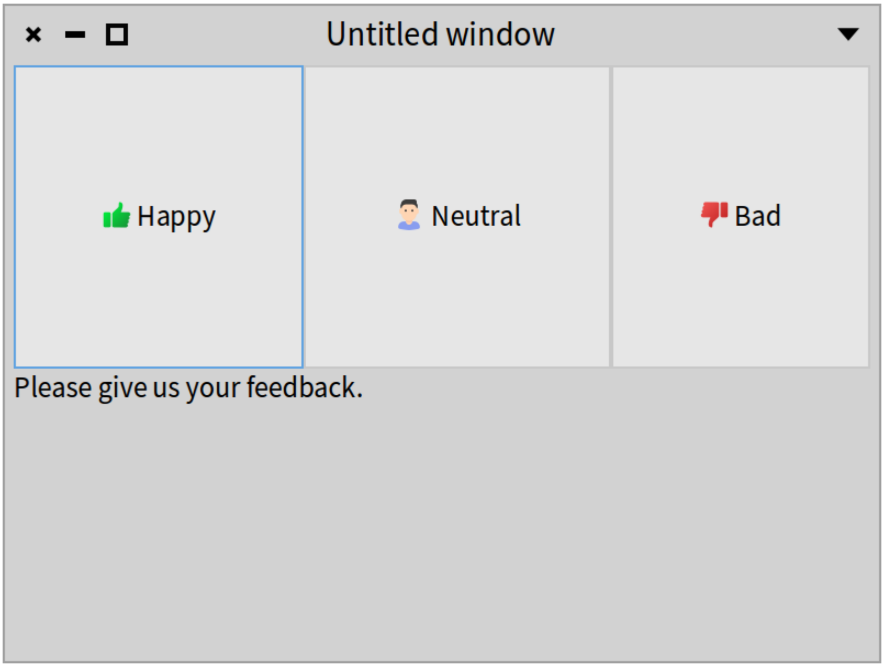
Specifying the presenter layout

The presenters have been defined and configured, but their placement in the UI has not yet been specified. This is the role of the method `defaultLayout`.

```
CustomerSatisfactionPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    add: (SpBoxLayout newLeftToRight
        add: buttonHappy;
        add: buttonNeutral;
        add: buttonBad;
        yourself);
    add: result;
    yourself
```

In this layout, we add two rows to the UI, one with the buttons and one with the result text. Defining presenter layout is a complex process with many different possible requirements, hence in this chapter we do not talk in detail about layout specification. For more information we refer to Chapter 10.

Once the method `defaultLayout` is defined, you can open your UI with `CustomerSatisfactionPresenter new open`. You should see a window similar to the one shown in Figure **??**.



Please give us your feedback.

Figure 2-2 A first version of the customer satisfaction UI. % width=50&anchor=fig-FirstCut

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

To set the window title and the initial size of your presenter, you have to specialize the method `initializeWindow:` as follows:

```
CustomerSatisfactionPresenter >> initializeWindow: aWindowPresenter

  super initializeWindow: aWindowPresenter.
  aWindowPresenter
    title: 'Customer Satisfaction Survey';
    initialExtent: 400@100
```

You are free to use helper methods to return the title and extent of your presenter. When you reopen your presenter, and you click the "Happy" button, you should see the window shown in Fig. **??**.

Sending the `open` message to a presenter will open a window and return an instance of `SpWindowPresenter`, which allows the window to be closed from code.

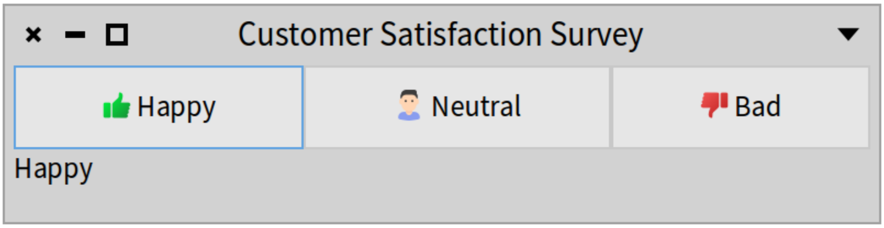


Figure 2-3 A final version of the customer satisfaction UI. % width=50&anchor=figSecondCut

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

Note that to update the contents of your window once it is open, you have the method `SpPresenter>>withWindowDo:`, but we will discuss it later in this book. More information about managing windows, e.g., opening dialog boxes or setting the about text is present in Chapter 9.

This concludes our first example of a Spec user interface. In the next chapter, we continue with more examples on how to configure the different presenters that can be used in a user interface.

2.5 Conclusion

In this chapter, we have given you a small example of Spec user interfaces. We have shown you what the different steps are to build a user interface with Spec.

More examples of Spec user interfaces are found in the Pharo image. Since all Spec user interfaces are subclasses of `SpPresenter`, they are easy to find and each of them may serve as an example. Furthermore, experimentation with presenters and user interfaces is made easy because all presenters can be opened as standalone windows.

We recommend that you at least read Chapter 7 about reuse of Spec presenters, 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.

CHAPTER 3
Most of Spec in one example

In this chapter, we will guide you through the building of a simple but non-trivial application to manage films as shown in Figure 3-1. We will show many aspects of Spec that we will revisit in depth in the rest of this book: the application, presenters, the separation between domain and presenter, layout, transmissions to connect widgets, and styles.

3.1 Application

Spec 2.0 introduces the concept of an application. An application is a small object responsible for keeping the state of your application. It manages, for example, the multiple windows that compose your application, and its backend (Morphic or GTK), and can hold properties shared by the presenters.

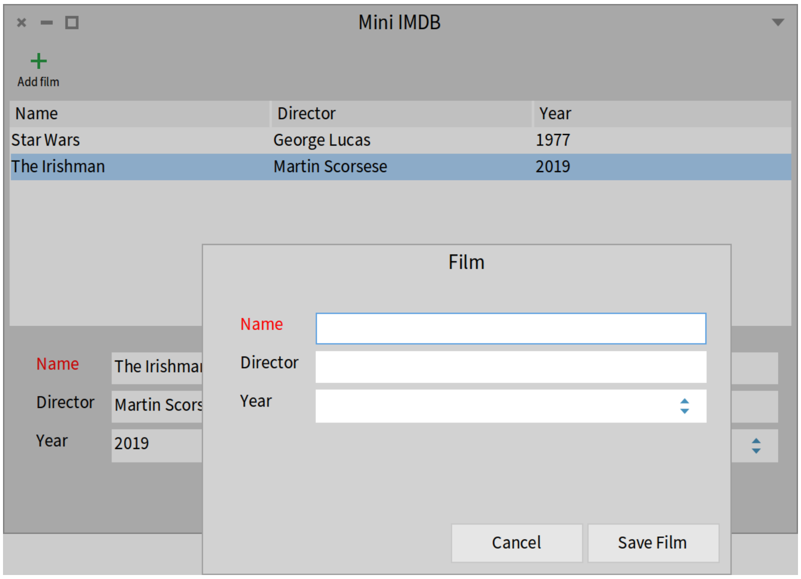
We start with the definition of the example application class:

```
SpApplication << #ImdbApp
  package: 'CodeOfSpec20Book'
```

3.2 A basic film model

Since we will manage films we define an `ImdbFilm` class as follows. It has a name, a year, and a director. We generate the companion accessors.

```
Object << #ImdbFilm
  slots: {#name . #year . #director};
  package: 'CodeOfSpec20Book'
```



Name	Director	Year
Star Wars	George Lucas	1977
The Irishman	Martin Scorsese	2019

Name	The Irishman
Director	Martin Scors
Year	2019

Name	
Director	
Year	

Figure 3-1 Film app: reusing the same component to edit and browsing a film.

We need a way to store and query films. We could use Voyage (https://github.com/pharo-nosql/voyage) since it works without an external Mongo DB. But we want to keep it extremely simple. So let's define a singleton.

We define a *class* instance variable called `films`.

```
Object class << ImdbFilm class
  slots: { #films }
```

We define a method that lazy initializes the `films` variable to an ordered collection.

```
ImdbFilm class >> films

  ^ films ifNil: [ films := OrderedCollection new ]
```

And to finish we define a way to add a film to the list.

```
ImdbFilm class >> addFilm: aFilm

  films add: aFilm
```

Now we are ready to define a first presenter that manages a list of films.

3.3 List of films

We define a presenter to manage a list of films by introducing a new class named `ImdbFilmListPresenter` which inherits from `SpPresenter`. We add an instance variable named `filmList` that will hold an elementary list presenter.

```
SpPresenter << #ImdbFilmListPresenter
  slots: { #filmList };
  package: 'CodeOfSpec20Book'
```

We define how the information should be presented by defining a method named `defaultLayout`. We specify a simple vertical box layout with the `filmList` as the only element.

defaultLayout

```
ImdbFilmListPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: filmList;
      yourself
```

When you do not define any other methods to represent layout, `defaultLayout` is the method that is invoked by Spec logic.

A presenter can have subpresenters. `ImdbFilmListPresenter` contains a table presenter and you will see later that:

1. a presenter can have multiple layouts
2. layouts can be defined dynamically

In Spec, layouts are dynamic by default and are expressed at the instance level. To allow backward compatibility, it is still possible to define a `defaultLayout` *class-side* method that returns a layout instead of using a `defaultLayout` instance-side method, but it is not the recommended way.

initializePresenters

So far, we have not initialized `filmList`.

The place to initialize the subpresenters is the method `initializePresenters` as shown below. There we define that `filmList` is a table with three columns. The message `newTable` instantiates a `SpTablePresenter`.

```
ImdbFilmListPresenter >> initializePresenters

  filmList := self newTable
    addColumn: (SpStringTableColumn title: 'Name'
      evaluated: #name);
    addColumn: (SpStringTableColumn title: 'Director'
      evaluated: #director);
    addColumn: (SpStringTableColumn title: 'Year'
      evaluated: #year);
    yourself
```

The following expression creates an instance of the film list presenter and opens it. You get the window shown in Figure **??**.

```
ImdbFilmListPresenter new open
```

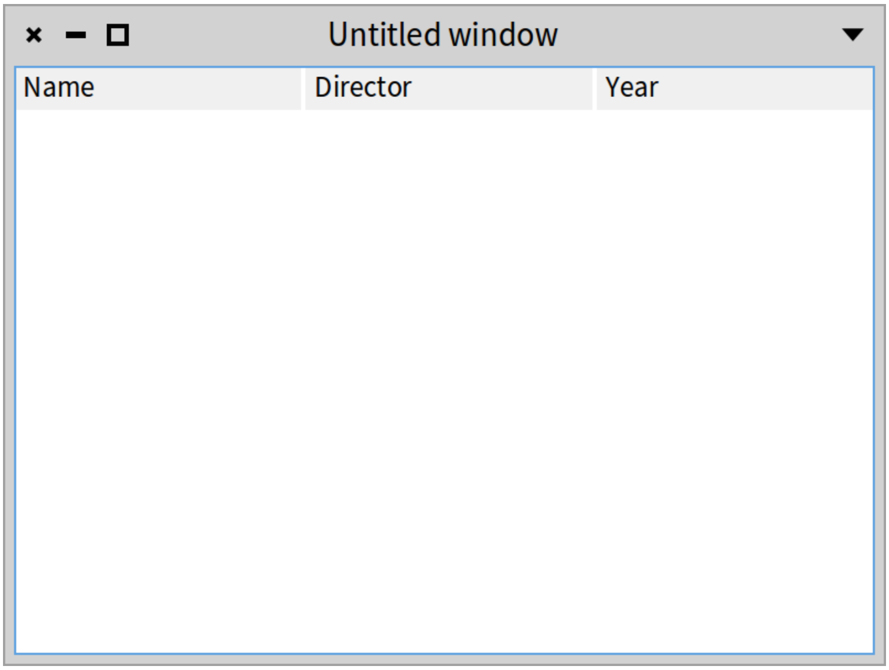


Figure 3-2 A layout and a simple `initializePresenters` showing an empty list of films. %width=60&anchor=LayoutInitilalizePresenters

3.4 Filling up the film list

We define the method `updatePresenter` which is automatically invoked after `initializePresenters`. It just queries the domain (`ImdbFilm`) to get the list of the recorded films and populates the internal table. Right now we do not have any film in the singleton so the list of films is empty.

```
ImdbFilmListPresenter >> updatePresenter

  filmList items: ImdbFilm films
```

If you want, just add a film and reopen the presenter. You should see the film on the list.

```
ImdbFilm addFilm: (ImdbFilm new
  name: 'E.T.';
  director: 'Steven Spielberg';
  year: '1982';
  yourself)
```

3.5 Opening presenters via the application

While directly creating a presenter is possible during development, a more canonical way to create a presenter is to ask the application using the message `newPresenter:` as follows.

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmListPresenter) open
```

The application is responsible for managing windows and other information, therefore it is important to use it to create presenters that compose the application.

3.6 Improving the window

A presenter can be embedded in another presenter as we will show later. It can also be placed within a window and this is what the message `open` does. Spec offers another hook, the method `initializeWindow:`, to specialize the information presented when a presenter is displayed within a window.

The method `initializeWindow:` allows you to define a title, a default size (message `initialExtent:`), and a toolbar.

```
ImdbFilmListPresenter >> initializeWindow: aWindowPresenter

  | addButton toolbar |
  addButton := self newToolbarButton
      label: 'Add film' ;
      icon: (self iconNamed: #smallAdd);
      action: [ self addFilm ];
      yourself.
  toolbar := self newToolbar
    add: addButton;
```

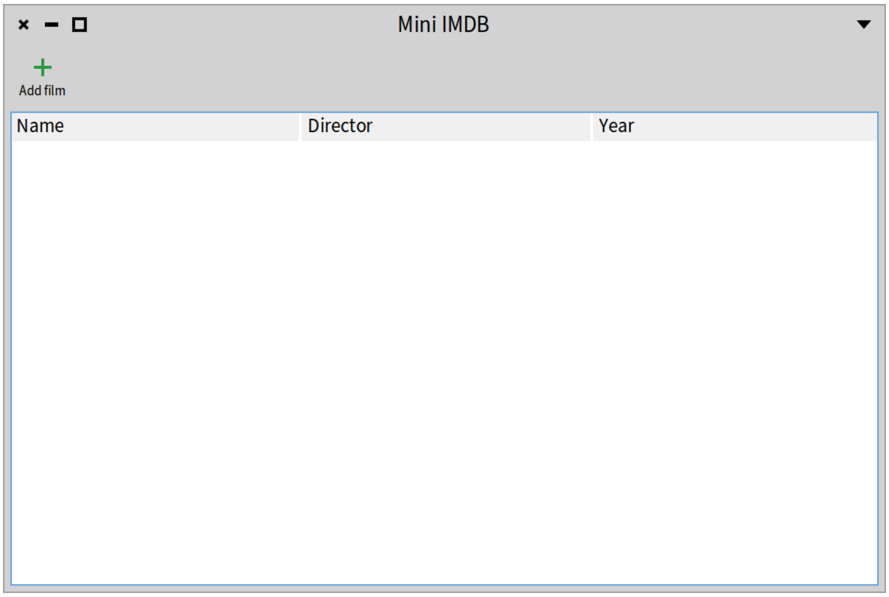


Figure 3-3 Film list presenter with a toolbar and a decorated window. %width=60&anchor=figFilmListPresenter2

```
    yourself.
  aWindowPresenter
    title: 'Mini IMDB';
    initialExtent: 600@400;
    toolbar: toolbar
```

You should obtain the window with a toolbar as shown in Figure **??**. To make sure that the `Add film` button does not raise an error, we trigger an `addFilm` method that is defined with no behavior. In fact, we will define a different presenter to be able to define a film.

```
ImdbFilmListPresenter >> addFilm

  "empty for now"
```

As we will see in Chapter 18, toolbars can be automatically created out of commands. We could have added the toolbar in that way to the `filmList` (e.g. using an instance variable) as part of the `ImdbFilmListPresenter` because the toolbar is also a presenter (similar to the table presenter or other predefined presenters). But doing it that way is less modular. Note also that the toolbar we created could be factored in a separate class to increase reuse too.

3.7 An application manages icons

What we can see from the definition of the method `initializeWindow:` is that an application manages icons with the message `iconNamed:`. Indeed, a presenter defines the `iconNamed:` message as a delegation to its application. In addition, your application can define its own icon set using the message `iconManager:`.

3.8 FilmPresenter

We are ready to define a simple presenter to edit a film. We will use it to add a new film or simply display it. We create a new subclass of `SpPresenter` named `ImdbFilmPresenter`. This class has three instance variables: `nameText`, `directorText`, and `yearNumber`.

```
SpPresenter << #ImdbFilmPresenter
  slots: { #nameText . #directorText . #yearNumber };
  package: 'CodeOfSpec20Book'
```

As we did previously, we define a default layout. This time we use a grid layout. With a grid layout, you can choose the position in the grid where your presenters will appear.

```
ImdbFilmPresenter >> defaultLayout

  ^ SpGridLayout new
      add: 'Name' at: 1@1; add: nameText at: 2@1;
      add: 'Director' at: 1@2; add: directorText at: 2@2;
      add: 'Year' at: 1@3; add: yearNumber at: 2@3;
      yourself
```

Note that it is not required to create the accessors for the presenter elements as we were forced to do in Spec 1.0. Here we only create getters because we will need them when creating the corresponding `ImbdFilm` instance.

```
ImdbFilmPresenter >> year

  ^ yearNumber text

ImdbFilmPresenter >> director

  ^ directorText text

ImdbFilmPresenter >> name

  ^ nameText text
```

For convenience, a SpGridLayout also comes with a builder that lets you add elements to the layout in the order they will appear. The previous layout definition can be rewritten as:

```
ImdbFilmPresenter >> defaultLayout

  ^ SpGridLayout build: [ :builder |
      builder
        add: 'Name'; add: nameText; nextRow;
        add: 'Director'; add: directorText; nextRow;
        add: 'Year'; add: yearNumber ]
```

Pay attention: do not add a yourself message here because you would return the class and not the layout instance.

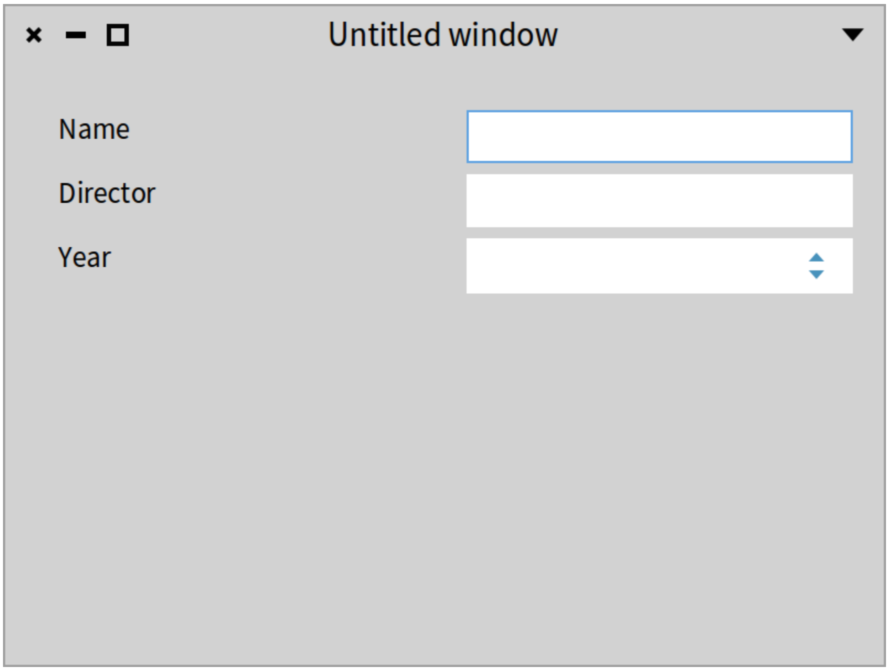


Figure 3-4 A single film presenter. % width=50&anchor=figFilmPresenter1

And as before, we define the method initializePresenters to initialize the variables to the corresponding elementary presenters. Here nameText and directorText are initialized to a text input, and yearNumber is a number input.

```
ImdbFilmPresenter >> initializePresenters

  nameText := self newTextInput.
  directorText := self newTextInput.
  yearNumber := self newNumberInput
    rangeMinimum: 1900 maximum: Year current year;
    yourself
```

Now we can try our little application with the following script and obtain a window similar to the one shown in Figure **??**:

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmPresenter) open
```

3.9 Better looking FilmPresenter

We improve the look of the film presenter by specifying column behavior and setting window properties. As you can see, the form to present Film data has very large labels. Indeed, they take half of the form width. We can solve that by using non-homogenous columns and asking the second column to take the biggest possible width with `column:expand:`. See Figure **??**.

```
ImdbFilmPresenter >> defaultLayout

  ^ SpGridLayout build: [ :builder |
      builder
        beColumnNotHomogeneous;
        column: 2 expand: true;
        add: 'Name'; add: nameText; nextRow;
        add: 'Director'; add: directorText; nextRow;
        add: 'Year'; add: yearNumber ]
```

Now we set the window properties by adding the following new `initializeWindow:` method. We get the situation shown in Figure **??**.

```
ImdbFilmPresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter
    title: 'Film';
    initialExtent: 400@250
```

3.10 Opening FilmPresenter in a modal dialog

Instead of opening the film presenter in a separate window, we like to open it in a modal dialog window. The modal dialog blocks the user interface until the

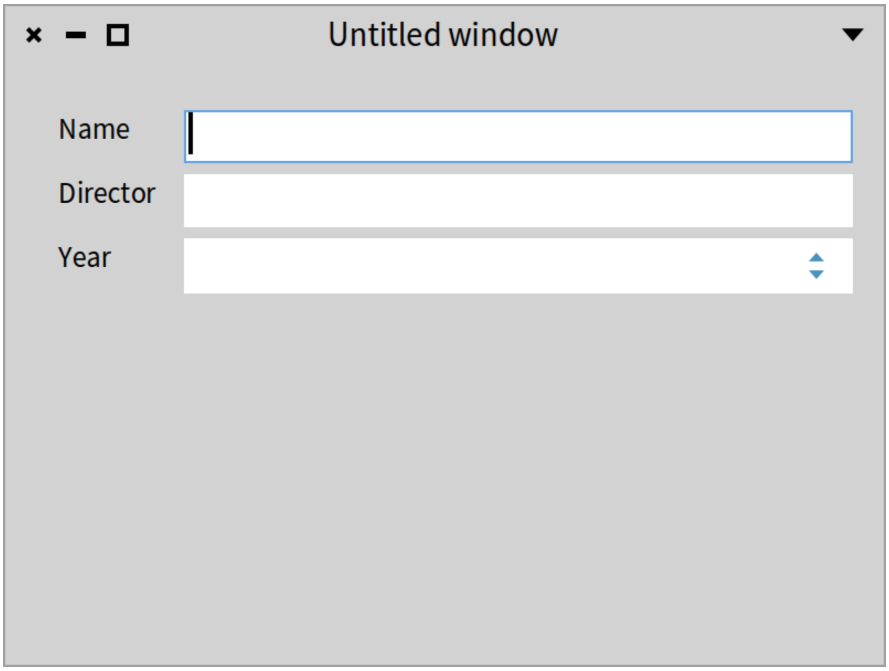


Figure 3-5 Using a non-homogenous grid layout. % width=50&anchor=FilmListPresenter2

user confirms or cancels the dialog. A modal dialog has no window decorations and it cannot be moved.

While a window can be opened by sending `open` to an instance of a presenter class, a dialog can be opened by sending `openModal`.

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmPresenter) openModal
```

Figure **??** shows the result. Note that there are no UI components to close the dialog. Press the "Esc" key on the keyboard to close it.

3.11 Customizing the modal dialog

Spec lets us adapt the dialog window, for example, to add interaction buttons. We specialize the method `initializeDialogWindow:` to add two buttons that control the behavior of the application, as shown in Figure **??**. We also center

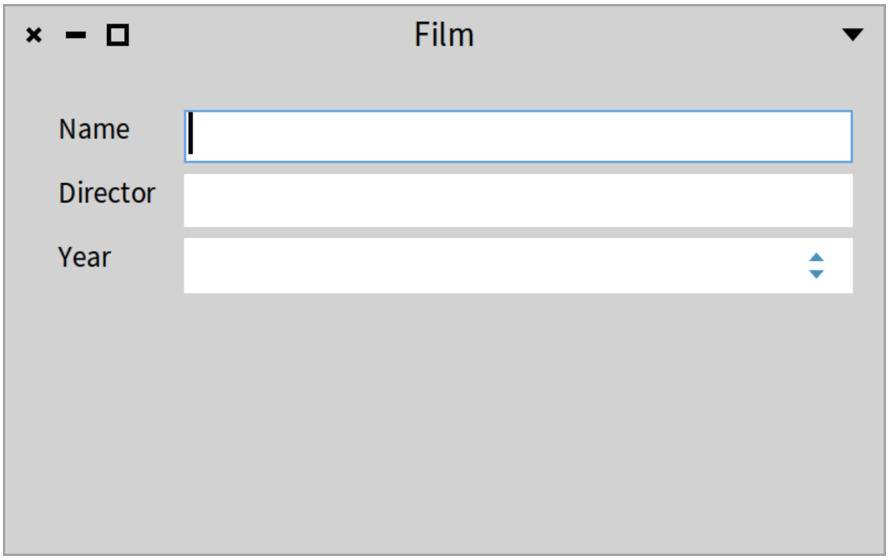
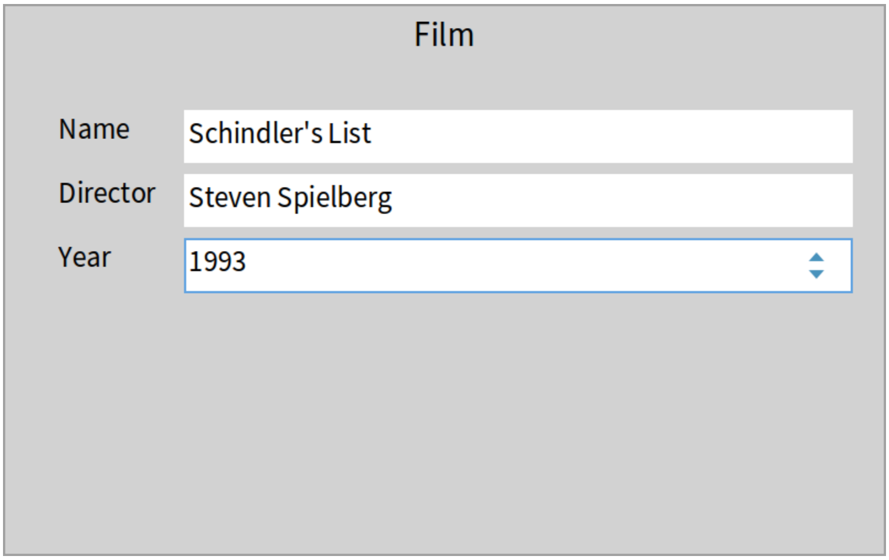


Figure 3-6 Better window. % width=50&anchor=FilmListPresenter3



Schindler's List

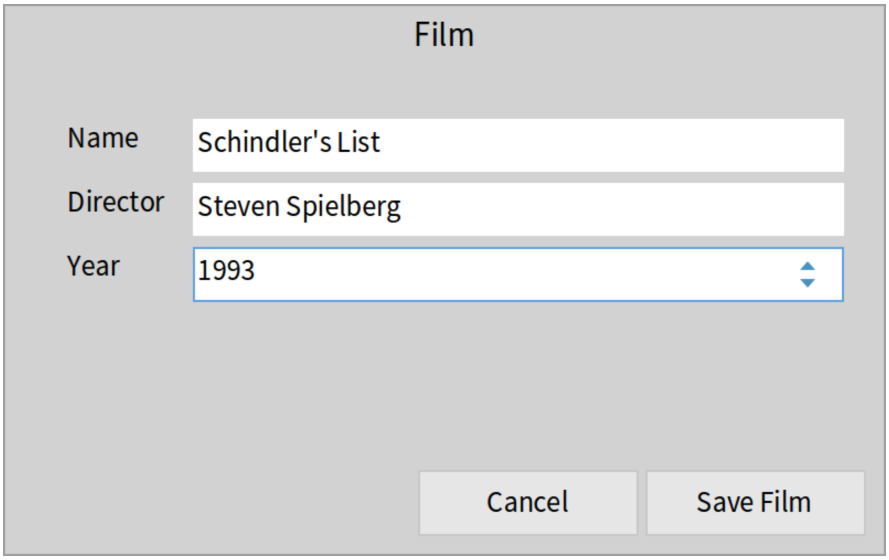
Steven Spielberg

1993

Figure 3-7 A modal dialog. %width=50&anchor=dialog

the dialog on screen by sending `centered` to the dialog presenter.

```
ImdbFilmPresenter >> initializeDialogWindow: aDialogPresenter

  aDialogPresenter centered.
  aDialogPresenter
    addButton: 'Cancel' do: [ :button | button close ];
    addButton: 'Save Film' do: [ :button | button beOk; close ].
```


Film

Name: Schindler's List

Director: Steven Spielberg

Year: 1993

Figure 3-8 Customizing the dialog window. % width=60&anchor=Customizeddialog

3.12 Invoking a presenter

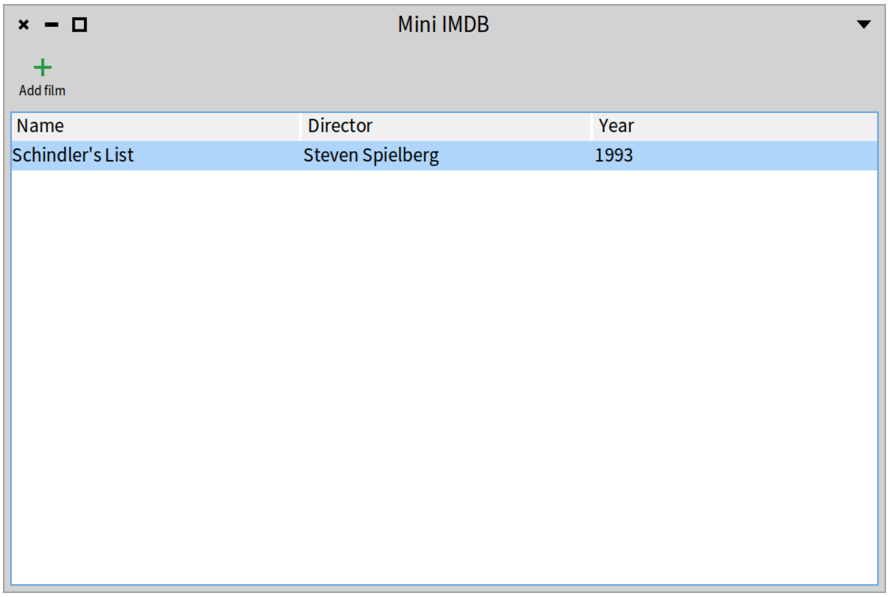
We are ready to use the film presenter from within the film list presenter. We define the method `addFilm` in the class `ImdbFilmListPresenter`. When the user clicks on the button, we create a new film presenter that we associate with the current application.

We open the film presenter as a modal dialog using the message `openModal`. When the user presses the "Save Film" button, a new film is added to our little database and we update the list.

```
ImdbFilmListPresenter >> addFilm

  | dialog windowPresenter film |
  dialog := ImdbFilmPresenter newApplication: self application.
  windowPresenter := dialog openModal.
  windowPresenter isOk ifFalse: [ ^ self ].

  film := ImdbFilm new
    name: dialog name;
    director: dialog director;
    year: dialog yearNumber.
  ImdbFilm addFilm: film.
  self updatePresenter
```

Now we can open the `FilmListPresenter` and click on the `Add film` button. When the film data has been entered and the `Save Film` button has been clicked, you will see that the FilmListPresenter is updated with the added film, as shown in Figure **??**.

```
app := ImdbApp new.
(app newPresenter: ImdbFilmListPresenter) open
```



Name	Director	Year
Schindler's List	Steven Spielberg	1993

Figure 3-9 The refreshed film list. % width=60&anchor=refreshed

3.13 Embedding a FilmPresenter into the FilmListPresenter

We have two main visual elements: a list of films and the film details. We can imagine that we would like to see the film details in the same container as the list, especially because a film description is larger than the list columns.

To achieve that, we add a new instance variable named `detail` to the class `ImdbFilmListPresenter`.

```
SpPresenter << #ImdbFilmListPresenter
  slots: { #filmList . #detail };
  package: 'CodeOfSpec20Book'
```

We redefine the default layout. We will show later that we can have different layouts.

```
ImdbFilmListPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    add: filmList;
    add: detail;
    yourself
```

Since we are going to use this presenter in different places, we have to add a method to control whether it is editable or not:

```
ImdbFilmPresenter >> editable: aBoolean

  nameText editable: aBoolean.
  directorText editable: aBoolean.
  yearNumber editable: aBoolean
```

Now we improve the `initializePresenters` of `ImdbFilmListPresenter`.

- First we instantiate `ImdbFilmPresenter`.
- Second, we configure it as read-only by sending the `editable: false` message.
- Third, when an element of the list is selected, we should display the information in the detail presenter. While we can express this in the `initializePresenters` method, we prefer specifying it in the `connectPresenters` method. See Section 3.14.

```
ImdbFilmListPresenter >> initializePresenters

  filmList := self newTable
    addColumn: (SpStringTableColumn title: 'Name'
      evaluated: #name);
    addColumn: (SpStringTableColumn title: 'Director'
      evaluated: #director);
    addColumn: (SpStringTableColumn title: 'Year'
      evaluated: #year);
    yourself.
  detail := self instantiate: ImdbFilmPresenter.
  detail editable: false
```

3.14 Define component communication

We add a helper method named `setModel:` in class `ImdbFilmPresenter` to be able to pass a film and populate the presenter accordingly.

```
ImdbFilmPresenter >> setModel: aFilm

  aFilm
    ifNil: [
      nameText text: ''.
      directorText text: ''.
      yearNumber number: '' ]
    ifNotNil: [
      nameText text: aFilm name.
      directorText text: aFilm director.
      yearNumber number: aFilm year ]
```

It is important to check for a `nil` value, otherwise sending `name`, `director`, or `year` would fail. If the given `aFilm` argument is `nil`, we clear the three subpresenters.

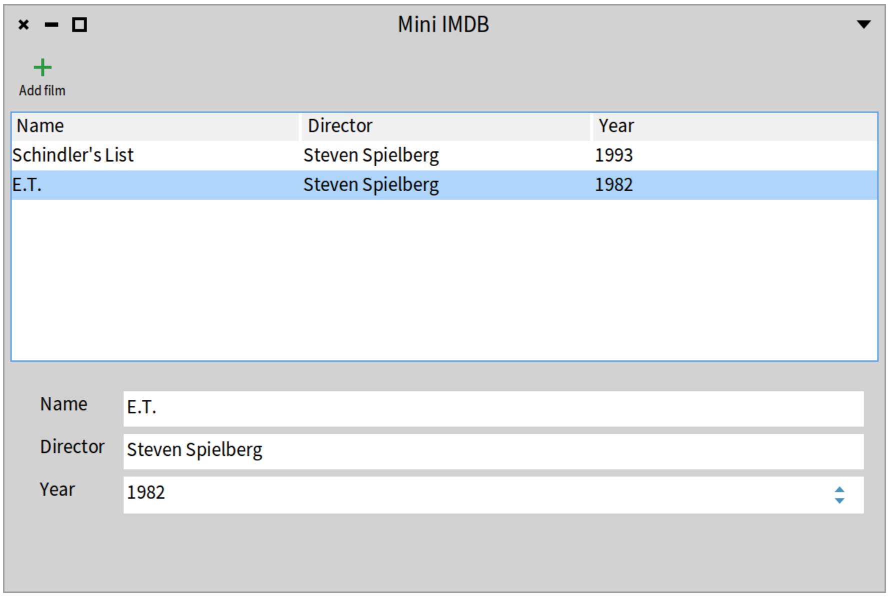
Note that the method `setModel:` is needed only if you do not subclass from `SpPresenterWithModel`. If you subclass from `SpPresenter`, it is the only way to have the model initialized before the setup of the presenter, and avoid errors when opening the presenter.

Defining interactions between presenters is done in the `connectPresenters` method. We implement it to define that, when an element of the list is selected, we display the information in the detail presenter. It is worth taking some time to look at the `whenSelectionChangedDo:` method.

The `whenSelectionChangedDo:` method expects a block with at most one argument. The argument does not hold the selected item directly, but a more complex object that represents the selection. Indeed a selection is different in a single selection list and a multiple selection list. Therefore Spec defines the concept of selection mode under the form of subclasses of `SpAbstractSelectionMode`.

```
ImdbFilmListPresenter >> connectPresenters

filmList whenSelectionChangedDo: [ :selectedItemMode |
  detail setModel: selectedItemMode selectedItem ]
```

With `connectPresenters` in place, selecting an item in the list results in showing the details of the selected item, as shown in Figure **??**.



Name	Director	Year
Schindler's List	Steven Spielberg	1993
E.T.	Steven Spielberg	1982

Name: E.T.
Director: Steven Spielberg
Year: 1982

Figure 3-10 Embedding the film description in the list: selecting a list item populates the detailed visual component. % width=60&anchor=embedded

3.15 Testing your application UI

A strong property of Spec is that we can write tests to describe the interaction and the logic of a UI. Tests are so powerful to help us create nice designs and make sure that we can spot errors, that we will show that writing tests for a UI is not complex.

We define `ImdbFilmListPresenterTest` as a subclass of `TestCase`.

```
TestCase << #ImdbFilmListPresenterTest
  package: 'CodeOfSpec20Book'
```

```
ImdbFilmListPresenterTest >>
    testWhenSelectingOneFilmThenDetailIsUpdated

  | presenter detail |
  "Arrange"
  presenter := ImdbFilmListPresenter new.
  presenter open.
  detail := presenter detail.
  self assert: detail name isEmpty.

  "Act"
  presenter clickFilmAtIndex: 1.

  "Assert"
  self deny: detail name isEmpty.
  presenter delete
```

As you see, we will have to define two methods on `ImdbFilmListPresenter` to support proper testing: a getter for `detail` and an interaction method `clickFilmAtIndex:`. We categorize them in the `testing - support` protocol to indicate that they are only intended for testing purposes.

```
ImdbFilmListPresenter >> detail

  ^ detail

ImdbFilmListPresenter >> clickFilmAtIndex: anIndex

  filmList clickAtIndex: anIndex
```

This test is a bit poor because we do not explicitly test the value of the film's name in the `detail` presenter. We did this to keep the test setup simple, partly because `ImdbFilm` stores the current films globally. Singletons are ugly and they also make testing more complex.

We define three helper methods on `ImdbFilm` to reset the stored films and add the E.T. film.

```
ImdbFilm class >> reset

  films := OrderedCollection new
```

```
ImdbFilm class >> addET

  films add: self ET
```

```
ImdbFilm class >> ET

  ^ self new
    name: 'E.T.';
    director: 'Steven Spielberg';
    year: '1982';
    yourself
```

Now we can define the `setUp` method.

```
ImdbFilmListPresenterTest >> setUp

  super setUp.
  ImdbFilm reset.
  ImdbFilm addET
```

Now we update the test to keep the opened presenter in an instance variable. This allows us to define a `tearDown` method that always closes the presenter, no matter if the test succeeds or fails.

```
ImdbFilmListPresenterTest >>
    testWhenSelectingOneFilmThenDetailIsUpdated

  | detail |
  "Arrange"
  presenter := ImdbFilmListPresenter new.
  presenter open.
  detail := presenter detail.
  self assert: detail name isEmpty.

  "Act"
  presenter clickFilmAtIndex: 1.

  "Assert"
  self deny: detail name isEmpty
```

```
ImdbFilmListPresenterTest >> tearDown

  presenter ifNotNil: [ presenter delete ].
  super tearDown
```

3.16 Adding more tests

Tests are addictive because we can change programs and check that they still work and limit our stress. So we will write another one.

Let us add the following getter method to support our tests.

```
ImdbFilmListPresenter >> filmList

  ^ filmList
```

Let us test that a list has one film and that if we select a non-existent index, the name is cleared.

```
ImdbFilmListPresenterTest >> testNoSelectionClearsDetails

  | name |
  "Arrange"
  presenter := ImdbFilmListPresenter new.
  presenter open.

  "Act"
  presenter clickFilmAtIndex: 1.

  "Assert"
  name := presenter detail name.
  self deny: name isEmpty.
  self assert: presenter filmList listSize equals: 1.

  presenter clickFilmAtIndex: 2.
  self assert: presenter detail name equals: ''
```

Multiple selection is not supported. Therefore we test that `filmList` is configured for single selection. There is no `isSingleSelection` method, so instead of asserting single selection, we deny multiple selection.

```
ImdbFilmListPresenterTest >> testListIsSingleSelection

  presenter := ImdbFilmListPresenter new.
  presenter open.
  self deny: presenter filmList isMultipleSelection
```

What you see is that it is relatively simple to test that the interaction you specified actually works as expected.

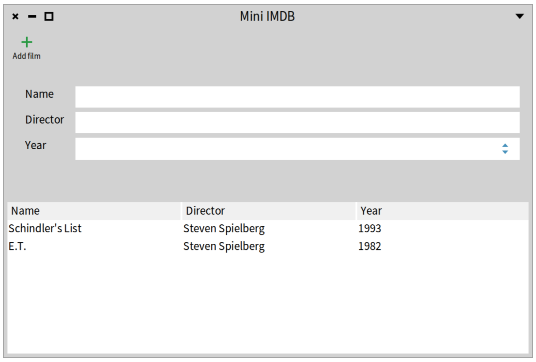
3.17 Changing layout

With Spec, a presenter can have multiple layouts, even layouts that are created on the fly as we will see with dynamic layouts. We can decide which layout to use when opening a presenter. Let us illustrate that. Imagine that we prefer to have the list positioned below the film details, or just the list alone.

```
ImdbFilmListPresenter >> listBelowLayout

  ^ SpBoxLayout newTopToBottom
    add: detail;
    add: filmList;
    yourself
```

The following example shows that we can open `ImdbFilmListPresenter` with the layout `listBelowLayout` that we just defined. See Figure 3-11.

```
| app presenter |
app := ImdbApp new.
presenter := app newPresenter: ImdbFilmListPresenter.
presenter openWithLayout: presenter listBelowLayout.
```



Name	Director	Year
Schindler's List	Steven Spielberg	1993
E.T.	Steven Spielberg	1982

Figure 3-11 A presenter can have multiple layouts for its subpresenters.

We can also define a layout with a part of the subpresenters. Here `listOnlyLayout` only shows the list.

```
ImdbFilmListPresenter >> listOnlyLayout

  ^ SpBoxLayout newTopToBottom
    add: filmList;
    yourself
```

The following example shows that we can open `ImdbFilmListPresenter` with one layout and dynamically change it by another layout. In a playground, do not declare the temporary variables so that they are bound and kept in the playground.

```
app := ImdbApp new.
presenter := app newPresenter: ImdbFilmListPresenter.
presenter open
```

The presenter opens with the default layout. Now in the playground execute the following line.

```
presenter layout: presenter listOnlyLayout
```

Now you can see that the layout with only one list has been applied dynamically.

3.18 Using transmissions

Spec 2.0 introduces a nice concept to propagate selections from one presenter to another, thinking about the "flow" of information more than the implementation details of this propagation, which can change from presenter to presenter.

With transmissions, each presenter can define a set of output ports (ports to transmit information) and input ports (ports to receive information). Widget presenters already have defined the output/input ports you can use with them, but you can add your own ports to your presenters.

The easiest way to declare a transmission is by sending the `transmitTo:` message from one presenter to another. We can now change the `connectPresenters` method to use transmissions.

```
ImdbFilmListPresenter >> connectPresenters

  filmList transmitTo: detail
```

Here, `filmList` is a table that will transmit its selection to the `detail` presenter.

Let us explain a bit. `ImdbFilmPresenter` is a custom presenter. Spec does not know how to "fill" it with input data. We need to tell Spec that `ImdbFilmPresenter` model will be the input port and receive the input data. Therefore we need to define an input port as follows:

```
ImdbFilmPresenter >> inputModelPort

  ^ SpModelPort newPresenter: self
```

```
ImdbFilmPresenter >> defaultInputPort

  ^ self inputModelPort
```

Note that we could have inlined `inputModelPort`'s definition into the `defaultInputPort` definition.

The input data will be set by using the `setModel:` method we already defined on `ImdbFilmPresenter`. `SpModelPort` takes care of that.

Now you can open the application and see that it still behaves as expected.

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmListPresenter) open
```

3.19 Styling the application

Different UI components in an application can have different look and feels, for example to change the size or color of a font for a header. To support this, Spec introduces the concept of "styles" for components.

In Spec, an application defines a stylesheet (or a set of them). A stylesheet defines a set of ”style classes” that can be assigned to presenter widgets. Defining a style class, however, works differently for each backend. While GTK accepts (mostly) regular CSS to style widgets, Morphic has its own subframework.

An application comes with a default configuration and a default stylesheet. If you do not need to style your application, there is no need to define them. In our example, we would like to define a `header` style to customize some labels. In Spec every presenter understands the message `addStyle:` that adds a tag (a CSS class) to the receiver.

To do so, you need to declare a stylesheet in a configuration. The configuration itself needs to be declared in your application. We will define a new presenter for the label and tag it with a specific CCS class using the message `addStyle:`. Our CCS class will be named `'customLabel'`.

First, we create the specific configuration for our application.

```
SpMorphicConfiguration << #ImdbConfiguration
  package: 'CodeOfSpec20Book'
```

Second, we use it in `ImdbApp`.

```
ImdbApp >> initialize

  super initialize.
  self
    useBackend: #Morphic
    with: ImdbConfiguration new
```

Then we can define our custom styles. The easiest way is to create a style from a String. Here we define that an element using the tag `customLabel` will have red text.

```
ImdbConfiguration >> customStyleSheet

  ^ '
.application [
  .customLabel [ Font { #color: #red } ] ]'
```

Pay attention not to forget the ’.’ in front of `application` and `customLabel`

We specialize the method `configure:` so that it includes the custom style as follows:

```
ImdbConfiguration >> configure: anApplication

  super configure: anApplication.
  self addStyleSheetFromString: self customStyleSheet
```

We are ready to use the tag for the label. Until now, Spec was creating a presenter for the label automatically, but it was not accessible by the developer. Therefore we have to add a label explicitly so that we can tag it with a CSS-like class. This is what the message `addStyle: 'customLabel'` below does.

We add a `nameLabel` instance variable to `ImdbFilmPresenter` to hold a label, and we initialize it in the method `initializePresenters` as follows:

```
ImdbFilmPresenter >> initializePresenters

  nameLabel := self newLabel
    label: 'Name';
    addStyle: 'customLabel';
    yourself.
  nameText := self newTextInput.
  directorText := self newTextInput.
  yearNumber := self newNumberInput
    rangeMinimum: 1900 maximum: Year current year;
    yourself
```

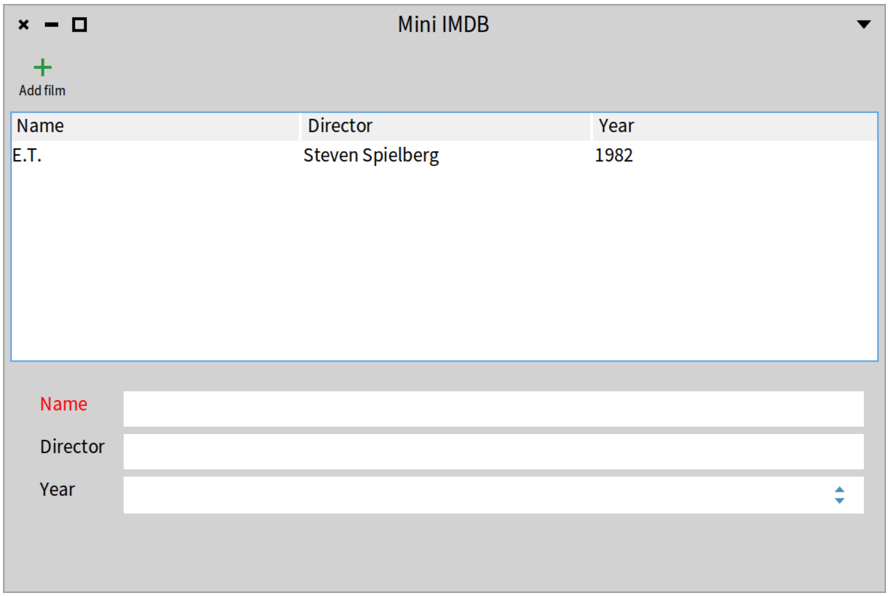
Then we update the layout to use the newly defined label presenter.

```
ImdbFilmPresenter >> defaultLayout

  ^ SpGridLayout build: [ :builder |
    builder
      beColumnNotHomogeneous;
      column:2 withConstraints: #beExpand;
      add: nameLabel; add: nameText; nextRow;
      add: 'Director'; add: directorText; nextRow;
      add: 'Year'; add: yearNumber ]
```

Now we see that the name label of a film detail has been styled, as shown in Figure **??**.

3.20 Conclusion

We saw that with Spec the developer defines how a visual element (a presenter) is composed of other visual elements. Such a presenter has the responsibility to describe the interaction with other presenters, but also with the domain objects. It has also the responsibility to describe its visual aspects.



Name	Director	Year
E.T.	Steven Spielberg	1982

Name	
Director	
Year	

Figure 3-12 Styled film name label. % width=60&anchor=FilmListPresenterStyled

Part II

Spec Essentials

CHAPTER 4

Spec core in a nutshell

Spec is Pharo's user interface framework. It provides the building blocks for constructing UIs, from simple windows to complex tools like browsers and debuggers. With Spec, developers can capture the layout and the interactions between the elements that compose a UI. For example, a developer can express that a tool has two components: a list on the left and a component displaying information on the right. Clicking on an item in the list will display detailed information about the selected item. In addition, Spec supports the reuse of the UI interaction logic.

Spec is the foundation of most tools in Pharo, such as the inspector, Spotter, the Pharo debugger, Iceberg, etc. In this short chapter, we place the key architectural elements of Spec in context.

4.1 Spec architecture overview

Figure **??** presents the general architecture of Spec. Basically, Spec is built around 5 concepts that we will describe in subsequent sections. The most important concepts are Presenter, Layout, and Application.

A Presenter represents the UI element logic and it is also the connection with the domain. The Application is also a place to be in contact with domain objects but generally, it handles application-specific resources (icons, windows,...).

Based on presenters and layouts, Spec builds the actual UI. Internally, it uses adapters that are specific to each widget and per backend. This way presenters are agnostic about backends and are reusable across them.

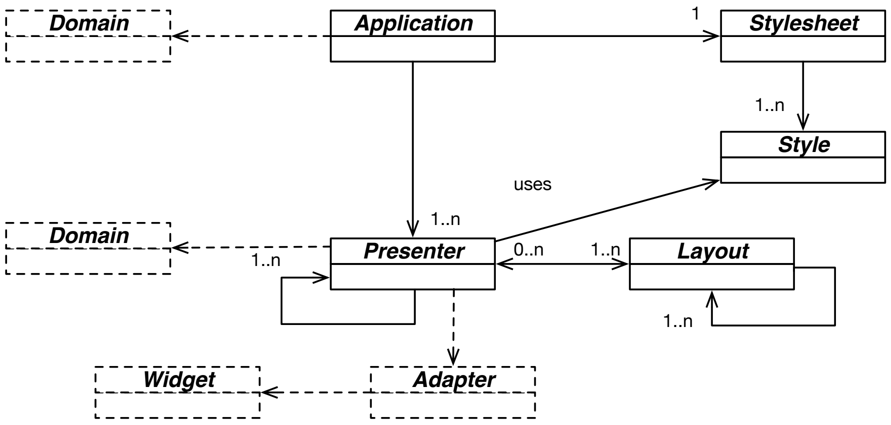


Figure 4-1 Architecture of Spec. % anchor=coreextended&width=80

4.2 Spec core architecture overview

Spec core is composed of the following elements:

- **Application.** An application is composed of multiple presenters and a stylesheet.
- **Presenters.** A presenter is a unit of interactive behavior. It is connected to domain objects and other presenters. Its visual representation is defined via at least one layout.
- **Layout.** A layout describes the positions of elements and it can be recursive.
- **Stylesheet and styles**. A stylesheet is composed of styles that describe visual properties such as fonts, colors, ...

We detail each of the main elements.

4.3 Presenters

A Spec presenter (an instance of a `SpPresenter` subclass), is an essential part of the Spec framework. It represents the logic of a UI element. It can define the behavior of a simple UI widget such as a button, as well as of a complex UI widget composed by many other presenters (either simple or complex). To build your user interface, you compose presenters.

Spec already comes with a predefined set of basic presenters (widgets) ready to use in your presenters. You can find them in the 'scripting - widgets' protocol

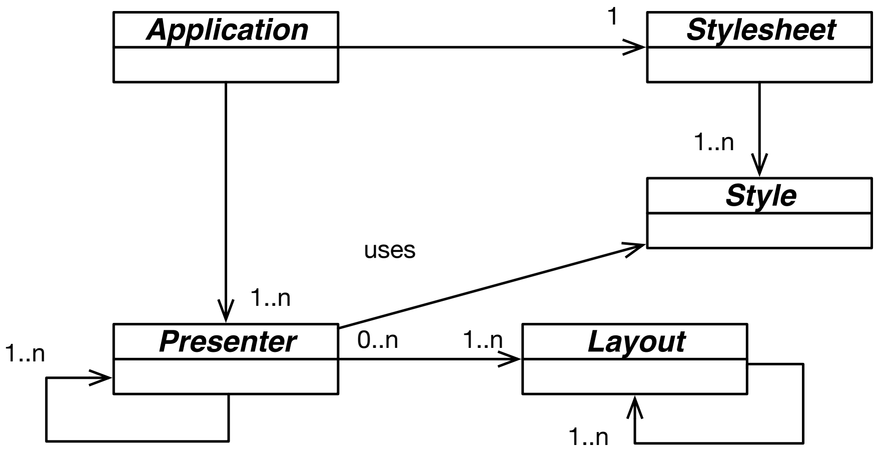


Figure 4-2 Presenter, Application, Layout and Style of Spec. % anchor=core&width=60

of the `SpPresenter` class. You will find buttons, labels, checkboxes, text input, drop lists, lists, menus, tables, trees, toolbars, action bars, but also more complex widgets like code diff presenters and notebooks. You can easily instantiate a new presenter and display it:

```
SpButtonPresenter new
  label: 'ok';
  open
```

A presenter may also have a model that is a domain object you need to interact with to display or update data. In this case, your presenter class should inherit from `SpPresenterWithModel` so that the presenter keeps a reference to the domain object and updates when the model changes (see Chapter 6).

A presenter defines layouts. One is mandatory. If you want to display a presenter with the default layout, you can use the `open` or `openDialog` methods. The former will open a new window with the presenter while the latter will open a blocking dialog with the presenter. You can use `openWithLayout:` or `openDialogWithLayout:` to open the presenter with the layout you will provide as an argument.

4.4 Application

A Spec application (an instance of the `SpApplication` class hierarchy) handles your application initialization, configuration, and resources. `SpAppli-`

cation is not a presenter because it does not have a graphical representation. An instance of SpApplication defines your application (keeping the backend, theme, icons, and other graphical resources), and keeps the opened windows that belong to the application, but it is not shown itself.

A Spec application also provides a way to access windows or resources such as icons, and provides abstractions for interactions with the user (inform, error, file, or directory selection).

Finally, an application provides the style used by Spec to style UI elements. A default style is available, but you can customize it as shown in Chapter 13.

You should also define a method to tell what is the main window / presenter to use when running the application. Here we specialize the method start as follows:

```
MyApplication >> start

  (MyMainPresenter newApplication: self) open
```

You can run your application with `MyApplication new run`. It will call the `start` method you defined.

4.5 Application configuration

In the application initialization, you can configure the backend you want to use: Morphic (default) or GTK. In the future, Spec will also support Toplo, a new widget library built on top of Bloc. It will replace Morphic.

Using Morphic

Here is an example using the Film application from Chapter 3. We define a configuration as a subclass of SpMorphicConfiguration.

```
SpMorphicConfiguration << #ImdbMorphicConfiguration
  package: 'CodeOfSpec20Book'
```

Then we define the method `configure:` as follows:

```
ImdbMorphicConfiguration >> configure: anApplication

  super configure: anApplication.
  "There are ways to write/read this from strings or files,
   but this is how you do it programatically."
  self styleSheet
    addClass: 'header' with: [ :style |
      style
        addPropertyFontWith: [ :font | font bold: true ];
        addPropertyDrawWith: [ :draw | draw color: Color red ] ]
```

Note that we could use a style described in a string as shown Chapter 13.

Finally, in the corresponding application class, we declare that the Morphic backend should use our configuration using the message `useBackend:with:`.

```
ImdbApp >> initialize

  super initialize.
  self useBackend: #Morphic with: ImdbMorphicConfiguration new
```

Using GTK theme and settings

For GTK the process is similar, we define a subclass of `SpGTKConfiguration`.

```
SpGTKConfiguration << #ImdbGTKConfiguration
  package: 'CodeOfSpec20Book'
```

Then we configure it by selecting and extending CSS.

```
ImdbGTKConfiguration >> configure: anApplication

  super configure: anApplication.
  "This will choose the theme 'Sierra-dark' if it is available"
  self installTheme: 'Sierra-dark'.
  "This will add a 'provider' (a stylesheet)"
  self addCSSProviderFromString: '.header {color: red; font-weight:
    bold}'
```

And in the application initialization, we declare that the configuration should be used for GTK.

```
ImdbApp >> initialize

  super initialize.
  self useBackend: #GTK with: ImdbGTKConfiguration new
```

4.6 Layouts

To display its elements, a presenter uses a layout. A layout describes how elements are placed on the display surface. To help you build nice user interfaces, several layouts are available:

- **GridLayout**: Choose this layout when you need to create a presenter with a label, and fields that need to be aligned (form style). You can specify in which box of the grid you want to place an element.

- **BoxLayout**: a `SpBoxLayout` arranges presenters in a box, vertically (top to bottom) or horizontally (left to right).
- **PanedLayout**: a `SpPanedLayout` is a layout with two elements called ”panes” and a splitter in between. The user can drag the splitter to resize the panes.
- **TabLayout**: a `SpTabLayout` shows all its elements as tabs. You can select a tab to display the content.
- **MillerLayout**: a layout to implement miller columns, also known as cascading lists (https://en.wikipedia.org/wiki/Miller_columns).

Any layout in Spec is dynamic and composable. In general, a layout is defined at the presenter instance level, but it can be defined on the class side.

Defining a layout is as simple as defining the `defaultLayout` method. This method is automatically invoked if a layout is not manually set.

Let’s revisit the `defaultLayout` method from Chapter 2.

```
CustomerSatisfactionPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    add: (SpBoxLayout newLeftToRight
          add: buttonHappy;
          add: buttonNeutral;
          add: buttonBad;
          yourself);
    add: result;
    yourself
```

The method defines two box layouts:

- one containing the three buttons
- one containing the first one and a result text below.

Each of the layouts refers to accessible subpresenters (`buttonHappy`, `buttonNeutral`, `buttonBad`, `result`) from the presenter. Figure **??** shows the corresponding result.

4.7 Styles and stylesheets

A Spec application always comes with a default stylesheet. A stylesheet contains style definitions that can be applied to presenters. Chapter 13 presents styles in detail.

A style is a property container to “style” components, and defines (to a certain degree) its behavior within the different layouts.

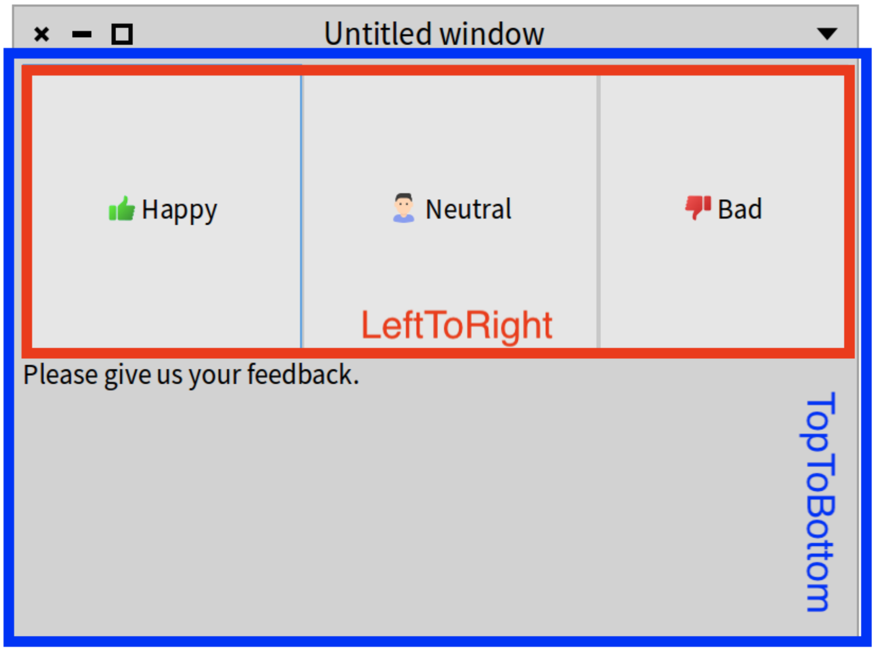


Figure 4-3 The layout corresponding to the `defaultLayout` method. % width=70&anchor=layout6B

Here is an example of a stylesheet for the Morphic backend:

```
'.application [
  .lightGreen [ Draw { #color: #B3E6B5 } ],
  .lightBlue [ Draw { #color: #lightBlue } ] ]'
```

The styles in Spec format are similar to CSS but expressed in STON. Pay attention not to forget the leading periods.

You can apply it on your Spec application by sending the `styleSheet:` message to an application:

```
myStyleSheet := SpStyleVariableSTONReader fromString:
  '.application [
    Font { #bold: true },
    .bgBlack [ Draw { #backgroundColor: #black } ],
    .blue [ Draw { #color: #blue } ]
]'
application styleSheet: SpStyle defaultStyleSheet, myStyleSheet.
```

Then you can style a presenter using the message `addStyle:` (think about a tag with a class in CSS) as follows:

```
presenter label: 'I am a label'.
presenter addStyle: 'blue'.
```

4.8 Navigation between presenters

Once the definition of your UI components (i.e., your Spec presenters and layouts) is done, you will need to define the behavior of the UI: what happens when you open a new presenter?

You will probably want to provide some data (a model) to the presenter so that it can be used to display data. It is called a transmission: you transmit data from one presenter to another presenter. Transmissions are defined as reactions to events.

It is quite easy to define the behavior of the UI by using widget-predefined events. You can find them in the api-events protocol of the presenter classes. Most used events are `whenSelectionChangedDo:`, `whenModelChangedDo:`, `whenTextChangedDo:`. Here are some examples:

```
messageList
  whenSelectionChangedDo: [ :selection |
    messageDetail model: selection selectedItem ];
  whenModelChangedDo: [ self updateTitle ].
textModel whenSubmitDo: [ :text | self accept: text ].
addButton action: [ self addDirectory ].
filterInput whenTextChangedDo: [ :text | self refreshTable ].
```

4.9 Conclusion

Class `SpPresenter` is a central class that has the following responsibilities:

- Initialization of presenter part and state.
- Definition of application layout.
- Connection of the elements to support the interaction flow.
- Update of the UI components.

We will illustrate these points in the following chapters.

CHAPTER 5

Testing Spec applications

Developers often think that testing a user interface is difficult. It is true that fully testing the placement and layout of widgets can be tedious. However, testing the logic of an application and in particular the interaction logic is possible. That is what we will show in this chapter. We will show that testing a Spec application is simple and effective.

5.1 Testing presenters

Tests are key to ensuring that everything works correctly. In addition, they free us from the fear of breaking something without being warned about it. Tests support refactorings. While such facts are general and applicable to many domains, they are also true for user interfaces.

Spec architecture

Spec is based on an architecture with three different layers as shown in Figure **??**:

- **Presenters:** Presenters define the interaction logic and manipulate domain objects. They access backend widgets but via an API that is specified by Adapters.
- **Adapters:** Adapters are objects exposing low-level backend widgets. They are a bridge between presenters and low-level widgets.
- **Backend widgets**. Backend widgets are plain widgets that can be used without Spec.

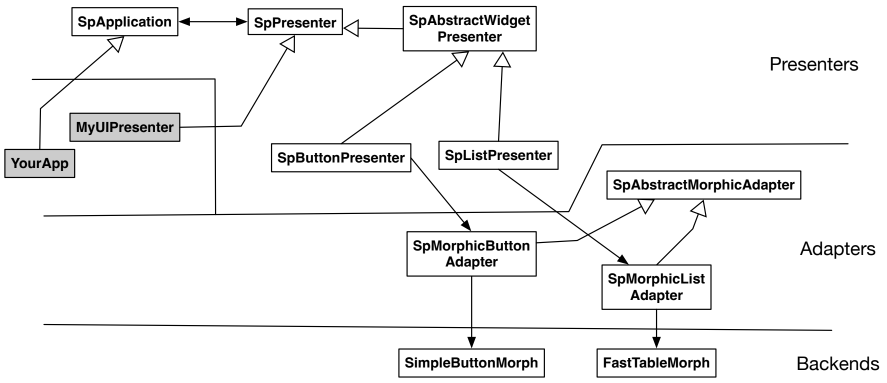


Figure 5-1 Spec Architecture: three layers Presenters - Adapters - Backends. % width=95&anchor=Architecture

Three roles and concerns

To help you understand the different possibilities of testing that you can engage in, we identify the following roles and their related concerns.

- **Spec Users.** Spec users are developers who build a new application. They define the logic of the application by assembling presenters and domain objects. We believe that this is the role that you will play most of the time.
- **Spec Developers.** Spec developers are more concerned with the development of new Spec presenters and their link with the adapters.
- **Widget Developers.** Widget developers are concerned about the logic and working of a given widget for a given backend.

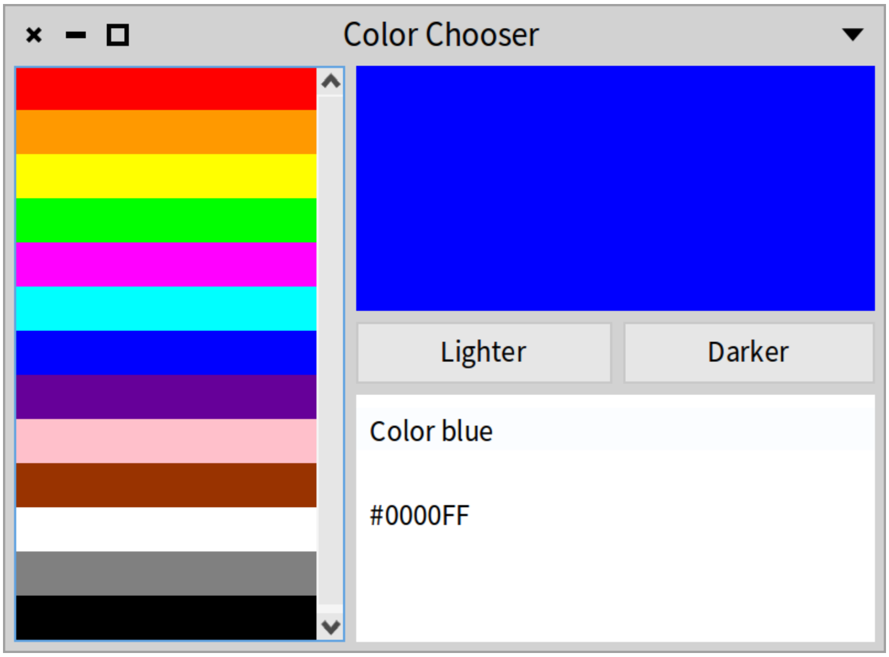
Spec user perspective

We will focus on the first role. For the reader interested in the second role, the class `SpAbstractBackendForTest` is a good starting place.

As a Spec user, you should consider that the backends are working and your responsibility is to test the logic of the user interface components. You should make sure that when the model changes, the user interface components reflect the changes. Inversely when the user interface components change, you should ensure that the model is updated. Let’s give an example.

5.2 Spec user example

We will test a simple spec application, as shown in Figure **??**. The model for this application is an instance of the `Color` class. The application shows a list of colors from which the user can choose one. After choosing a color, the application shows the color in a big box, and it shows the `printString` of the color, together with the hexadecimal code. The application also provides two buttons to make the chosen color lighter or darker.



Color blue

#0000FF

Figure 5-2 A Spec application. % width=70&anchor=exampleapplication

The presenter is defined as described below. The class has six instance variables. The first five instance variables hold subpresenters that compose the application window. The sixth instance variable holds the color that serves as the model of the application.

```
SpPresenter << #ColorChooser
  slots: { #colorList . #colorDetails . #colorBox . #lighterButton .
    #darkerButton . #currentColor };
  package: 'CodeOfSpec20Book'
```

The method `initializePresenters` initializes the subpresenters. `colorList` holds a list presenter with the colors. `colorBox` displays the chosen color in a

SpRoassalPresenter. colorDetails holds a text presenter that shows information about the color. lighterButton and darkerButton are the buttons to make the current color lighter or darker.

```
ColorChooser >> initializePresenters

  colorList := self newList
    display: [ :color | '' ];
    displayBackgroundColor: [ :color | color ];
    yourself.
  colorBox := self instantiate: SpRoassalPresenter.
  lighterButton := self newButton
    label: 'Lighter';
    action: [ self lighter ];
    yourself.
  darkerButton := self newButton
    label: 'Darker';
    action: [ self darker ];
    yourself.
  colorDetails := self newText
```

currentColor is not initialized by initializePresenters. It is initialized in setModelBeforeInitialization: because a color can be given when creating a new ColorChooser instance.

```
ColorChooser >> setModelBeforeInitialization: aColor

  currentColor := aColor
```

defaultLayout defines the layout with a left side and a right side. The left side is the color list. The right side consists of the color box, the two buttons, and the color details. Composition with horizontal and vertical BoxLayouts, together with a 5-pixel spacing, results in the window shown in Figure **??**.

```
ColorChooser >> defaultLayout

  | colorBoxAndDetails buttons |
  buttons := SpBoxLayout newLeftToRight
    spacing: 5;
    add: lighterButton;
    add: darkerButton;
    yourself.
  colorBoxAndDetails := SpBoxLayout newTopToBottom
    spacing: 5;
    add: colorBox;
    add: buttons expand: false;
    add: colorDetails;
    yourself.
  ^ SpBoxLayout newLeftToRight
    spacing: 5;
    add: colorList expand: false;
    add: colorBoxAndDetails;
    yourself
```

`initializeWindow:` sets the title and the initial dimensions of the window.

```
ColorChooser >> initializeWindow: aWindowPresenter

  aWindowPresenter
    title: 'Color Chooser';
    initialExtent: 400@294
```

Connecting the subpresenters is expressed easily. When a selection in the color list is made, the color is updated.

```
ColorChooser >> connectPresenters

  colorList whenSelectionChangedDo: [ :selection |
    self updateColor: selection selectedItem ]
```

`connectPresenters` delegates to `updateColor:` to update the color box and the color details. As you can see, `updateColor:` takes care of a possible `nil` value for `currentColor`.

```
ColorChooser >> updateColor: color

  | details |
  currentColor := color.
  colorBox canvas
    background: (currentColor ifNil: [ Color transparent ]);
    signalUpdate.
  details := currentColor
    ifNil: [ '' ]
    ifNotNil: [ self detailsFor: currentColor ].
  colorDetails text: details
```

`updateColor:` delegates the responsability of producing the text with color details to `detailsFor:`.

```
ColorChooser >> detailsFor: color

  ^ String streamContents: [ :stream |
    stream
      print: color; cr; cr; nextPut: $#;
      nextPutAll: color asHexString ]
```

We also define `updatePresenter` to set the initial state of the subpresenters. It populates the color list with default colors, as defined by `defaultColors`, and the initial color is set with `updateColor:`.

```
ColorChooser >> updatePresenter

  | initialColor |
  initialColor := currentColor.
  colorList items: self defaultColors.
  self updateColor: initialColor
```

Note that keeping the initial color with `initialColor := currentColor` is necessary because `colorList items: self defaultColors` resets the selection in the list, which triggers the block in `connectPresenters`. That block sends `updateColor: nil` because there is no selection. So this method keeps the initial color and applies it with `self updateColor: initialColor`.

To keep things simple, `defaultColors` answers only a handful of colors. This method can be changed easily to answer a different collection of colors. For instance, you could try `Color red wheel: 20`.

```
ColorChooser >> defaultColors

  ^ {
    Color red .
    Color orange .
    Color yellow .
    Color green .
    Color magenta .
    Color cyan .
    Color blue .
    Color purple .
    Color pink .
    Color brown .
    Color white .
    Color gray .
    Color black }
```

There are only two methods missing from the code above to complete the class implementation. `initializePresenters` sets actions for the buttons, which invoke the following two methods. These methods delegate to `updateColor:` to do the heavy lifting.

```
ColorChooser >> lighter

  self updateColor: currentColor lighter
```

```
ColorChooser >> darker

  self updateColor: currentColor darker
```

With the code above in place, we can open the application. Let's start with opening the default with:

```
ColorChooser new open
```

In this case, there is no initial color, which results in the window shown in Figure **??**. The color box does not show a color and the color details are empty.

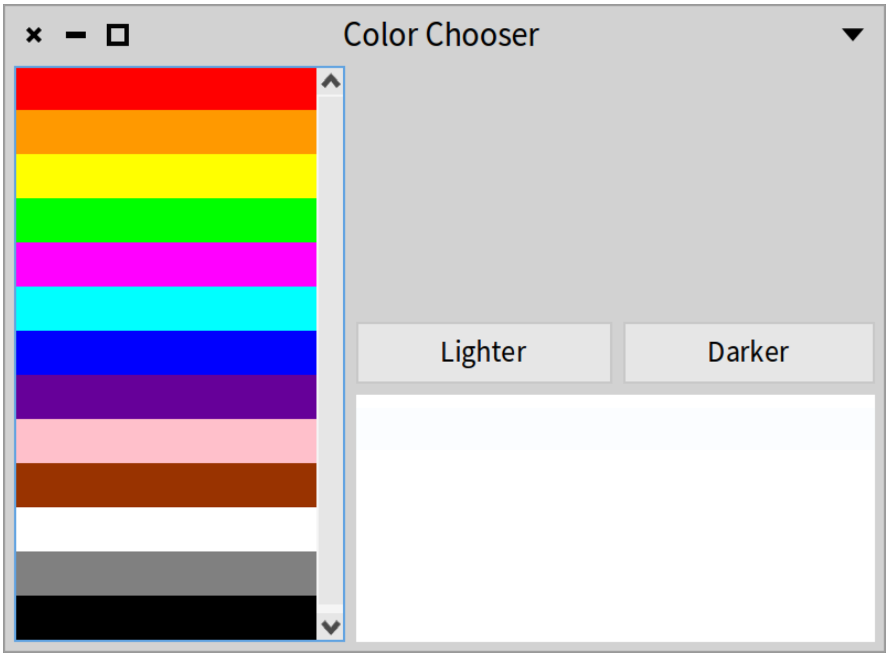


Figure 5-3 The default ColorChooser. % width=70&anchor=defaultapplication

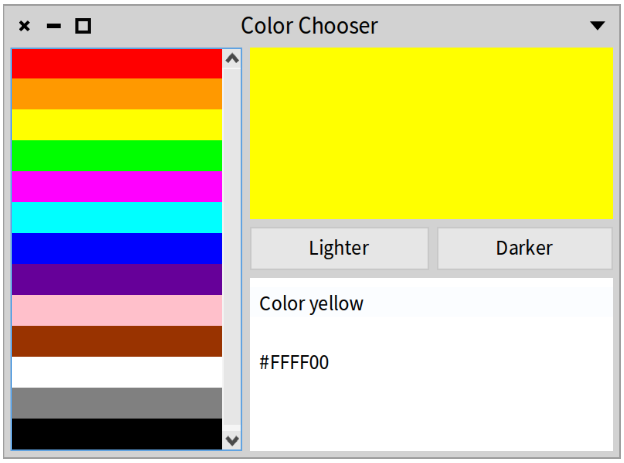
Let's see what happens when we provide a color with:

```
(ColorChooser on: Color yellow) open
```

In this case, yellow is given as the initial color that should be shown when the window opens. Note that `on:` has not been defined as a class method by `ColorChooser`. The class method is inherited from the superclass `SpAbstractPresenter`. The result is shown in Figure 5-4.

5.3 Tests

With all the code in place, it is time to write some tests. First, we define the test class.



Color yellow

#FFFF00

Figure 5-4 The ColorChooser opened on the color yellow.

```
TestCase << #ColorChooserTest
  slots: { #chooser };
  package: 'CodeOfSpec20Book'
```

Each test will open a new instance of `ColorChooser`. It is expected that the instance variable `chooser` will hold the instance used in a test. To ensure that the instance is cleaned up, we define `tearDown`. It takes into account that a test can fail before `chooser` is bound to an instance of `ColorChooser`.

```
ColorChooserTest >> tearDown

  chooser ifNotNil: [ chooser delete ].
  super tearDown
```

With that infrastructure in place, we can write our tests.

Opening the default application

Our first test describes the state of the application after opening the default application.

```
ColorChooserTest >> testDefault
  "When a ColorChooser opens without a color,
   the color box shows a transparent color and the details are empty."

  chooser := ColorChooser new.
  chooser open.
  self assert: chooser boxColor equals: Color transparent.
  self assert: chooser detailsText equals: ''
```

We have to add a few so-called ’test support’ methods to make this work. These methods belong to the test api of the `ColorChooser`, because they are intended to be used for testing purposes only.

```
ColorChooser >> boxColor

  ^ colorBox canvas color
```

```
ColorChooser >> detailsText

  ^ colorDetails text
```

Correct initialization

The second test describes the state of the application after opening the application with a color.

```
ColorChooserTest >> testInitialization
  "When a ColorChooser opens on a color,
   the color box shows that color
   and the details show the print string and the HEX code."

  chooser := ColorChooser on: Color palePeach.
  chooser open.

  self assert: chooser boxColor equals: Color palePeach.
  self assert: chooser detailsText equals: 'Color palePeach\\#FFEDD5'
    withCRs
```

Choosing a color

The third test describes what happens when the user chooses a color.

First, the test selects the first color in the list and verifies the state of the subpresenters. Then it selects the seventh color in the list and verifies the expected state changes in the subpresenters.

```
ColorChooserTest >> testChooseColor
  "When the user chooses a color in the list,
   the color box shows the color
   and the details show the print string and the HEX code."

  chooser := ColorChooser new.
  chooser open.

  chooser clickColorAtIndex: 1.
  self assert: chooser boxColor equals: Color red.
  self assert: chooser detailsText equals: 'Color red\\#FF0000'
    withCRs.

  chooser clickColorAtIndex: 7.
  self assert: chooser boxColor equals: Color blue.
  self assert: chooser detailsText equals: 'Color blue\\#0000FF'
    withCRs
```

This test uses an extra test support method to click on a color in the list.

```
ColorChooser >> clickColorAtIndex: index

  colorList clickAtIndex: index
```

Making the current color lighter

Now it is time to describe the application behavior after clicking the 'Lighter' button.

The test consists of four parts. First, the first color in the list is clicked. That results in an update of the color box and the color details. After a click on the button, the test verifies the changed state of the color box and the color details. Then it clicks the button a second time to describe that the current color can be made lighter over and over again. Finally, the test selects the seventh color in the list and verifies the expected state changes in the subpresenters.

```
ColorChooserTest >> testLighter
  "When the user presses the 'Lighter' button,
   the color box shows the ligher color
   and the details show the print string and the HEX code."

  chooser := ColorChooser new.
  chooser open.

  chooser clickColorAtIndex: 1.
  chooser clickLighterButton.
  self
    assert: chooser boxColor
    equals: (Color r: 1.0 g: 0.030303030303030304 b:
    0.030303030303030304 alpha: 1.0).
  self
    assert: chooser detailsText
    equals: '(Color r: 1.0 g: 0.030303030303030304 b:
    0.030303030303030304 alpha: 1.0)\\#FF0707' withCRs.

  chooser clickLighterButton.
  self
```

```
    assert: chooser boxColor
    equals: (Color r: 1.0 g: 0.06060606060606061 b:
     0.06060606060606061 alpha: 1.0).
  self
    assert: chooser detailsText
    equals: '(Color r: 1.0 g: 0.06060606060606061 b:
     0.06060606060606061 alpha: 1.0)\\#FF0F0F' withCRs.

  chooser clickColorAtIndex: 7.
  chooser clickLighterButton.
  self
    assert: chooser boxColor
    equals: (Color r: 0.030303030303030304 g: 0.030303030303030304 b:
     1.0 alpha: 1.0).
  self
    assert: chooser detailsText
    equals: '(Color r: 0.030303030303030304 g: 0.030303030303030304 b:
     1.0 alpha: 1.0)\\#0707FF' withCRs
```

As the other tests, this test requires an extra test support method.

```
ColorChooser >> clickLighterButton

  lighterButton click
```

Making the current color darker

This test is very similar to the previous test. Instead of clicking the 'Lighter' button, this test clicks the 'Darker' button.

```
ColorChooserTest >> testDarker
  "When the user presses the 'Darker' button,
   the color box shows the darker color
   and the details show the print string and the HEX code."

  chooser := ColorChooser new.
  chooser open.

  chooser clickColorAtIndex: 1.
  chooser clickDarkerButton.
  self
    assert: chooser boxColor
    equals: (Color r: 0.9198435972629521 g: 0.0 b: 0.0 alpha: 1.0).
  self
    assert: chooser detailsText
    equals: '(Color r: 0.9198435972629521 g: 0.0 b: 0.0 alpha:
     1.0)\\#EB0000' withCRs.
  chooser clickDarkerButton.
  self
    assert: chooser boxColor
    equals: (Color r: 0.8396871945259042 g: 0.0 b: 0.0 alpha: 1.0).
  self
    assert: chooser detailsText
    equals: '(Color r: 0.8396871945259042 g: 0.0 b: 0.0 alpha:
     1.0)\\#D60000' withCRs.

  chooser clickColorAtIndex: 7.
  chooser clickDarkerButton.
  self
    assert: chooser boxColor
    equals: (Color r: 0.0 g: 0.0 b: 0.9198435972629521 alpha: 1.0).
  self
    assert: chooser detailsText
    equals: '(Color r: 0.0 g: 0.0 b: 0.9198435972629521 alpha:
     1.0)\\#0000EB' withCRs
```

Again, this test requires an extra test support method.

```
ColorChooser >> clickDarkerButton

  darkerButton click
```

Verifying window properties

Now we want to check that the window is built correctly. We will verify that the title and the initial extent of the window are correct.

```
ColorChooserTest >> testInitializeWindow

  | window |
  chooser := ColorChooser new.
  window := chooser open.
  self assert: window isBuilt.
  self assert: window title equals: 'Color Chooser'.
  self assert: window initialExtent equals: 400@294
```

5.4 Testing your application

In Spec, an application is responsible to run and gather the windows of your application. The pattern is to override the `start` method of your application. The method `start` is a hook method that is invoked when you execute your application using the `run` message as in `ColorChooserApplication new run`.

It is important to see that in the `start` method you should configure the presenter you are opening so that it knows its application. This is important so that the application knows the windows it is opening.

In a TDD fashion, we define the test class first:

```
TestCase << #ColorChooserApplicationTest
  slots: { #application };
  package: 'CodeOfSpec20Book'
```

```
ColorChooserApplicationTest >> setUp

  super setUp.
  application := ColorChooserApplication new
```

```
ColorChooserApplicationTest >> tearDown

  application ifNotNil: [ application closeAllWindows ].
  super tearDown
```

```
ColorChooserApplicationTest >> testWindowRegistration

  self assert: application windows size equals: 0.
  application start.
  self assert: application windows size equals: 1.
  application start.
  self assert: application windows size equals: 2
```

`testWindowRegistration` describes the expected behaviour of our application. When opened windows are correctly registered, the application should have access to all the opened windows. The test opens two windows and verifies that the number of windows increases.

The test fails, because `ColorChooserApplication` does not exist yet. Let's define it:

```
SpApplication << #ColorChooserApplication
  slots: {};
  package: 'CodeOfSpec20Book'
```

The test still fails. It fails in the second assert because the application does not register the open windows. Let's implement the `start` method to register the windows.

```
ColorChooserApplication >> start

  ColorChooser new
    application: self;
    open
```

Tada! The test passes.

5.5 Known limitations and conclusion

In this chapter we showed that you can take advantage of Spec to define tests that will help you to evolve the visual part of your application. This is really key for modern software development and to lower your stress in the future. So take advantage of agile development.

Currently, Spec does not offer a way to script and control popup windows. It is not possible to script a button that opens a dialog for a value. Future versions of Spec should cover this missing feature.

CHAPTER 6

The dual aspects of presenters: Domain and interaction model

A presenter has a dual role in Spec. On the one hand, it acts as the glue between domain objects and widgets, and on the other hand, it implements the user interface logic by connecting subpresenters together. These two aspects compose the core of a presenter and this is what this chapter describes.

We start by presenting an important aspect of presenters: the way they handle communication with domain objects that here we call a model.

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

6.1 About presenters on a model

Frequently you want to open a presenter on a given object such as your list of to-do items. In that case, you would like the subpresenters (list, text,..) to be initialized based on the object that you passed. For example, you may want to get all the items in your basket.

However, simply instantiating a presenter using the message `new` and passing the object will not work because messages such as `initializePresenters` will be already sent.

There are two ways to address this situation in Spec and in particular, Spec offers a special presenter called `SpPresenterWithModel`. Let us explain how to take advantage of it.

We will build the simplest example to show how to do it. We will implement a presenter that lists the method signatures of a class, first using a presenter inheriting from the default superclass (`SpPresenter`) and second using a presenter (subclass of `SpPresenterWithModel`) dedicated to handling a model.

6.2 Example with SpPresenter

If you do not need to react to model changes, you can simply inherit from `SpPresenter`, override the `setModelBeforeInitialization:` method to set your domain object, and use `YourPresenter on: yourDomainObject` to instantiate it.

This is exactly what we do hereafter.

First, we create a new presenter class.

```
SpPresenter << #MethodLister
  slots: { #sourceClass . #list};
  package: 'Spec2Book'
```

We define a list presenter and populate it.

```
MethodLister >> initializePresenters

  list := self newList.
  list items: sourceClass selectors sorted
```

Specializing the method `setModelBeforeInitialization:`, we assign its argument coming from the `on:` message to the instance variable `sourceClass` for future use.

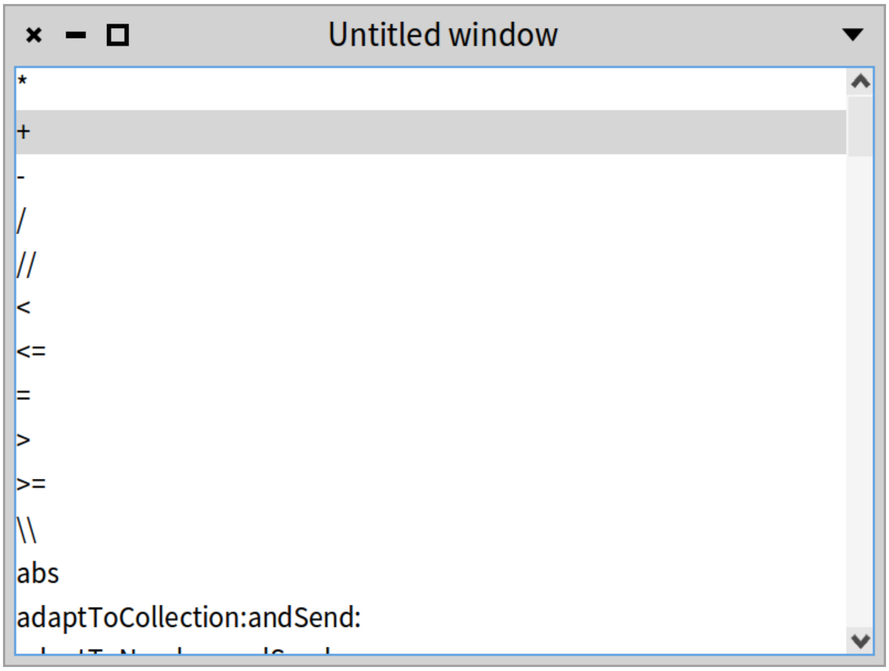
```
MethodLister >> setModelBeforeInitialization: aModel

  sourceClass := aModel
```

We define a basic layout for the list presenter.

```
MethodLister >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: #list;
      yourself
```

The following snippet opens a window with the list of methods of the class `Point` as shown in Figure **??**.

```
(MethodLister on: Point) open.
```



```
*
+
-
/
//
<
<=
=
>
>=
\\
abs
adaptToCollection:andSend:
```

Figure 6-1 A simple list of sorted selectors of the class Point. %anchor=pointselectors&width=45

6.3 SpPresenter vs. SpPresenterWithModel

The key difference between using `SpPresenter` and `SpPresenterWithModel` is if you need to react to changes of the model. We mean that while the presenter is open, an event changes the model that was used to build the UI. In our example, that means that when you change the class, the method list displays its selectors. If you need this behavior, then you should use `SpPresenterWithModel`.

The following snippet shows that the change of model is not taken into account in the sense that the list is not refreshed and still displays methods of the class `Point`, while the methods of the class ‘Rectangle should be displayed.

```
| lister |
lister := MethodLister on: Point.
lister open.
lister class: Rectangle
```

6.4 Example with SpPresenterWithModel

A presenter may also have a model that is a domain object you need to interact with to display or update data. In that case, you should inherit from SpPresenterWithModel so that the presenter keeps a reference to the domain object and manages its changes. As a client of this presenter, we use the message model: to change the model.

The method is inherited from the superclass. This model: method implements the following behavior:

- If the domain object is an instance of Model, it is stored as is in the presenter.
- Else a value holder is created to hold the domain object so that you can be notified when the domain object used by the presenter changes.

You do not need to define the method setModelBeforeInitialization: as we previously showed.

Let us revisit our little example. First, we inherit from SpPresenterWithModel.

```
SpPresenterWithModel << #MethodListerWithModel
  slots: { #list };
  package: 'Spec2Book'
```

Second, we define initializePresenters.

```
MethodListerWithModel >> initializePresenters

  list := self newList
```

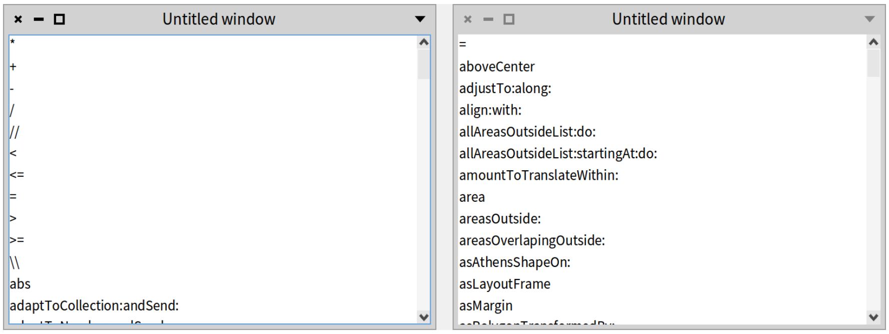
You can then implement the modelChanged method to refresh your UI when the model changes.

```
MethodListerWithModel >> modelChanged

  list items: self model selectors sorted
```

We define the same layout method as before:

```
MethodListerWithModel >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: #list;
      yourself
```

Now we can open our widget. As the following script shows, it will react to the change of the model (see Figure **??**).



```
*
+
-
/
//
<
<=
=
>
>=
\\
abs
adaptToCollection:andSend:
```

```
=
aboveCenter
adjustTo:along:
align:with:
allAreasOutsideList:do:
allAreasOutsideList:startingAt:do:
amountToTranslateWithin:
area
areasOutside:
areasOverlapingOutside:
asAthensShapeOn:
asLayoutFrame
asMargin
```

Figure 6-2 A simple list of sorted selectors changing based on its model. %anchor=pointRectangeSelectors&width=90

```
| lister |
lister := MethodListerWithModel on: Point.
lister open.
lister model: Rectangle
```

Note that the right way to create a presenter is to use the method `newApplication: anApplication` because it ensures that the application knows its constituents.

So the code above should be:

```
| lister app |
app := SpApplication new
lister := MethodListerWithModel newApplication: app.
```

Then we have a problem because we want to specify the model too. The correct and idiomatic way is to use the method `newApplication:model:` so the final code version is:

```
| lister |
app := SpApplication new.
lister := MethodListerWithModel newApplication: app model: Point.
lister open.
lister model: Rectangle
```

You saw that you can easily build an application user interface populated from a model and reacting to model changes.

Now we will focus on the user interface logic modeling.

6.5 User interface building: a model of UI presentation

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 presentation model and the UI elements that make up the resulting user interface that is shown. This composition of the presentation models is represented as a Presenter object as in Model-View-Presenter. The presenter that is defined in Spec corresponds to a presenter in the MVP triad as shown in Figure **??**.

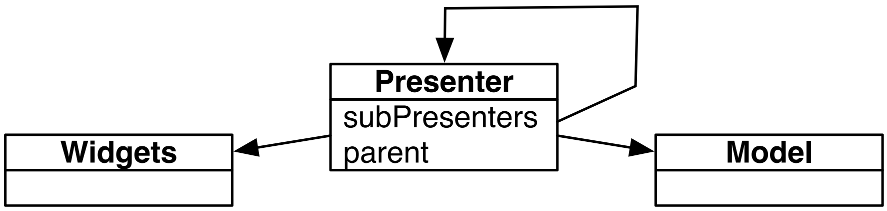


Figure 6-3 A presenter is a model of presentation: It is in relationships with the widgets and its domain model. It composes other presenters to form a presenter tree. %anchor=mvpfig&width=60

To define a new user interface, the developer should create a subclass of `SpPresenter`.

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

- The method `initializePresenters` treats the subpresenters themselves.
- The method `connectPresenters` treats the interactions between the subpresenters.
- The method `defaultLayout` treats the layout of the subpresenters.

Hence, these methods are typically found in the model of each user interface. You can read the code of the small interface presented in Chapter 2 to get examples of each of the points we will present now.

In this chapter, we describe the finer points of each method and how these three methods work together to build the overall UI.

6.6 The *initializePresenters* method

The method `initializePresenters` instantiates, holds in instance variables, and partially configures the different widgets that will be part of the UI.

The instantiation of the presentation models will cause the instantiation and initialization of the different lower-level user interface components, constructing the UI that is shown to the user. The first part of the configuration of each widget is specified in `initializePresenters` as well.

The focus of this method is to specify what the widgets will look like and what their self-contained behavior is. The behavior to update the 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 `initializePresenters` method should follow the pattern:

- Widget instantiation
- Widget configuration
- Specification of focus order

The last step is not mandatory since the focus order is by default given by the order of declaration of the subpresenters.

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

Subpresenter instantiation

The instantiation of a subpresenter (i.e., the model for a widget composing the UI) 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 `SpPresenter` in the protocol `scripting - widgets`.
- The second option is more general: to reuse a `SpPresenter` 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` presenter, the code is `self instantiate: MessageBrowser`. The `instantiate:` method has the responsibility to build an internal parent presenter tree.

6.7 The *connectPresenters* method

The method `connectPresenters` 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 `connectPresenters` is an optional method for a Spec UI, but we recommend to separate this behavior clearly.

In Spec, the different UI models are contained in value holders, and the event mechanism relies on the announcements from 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 message 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 (`whenSelectionChangedDo:`).

6.8 The *defaultLayout* method

Widget layout is defined by specifying methods that state how the different widgets are placed in the UI. 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.

The method `defaultLayout` is an instance method, but it can be also defined at the class level. Put differently, typically all the instances of the same user interface have the same layout, but a layout can be specific to one instance and be dynamic.

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

Using setter message `layout:`

We recommend to clearly separate presenter initialization (`initializePresenters` and `defaultLayout`). You can, however, also use the `layout:` message to set a layout during the presenter initialization phase.

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 `open` (as we have done until now), use the `openWithLayout:` message with a layout as an argument.

6.9 Conclusion

In this chapter, we have given a more detailed description of how the three fundamental methods of Spec, `initializePresenters`, `defaultLayout`, and `connectPresenters`, are each responsible for a different aspect of the user interface building process.

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

CHAPTER 7

Reuse and composition at work

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

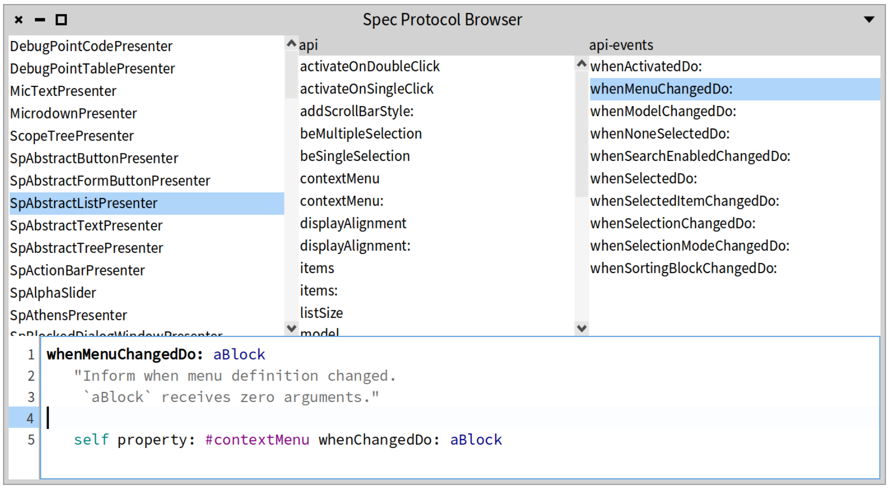
This focus on reuse was actually already visible in the previous chapters, where we have seen that basic widgets can be used as if they were complete user interfaces. In this section we focus on the reuse and composition of presenters, showing that it basically comes for free. The only requirement when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

Said differently, in this chapter, you will learn how you can build a new UI by reusing already defined elements.

7.1 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 **??** as a composition of four parts:

1. The **WidgetClassListPresenter**: a widget containing a `SpListPresenter` specifically for displaying the subclasses of `SpAbstractWidgetPresenter`.
2. The **ProtocolMethodListPresenter**: a widget composed of a `SpListPresenter` and a `SpLabelPresenter` for displaying methods of a protocol.



```
whenMenuChangedDo: aBlock
   "Inform when menu definition changed.
    `aBlock` receives zero arguments."

   self property: #contextMenu whenChangedDo: aBlock
```

Figure 7-1 ProtocolCodeBrowser: Browsing the public APIs of widgets. %width=80&anchor=figprotocolbrowser&label=figprotocolbrowser

3. The **ProtocolViewerPresenter**: a composition of one `WidgetClassListPresenter` and two `ProtocolMethodListPresenter`. It allows browsing the methods of all subclasses of `SpAbstractWidgetPresenter`.
4. The **ProtocolCodeBrowserPresenter**: reuses a `ProtocolViewerPresenter`, changes its layout, and adds a `SpTextPresenter` to see the source code of the methods.

7.2 Creating a basic UI to be reused as a widget

The first UI we build displays a list of all subclasses of the class `SpAbstractWidgetPresenter`. This UI will later be reused as a widget for a more complete UI. The code is as follows.

First, we create a subclass of `SpPresenter` with one instance variable `list` which will hold an instance of `SpListPresenter`.

```
SpPresenter << #WidgetClassListPresenter
  slots: { #list };
  package: 'CodeOfSpec20Book'
```

In the method `initializePresenters`, we create the list and populate it with the required classes, in alphabetical order.

```
WidgetClassListPresenter >> initializePresenters

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

We also add a title for the window.

```
WidgetClassListPresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter title: 'Widgets'
```

The layout contains only the list.

```
WidgetClassListPresenter >> defaultLayout

  ^ SpBoxLayout newLeftToRight
      add: #list;
      yourself
```

When doing `WidgetClassListPresenter new open`, you should see the UI shown in Figure **??**.

7.3 Supporting reuse

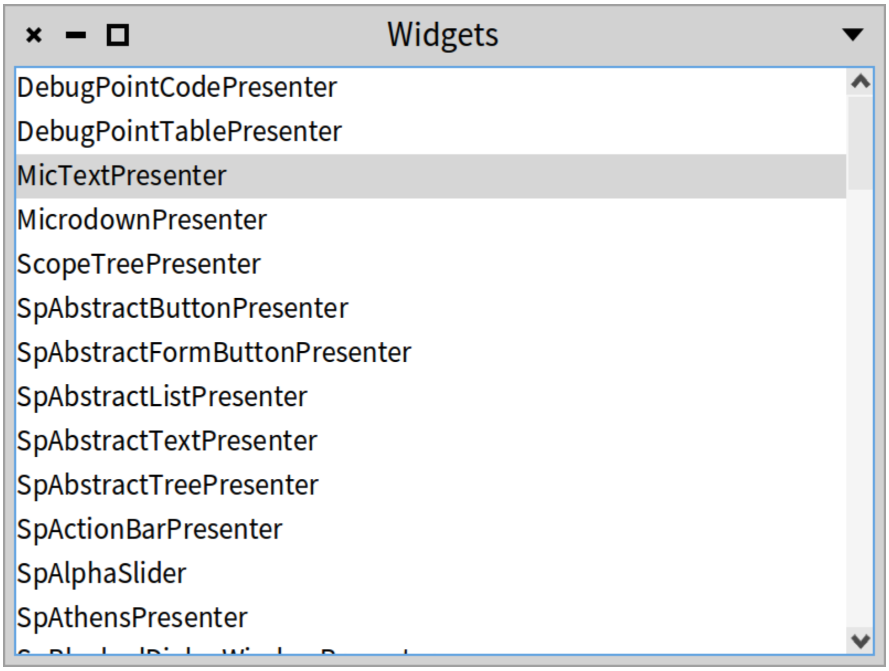
Since this UI will later be used together with other widgets to provide a more complete user interface, some actions will need to occur when a list item is clicked. However, we cannot know beforehand what all these possible actions will be everywhere that it will be reused. Therefore the best solution 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 named `whenSelectionChangedDo:` as follows:

```
WidgetClassListPresenter >> whenSelectionChangedDo: aBlock

  list whenSelectionChangedDo: aBlock
```

Now, whoever reuses this widget can parameterize it with a block that will be executed whenever the selection changes.

7.4 Combining two basic presenters into a reusable UI

The UI we build next 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 from the previous UI. This is because the reuse of a UI as a widget is **not**



DebugPointCodePresenter
DebugPointTablePresenter
MicTextPresenter
MicrodownPresenter
ScopeTreePresenter
SpAbstractButtonPresenter
SpAbstractFormButtonPresenter
SpAbstractListPresenter
SpAbstractTextPresenter
SpAbstractTreePresenter
SpActionBarPresenter
SpAlphaSlider
SpAthensPresenter

Figure 7-2 WidgetClassListPresenter. %anchor=WidgetClassList&width=50

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.

```
SpPresenter << #ProtocolMethodListPresenter
  slots: { #label . #methods };
  package: 'CodeOfSpec20Book'
```

The `initializePresenters` method for this UI is straightforward. We specify the default label text as 'Protocol', which will be changed when the widget is reused.

```
ProtocolMethodListPresenter >> initializePresenters

  methods := self newList.
  methods display: [ :m | m selector ].
  label := self newLabel.
  label label: 'Protocol'.
  self focusOrder add: methods
```

To make sure that we have a nice title when the widget is opened in a window, we define the method `initializeWindow:`.

```
ProtocolMethodListPresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter title: 'Protocol widget'
```

The layout code builds a column with the label above the method list.

```
ProtocolMethodListPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: #label;
      add: #methods;
      yourself
```

This UI can be seen by executing `ProtocolMethodList new open`. As shown in Figure **??** the list is empty and the result is not really nice. This is normal because we did not set any items. We should also place the elements better.

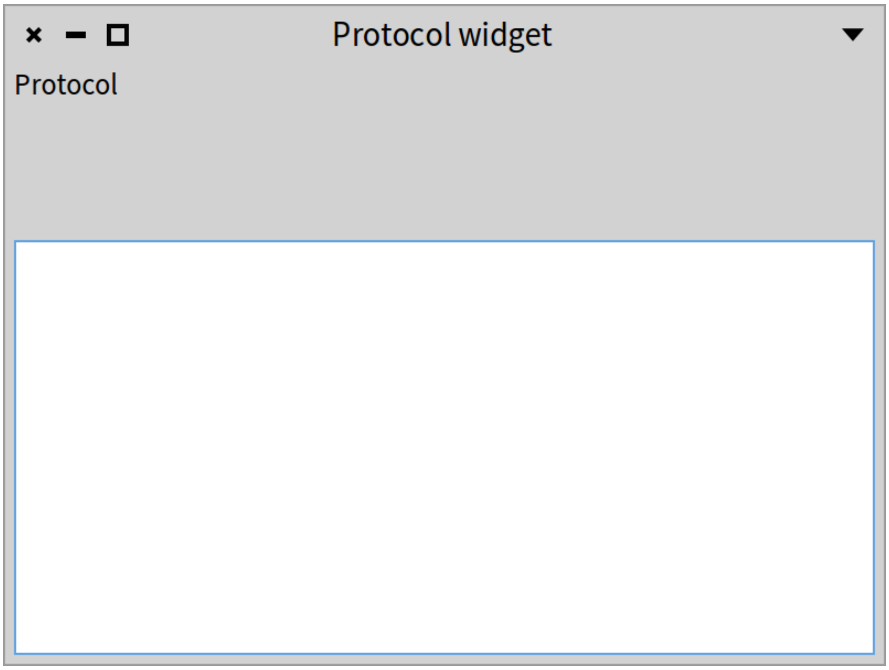


Figure 7-3 ProtocolMethodListPresenter with bad layout. % width=50&anchor=fig-protocollist

```
ProtocolMethodListPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: #label expand: false;
      add: #methods;
      yourself
```

Now you should get a better UI as shown in Figure **??**.

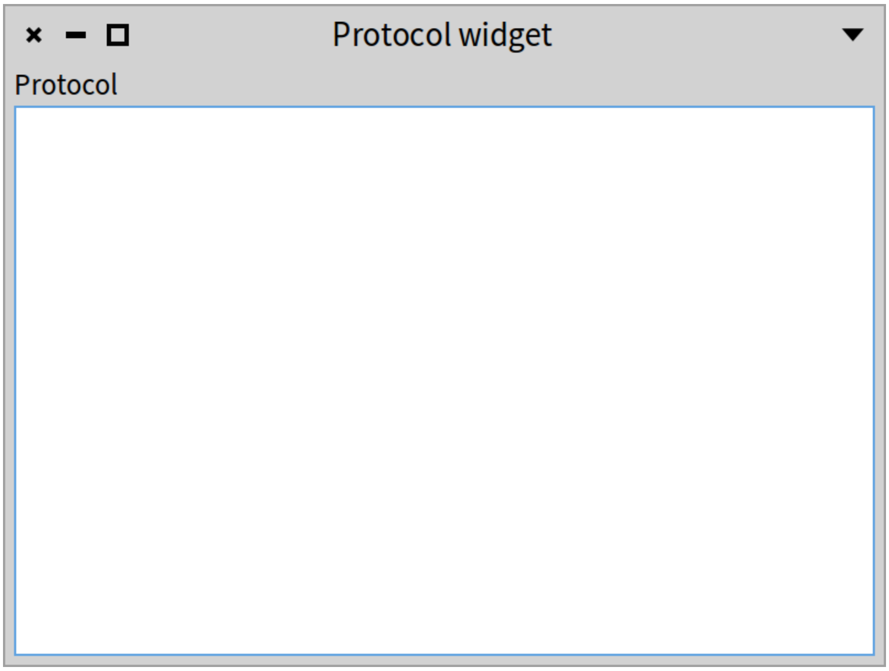


Figure 7-4 ProtocolMethodListPresenter with nicer layout. % width=50&anchor=figprotocollist2

Our protocol method list needs to be configured when it is used, by filling the list of methods and specifying what the name of the protocol is. To allow this, we add some configuration methods:

```
ProtocolMethodListPresenter >> items: aCollection

  methods items: aCollection
```

```
ProtocolMethodListPresenter >> label: aText

  label label: aText
```

```
ProtocolMethodListPresenter >> resetSelection

  methods selection unselectAll
```

```
ProtocolMethodListPresenter >> whenSelectionChangedDo: aBlock

  methods whenSelectionChangedDo: aBlock
```

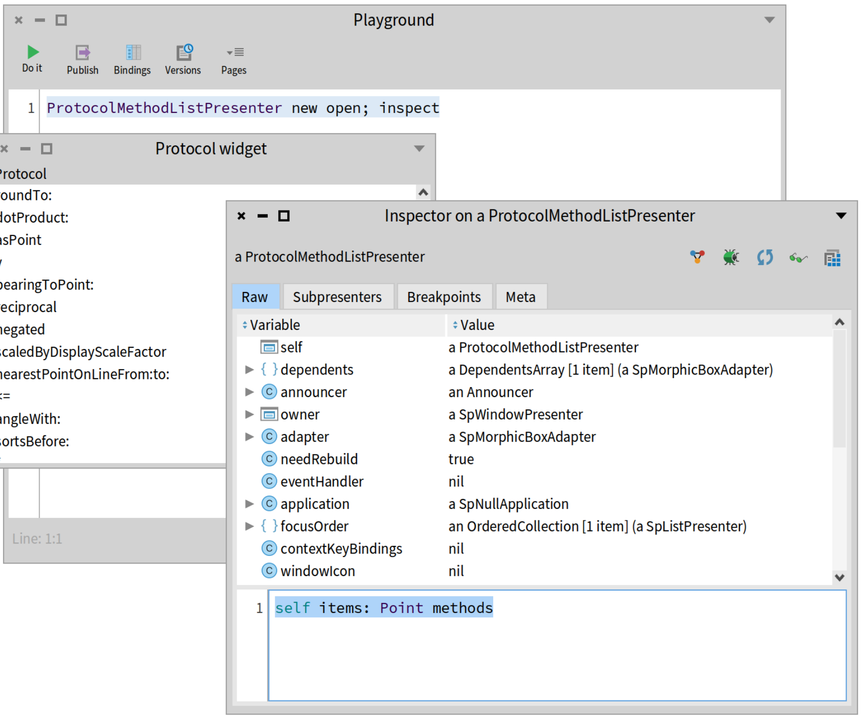
7.5 Live inspection of the widgets

Now we can check manually if the widget is working by doing:

```
ProtocolMethodListPresenter new open; inspect
```

Then in the inspector, we can use the newly created presenter to pass a collection of methods. See the result in Figure **??**.

```
self items: Point methods
```



```
ProtocolMethodListPresenter new open; inspect
```

Protocol
roundTo:
dotProduct:
asPoint
y
bearingToPoint:
reciprocal
negated
scaledByDisplayScaleFactor
nearestPointOnLineFrom:to:
<=
angleWith:
sortsBefore:
*

Variable	Value
self	a ProtocolMethodListPresenter
dependents	a DependentsArray [1 item] (a SpMorphicBoxAdapter)
announcer	an Announcer
owner	a SpWindowPresenter
adapter	a SpMorphicBoxAdapter
needRebuild	true
eventHandler	nil
application	a SpNullApplication
focusOrder	an OrderedCollection [1 item] (a SpListPresenter)
contextKeyBindings	nil
windowIcon	nil

```
self items: Point methods
```

Figure 7-5 Live coding your widgets. % width=90&anchor=figinspectingLive

Now we can play and for example, decide to sort the items as follows:

```
self items: (Point methods sort: #selector ascending)
```

7.6 Writing tests

When we start to feel the need to check manually what we have done, that is a sign that we should write a test instead. It is easy to write simple tests for widgets when we do not use popups. So let's take advantage of that.

We add an accessor to access the method list.

```
ProtocolMethodListPresenter >> methods

  ^ methods
```

```
TestCase << #ProtocolMethodListPresenterTest
  slots: {};
  package: 'CodeOfSpec20Book'
```

```
ProtocolMethodListPresenterTest >> testItems

  | proto methods |
  methods := Point methods sort: #selector ascending.
  proto := ProtocolMethodListPresenter new.
  proto items: methods.
  self assert: proto methods items first class equals: CompiledMethod.
  self assert: proto methods items first selector equals: methods
    first selector
```

We hope that we convinced you that writing simple UI tests is easy with Spec. Do not miss this opportunity to control the complexity of your software.

7.7 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 `WidgetClassListPresenter` and two `ProtocolMethodListPresenters`. It specifies that when a model class is selected in the `WidgetClassListPresenter`, the methods in the protocols 'api' and 'api-events' will be shown in the two `ProtocolMethodListPresenter` widgets.

```
SpPresenter << #ProtocolViewerPresenter
  slots: { #models . #api . #events };
  package: 'CodeOfSpec20Book'
```

The `initializePresenters` method shows the use of `instantiate:` to instantiate widgets, and some of the different parameterization methods of the `ProtocolMethodListPresenter` class.

```
ProtocolViewerPresenter >> initializePresenters

  models := self instantiate: WidgetClassListPresenter.
  api := self instantiate: ProtocolMethodListPresenter.
  events := self instantiate: ProtocolMethodListPresenter.

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

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

```
ProtocolViewerPresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter title: 'Protocol viewer'
```

To describe the interactions between the different widgets we define the `connectPresenters` 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.

```
ProtocolViewerPresenter >> connectPresenters

  models whenSelectionChangedDo: [ :selection |
    | class |
    api resetSelection.
    events resetSelection.
    class := selection selectedItem.
    class
      ifNil: [
        api items: #().
        events items: #() ]
      ifNotNil: [
        api items: (self methodsIn: class for: 'api').
        events items: (self methodsIn: class for: 'api - events') ] ].

  api whenSelectionChangedDo: [ :selection |
    selection selectedItem ifNotNil: [ events resetSelection ] ].
  events whenSelectionChangedDo: [ :selection |
    selection selectedItem ifNotNil: [ api resetSelection ] ]
```

```
ProtocolViewerPresenter >> methodsIn: class for: protocol

  ^ (class methodsInProtocol: protocol)
      sorted: [ :a :b | a selector < b selector ]
```

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

```
ProtocolViewerPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    add: #models;
    add: #api;
    add: #events;
    yourself
```

As previously, the result can be seen by executing the following snippet of code. The result is shown in Figure **??**.

```
ProtocolViewerPresenter new open
```

This user interface is functional. Clicking on a class will show the methods of the 'api' and the 'api-events' protocols of that class.

7.8 Having different layouts

Note that you can change the layout as follows to get all the widgets in a row as shown in Figure **??**. We will show later that a presenter can have multiple layouts and that the programmer decides which one to use.

We can do better. Let us define two methods as follows:

```
ProtocolViewerPresenter >> horizontalLayout

  ^ SpBoxLayout newLeftToRight
    add: #models;
    add: #api;
    add: #events;
    yourself
```

```
ProtocolViewerPresenter >> verticalLayout

  ^ SpBoxLayout newTopToBottom
    add: #models;
    add: #api;
    add: #events;
    yourself
```

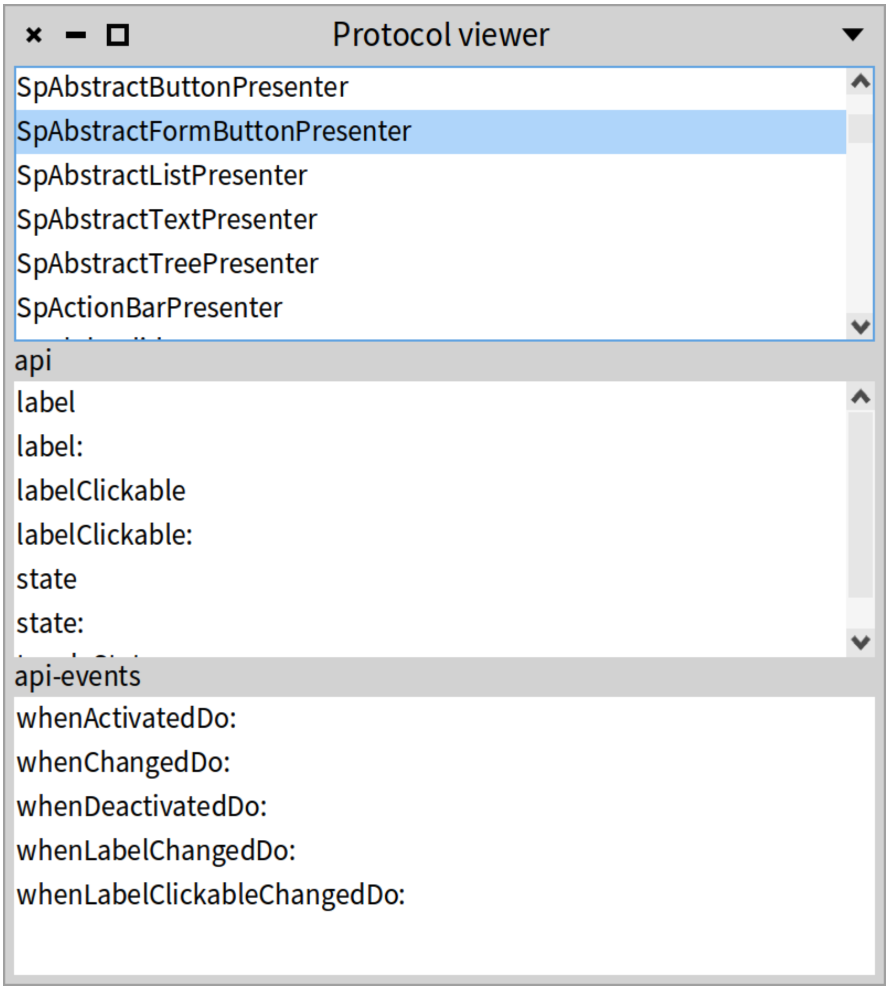


Figure 7-6 ProtocolViewerPresenter in vertical mode. % width=50&anchor=figProtocolViewerVertical

```
ProtocolViewerPresenter >> defaultLayout

  ^ self verticalLayout
```

Now we can decide to open the viewer with different layouts using the message `openWithLayout:` as follows. See Figure **??** for the result.

```
ProtocolViewerPresenter class >> exampleHorizontal

    | inst |
    instance := self new.
    instance openWithLayout: instance horizontalLayout
```

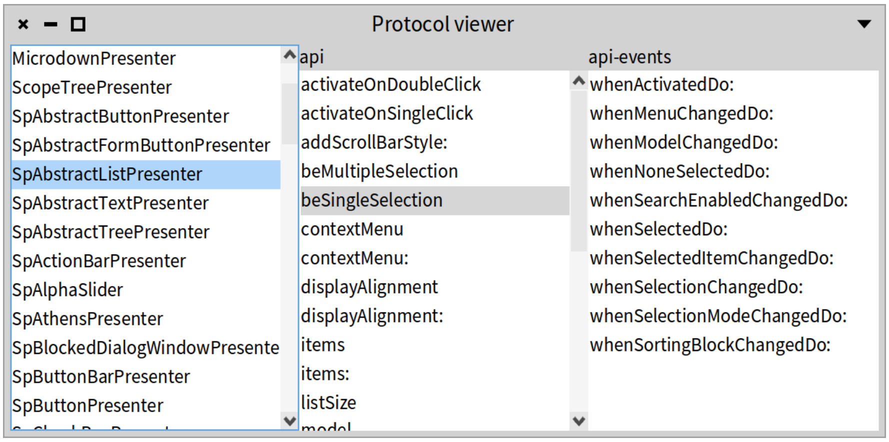


Figure 7-7 ProtocolViewerPresenter in horizontal mode. %width=70&anchor=fig-ProtocolViewerHorizontal

7.9 Enhancing our API

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. Hence we add the following three methods to the 'api' protocol.

```
ProtocolViewerPresenter >> whenSelectionInAPIChanged: aBlock

  api whenSelectionChangedDo: aBlock
```

```
ProtocolViewerPresenter >> whenSelectionInClassChanged: aBlock

  models whenSelectionChangedDo: aBlock
```

```
ProtocolViewerPresenter >> whenSelectionInEventChanged: aBlock

  events whenSelectionChangedDo: aBlock
```

Note. 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 subpresenters accessible.

7.10 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 for your needs. Nonetheless Spec allows you to reuse such a UI by overriding the layout of its widgets, and we show this here.

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

```
SpPresenter << #ProtocolCodeBrowserPresenter
  slots: { #text . #viewer };
  package: 'CodeOfSpec20Book'
```

```
ProtocolCodeBrowserPresenter >> initializePresenters

  text := self instantiate: SpCodePresenter.
  viewer := self instantiate: ProtocolViewerPresenter.
  text syntaxHighlight: true.
  self focusOrder
    add: viewer;
    add: text
```

```
ProtocolCodeBrowserPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: (SpBoxLayout newLeftToRight add: #viewer; yourself);
      add: #text;
      yourself
```

```
ProtocolCodeBrowserPresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter title: 'Spec Protocol Browser'
```

The `connectPresenters` 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.

```
ProtocolCodeBrowserPresenter >> connectPresenters

  viewer whenSelectionInClassChanged: [ :selection |
    text behavior: selection selectedItem ].
  viewer whenSelectionInAPIChanged: [ :selection |
    selection selectedItem
      ifNotNil: [ :item | text beForMethod: item; text: item
    sourceCode ] ].
  viewer whenSelectionInEventChanged: [ :selection |
    selection selectedItem
      ifNotNil: [ :item | text beForMethod: item; text: item
    sourceCode ] ]
```

With the current implementation of `initializePresenters`, opening a window with `ProtocolCodeBrowserPresenter new open` results in a vertical layout for the `ProtocolViewerPresenter` instance held in the `viewer` instance variable because its default layout is the vertical layout. Our objective was to use a different layout. That can be achieved by sending `layout:` to the `viewer`. So let's adapt `initializePresenters` that way.

```
initializePresenters

  text := self instantiate: SpCodePresenter.
  viewer := self instantiate: ProtocolViewerPresenter.
  viewer layout: viewer horizontalLayout.
  text syntaxHighlight: true.
  self focusOrder
    add: viewer;
    add: text
```

Now a window opens as shown in Figure **??**.

7.11 Changing layouts

There are different ways to configure the layout of a presenter. Let's demonstrate that with `ProtocolViewerPresenter`. The first option is using `openWithLayout:` to open a window.

```
presenter := ProtocolViewerPresenter new.
presenter openWithLayout: (SpBoxLayout newLeftToRight
  add: #models;
  add: #api;
  add: #events;
  yourself)
```

Or you can send `layout` to the presenter and open the window afterwards.

```
presenter := ProtocolViewerPresenter new.
presenter layout: (SpBoxLayout newLeftToRight
  add: #models;
  add: #api;
  add: #events;
  yourself).
presenter open
```

An alternative is to use a layout provided by the presenter, like we did in the previous section.

```
presenter := ProtocolViewerPresenter new.
presenter layout: presenter horizontalLayout.
presenter open
```

7.12 Considerations about a public configuration API

In this chapter, we have seen several definitions of methods in the public configuration API of the presenter being built. The implementation of our configuration methods is 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. Fundamentally this is a design decision to be made by the programmer. Not having such methods makes the implementation of the presenter more lightweight but comes at the cost of less clear intent and of breaking encapsulation.

For the former cost, we have seen an example in the protocol method list of Section 7.4. 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. Fundamentally the same 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 presenter will be configured. Without such declared methods, it is less clear to the reuser what can be done to effectively reuse a presenter.

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 presenter and hence severely limit how we can evolve this presenter in the future. It is safer to define a public API and ensure in future versions of the presenter that the functionality of this API remains the same.

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

7.13 New versus old patterns

In Spec 1.0, list presenters exposed a different API, namely `whenSelectedItemChanged:`, as in the following example.

```
initializePresenters

  models := self instantiate: WidgetClassListPresenter.
  api := self instantiate: ProtocolMethodListPresenter.
  events := self instantiate: ProtocolMethodListPresenter.

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


connectPresenters

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

In Spec 2.0, list presenters and friends expose a different object that represents the selection of the list. The design rationale is that a selection is a complex object (single selection, multiple selection). So we have:

```
connectPresenters
  api whenSelectionChangedDo: [ :selection |
    selection selectedItem ifNotNil: [ events resetSelection ] ].
  events whenSelectionChangedDo: [ :selection |
    selection selectedItem ifNotNil: [ api resetSelection ] ]
```

The question for your presenters is what is the API that you should expose to your users. If you like the Spec 1.0 way, that is still possible as shown below.

```
whenSelectedItemChangedDo: aBlock
  methods whenSelectionChangedDo: [ :selection |
    selection selectedItem ifNotNil: [ :item | aBlock value: item ] ]
```

But we advise using the Spec 2.0 way because it will give your presenters consistency with the core presenters of Spec and it will be easier to make them collaborate.

7.14 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.

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 subparts with clear interfaces. We therefore encourage you to think of your UI as a composition of such subparts and construct it modularly, to yield greater productivity.

CHAPTER 8

Lists, tables and trees

An important part of user interfaces is displaying lists of data. Such lists can be structured as tables, plain lists, but also trees supporting the nesting of data.

Spec provides three main presenters: `SpListPresenter`, `SpTreePresenter`, and `SpTablePresenter`. In addition, it offers `SpComponentListPresenter` which allows one to embed any presenter in a list. In this chapter, we present some of the functionality of these presenters.

8.1 Lists

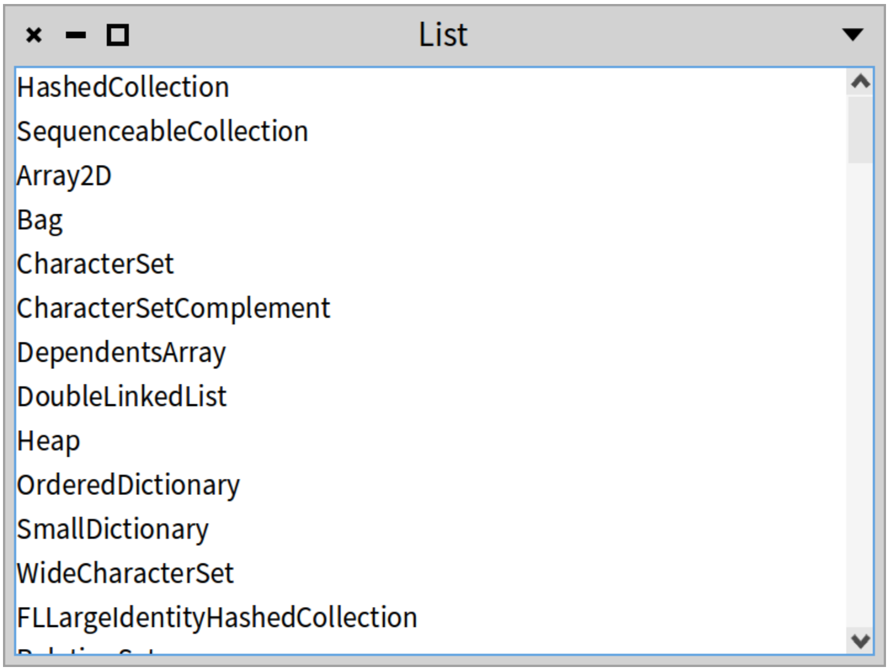
Creating a list is as simple as instantiating a `SpListPresenter` and specifying a list of items that the list should display. The following script illustrates this and the result is shown in Figure **??**.

```
SpListPresenter new
  items: Collection withAllSubclasses;
  open
```

We can change the header title of the list using the message `headerTitle:`. The header title can be hidden using the message `hideHeaderTitle`.

8.2 Controlling item display

By default a list item is displayed using the result of the `asStringOrText` message sent to the item. We can configure a list to apply a block to control the display of each item using the message `display:`. The following script configures



HashedCollection
SequenceableCollection
Array2D
Bag
CharacterSet
CharacterSetComplement
DependentsArray
DoubleLinkedList
Heap
OrderedDictionary
SmallDictionary
WideCharacterSet
FLLargeIdentityHashedCollection

Figure 8-1 A simple list showing class names. % width=50&anchor=figSimpleList

a list presenter to display the name of the methods of the class `Point` instead of showing the result of `asStringOrText`. See Figure 8-2.

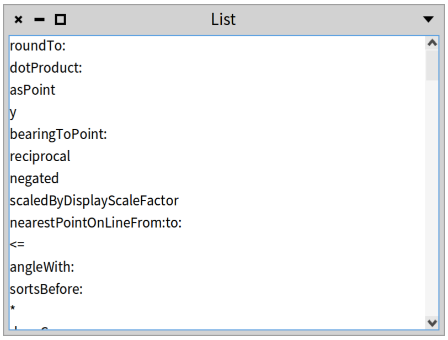
```
SpListPresenter new
  items: Point methods;
  display: [ :item | item selector ];
  open
```

We can sort the items using the message `sortingBlock:`.

```
SpListPresenter new
  items: Point methods;
  display: [ :item | item selector ];
  sortingBlock: [ :a :b | a selector < b selector ];
  open
```

8.3 Decorating elements

We can configure the way items are displayed in a more finer-grained way. The following example illustrates it. We can control the icon associated with the



```
roundTo:
dotProduct:
asPoint
y
bearingToPoint:
reciprocal
negated
scaledByDisplayScaleFactor
nearestPointOnLineFrom:to:
<=
angleWith:
sortsBefore:
*
```

Figure 8-2 A simple list controlling the way items are displayed.

item using the message `displayIcon:`, and the item color using the message `displayColor:`. The format (bold, italic, underline) can the controlled by the corresponding messages `displayItalic:`, `displayBold:` and `displayUnderline:`. See Figure **??**.

```
SpListPresenter new
  items: Collection withAllSubclasses;
  displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
  displayColor: [ :aClass |
    (aClass name endsWith: 'Set')
      ifTrue: [ Color green ]
      ifFalse: [ self theme textColor ] ];
  displayItalic: [ :aClass | aClass isAbstract ];
  displayBold: [ :aClass | aClass hasSubclasses ];
  displayUnderline: [ :aClass | aClass numberOfMethods > 10 ];
  open
```

8.4 About single/multiple selection

Lists support multiple selections. The message `beMultipleSelection` controls that aspect.

```
SpListPresenter new
  items: Collection withAllSubclasses;
  beMultipleSelection;
  open
```

Since selection can hold multiple items, there is an impact on the protocol to react to selection changes. Indeed, lists, filtering lists, trees, and tables offer the `whenSelectionChangedDo:` API and not `whenSelectedItemDo:`. The ar-

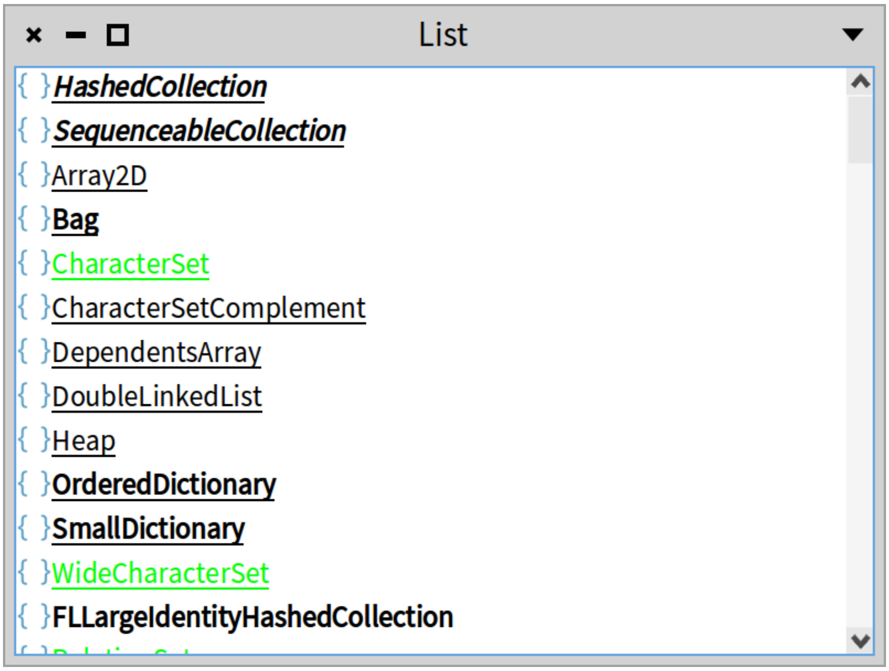


Figure 8-3 A decorated list: icons, text styling, and color.%width=50&anchor=figSimpleListDecorated

gument of the block is an instance of `SpSingleSelectionMode`, `SpMultipleSelectionMode`, `SpTreeMultipleSelectionMode` or `SpTreeSingleSelectionMode`.

Here is a typical use case of the method `whenSelectionChangedDo:`.

```
connectPresenters

  changesTree whenSelectionChangedDo: [ :selection | | diff |
    diff := selection selectedItem
      ifNil: [ '' ]
      ifNotNil: [ :item | self buildDiffFor: item ].
    textArea text: diff ]
```

8.5 Drag and drop

Lists and other container structures support drag and drop. The following script shows how to configure two lists to support dragging from one and dropping in another.

```
| list1 list2 |
list1 := SpListPresenter new.
list1
  items: #( 'abc' 'def' 'xyz' );
  dragEnabled: true.

list2 := SpListPresenter new.
list2 dropEnabled: true;
  wantsDrop: [ :transfer | transfer passenger allSatisfy: #isString ];
  acceptDrop: [ :transfer | list2 items: list2 items , transfer
     passenger ].

SpPresenter new
  layout: (SpBoxLayout newLeftToRight
    add: list1;
    add: list2;
    yourself);
  open
```

The following script illustrates the API.

- `dragEnabled:` configures the receiver to allow dragging of its items.
- `dropEnabled:` configures the receiver to accept dropped items.
- `wantsDrop: [ :transfer | transfer passenger allSatisfy: #isString ]`. With the message `wantsDrop:` we can specify a predicate to accept a dropped elements.
- `acceptDrop: [ :transfer | list2 items: list2 items , transfer passenger ]`. The message `acceptDrop:` specifies the treatment performed once the dropped items are accepted.

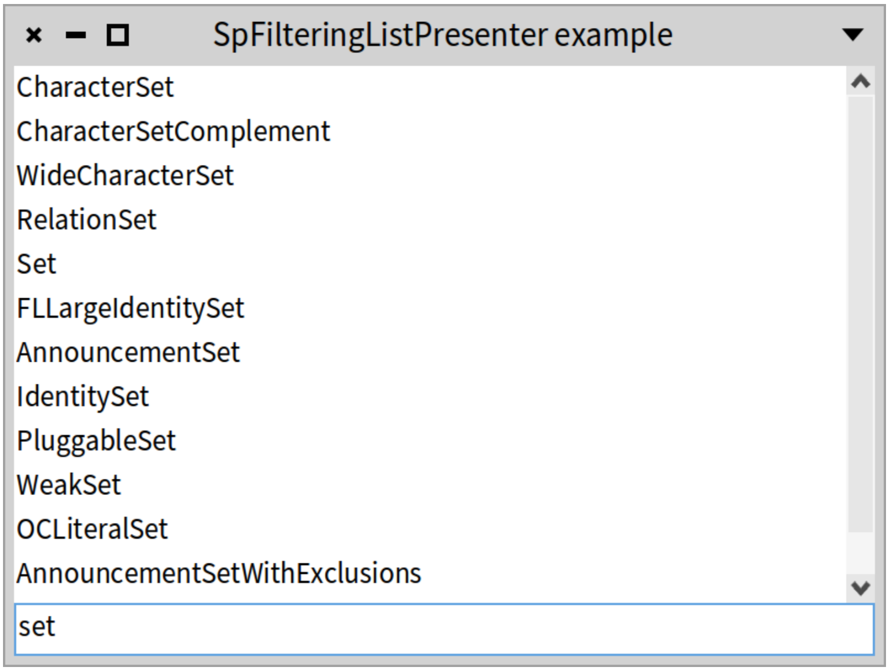
8.6 Activation clicks

An element on a list can be *activated*, meaning it will trigger an event to execute an action on it. Note that an activation is different than a selection: one can *select* an element without activating it. The message `activateOnDoubleClick` configures the list to react to double click, while its counterpart is `activateOnSingleClick`.

8.7 Filtering lists

Lists can also be filtered as shown in Figure **??**. The following script shows the use of the `SpFilteringListPresenter`.

```
SpFilteringListPresenter new
  items: Collection withAllSubclasses;
  open;
  withWindowDo: [ :window |
    window title: 'SpFilteringListPresenter example' ]
```



CharacterSet
CharacterSetComplement
WideCharacterSet
RelationSet
Set
FLLargeIdentitySet
AnnouncementSet
IdentitySet
PluggableSet
WeakSet
OCLiteralSet
AnnouncementSetWithExclusions
set

Figure 8-4 A filtering list with bottom filter. % width=50&anchor=figFiltering

The following script shows that the filter can be placed at the top.

```
SpFilteringListPresenter new
  items: Collection withAllSubclasses;
  openWithLayout: SpFilteringListPresenter topLayout;
  withWindowDo: [ :window |
    window title: 'SpFilteringListPresenter example' ]
```

Note that a filter can be declared upfront using the message `applyFilter:`.

```
SpFilteringListPresenter new
  items: Collection withAllSubclasses;
  openWithLayout: SpFilteringListPresenter topLayout;
  applyFilter: 'set';
  withWindowDo: [ :window |
    window title: 'SpFilteringListPresenter prefiltered example' ]
```

8.8 Selectable filtering lists

Often lists are used to select items. This is what the class `SpFilteringSelectableListPresenter` offers. In addition to being able to filter items, it lets the user select items by ticking them as shown by Figure **??**.

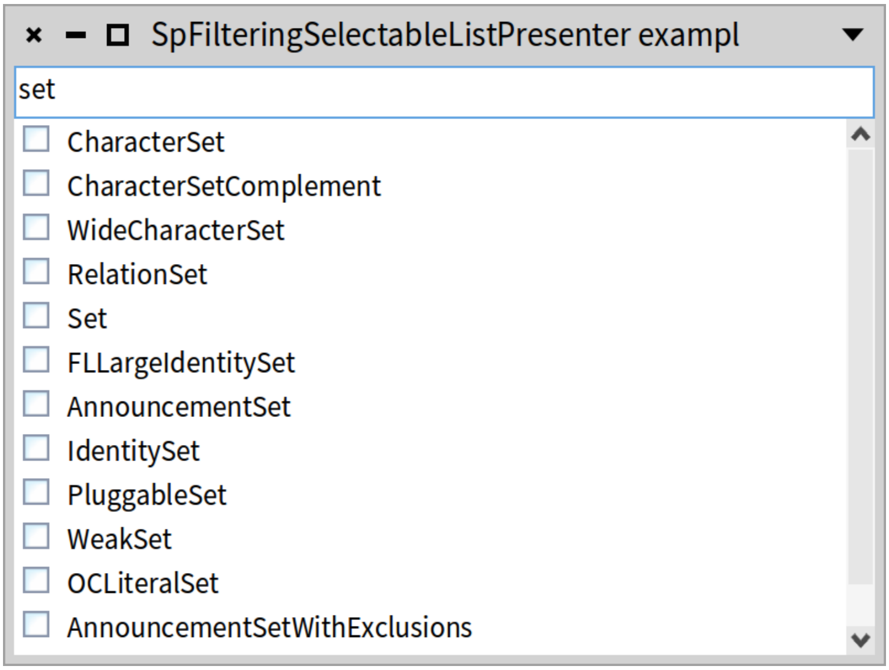


Figure 8-5 A selectable filtering list with a filter at the top. % width=50&anchor=figSelectable

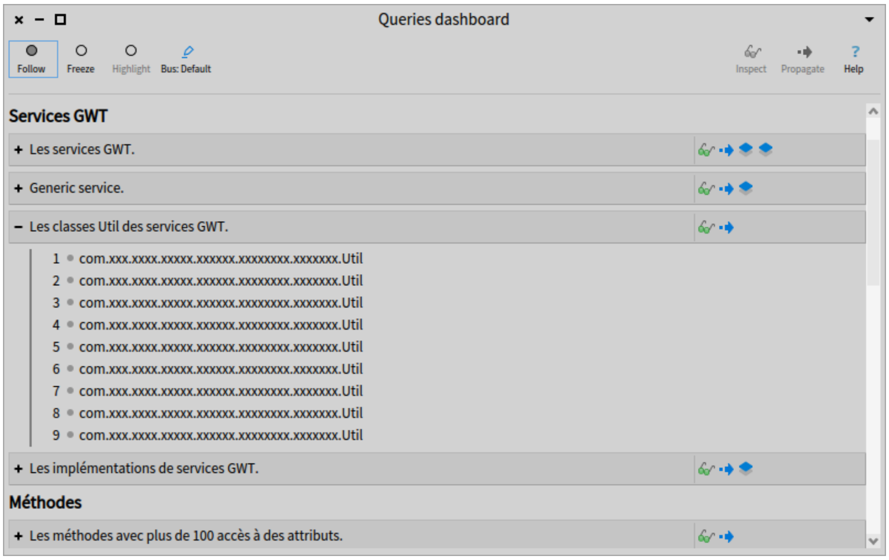
The following script produces this situation.

```
(SpFilteringSelectableListPresenter new
  items: Collection withAllSubclasses;
  layout: SpFilteringListPresenter topLayout;
  applyFilter: 'set';
  asWindow)
    title: 'SpFilteringSelectableListPresenter example';
    open
```

8.9 Component lists

While the lists we saw until now are homogeneous in the sense that they all display strings, Spec offers the possibility to display a list of presenters. It means that elements in the list do not have the same size and can contain other presenters.

This lets developers produce advanced user interfaces such as the one of the report builder of the ModMoose tool suite shown in Figure **??**.



Services GWT

+ Les services GWT.

+ Generic service.

− Les classes Util des services GWT.

1 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
2 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
3 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
4 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
5 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
6 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
7 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
8 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util
9 com.xxx.xxxx.xxxxx.xxxxxx.xxxxxxxx.xxxxxxx.Util

+ Les implémentations de services GWT.

Méthodes

+ Les méthodes avec plus de 100 accès à des attributs.

Figure 8-6 An example of a component list from the ModMoose platform.%width=80&anchor=figModMoose

The following script shows how to define a `SpComponentListPresenter` as shown in Figure **??**.

```
| list |
list := {
  (SpLabelPresenter new
    label: 'Test 1';
    yourself).
  (SpImagePresenter new
    image: (self iconNamed: #smallOk);
    yourself).
  (SpButtonPresenter new
    label: 'A button';
    yourself).
  (SpImagePresenter new
    image: PolymorphSystemSettings pharoLogo asForm;
    yourself) }.

SpComponentListPresenter new
  presenters: list;
  open
```

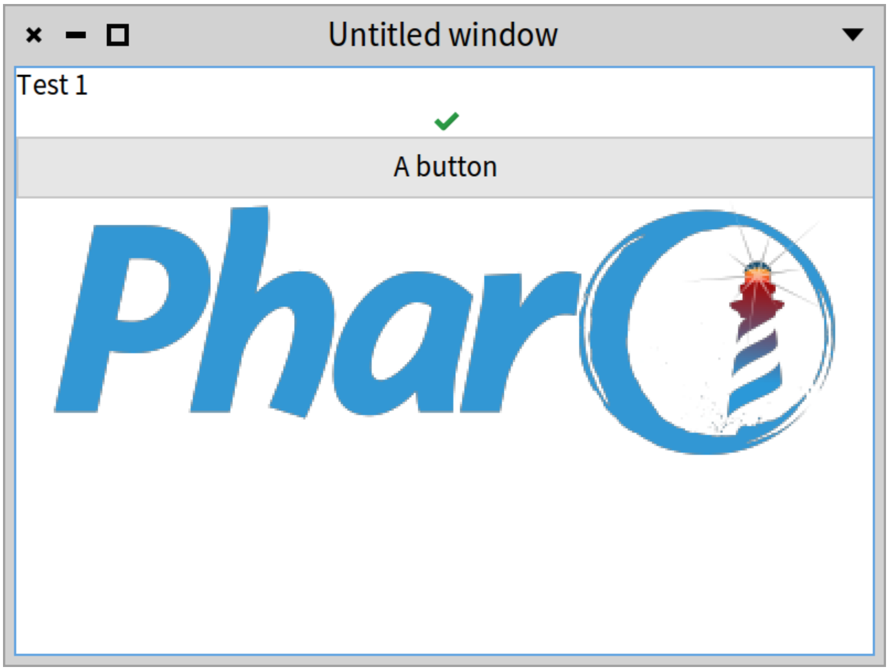


Figure 8-7 A component list with several different presenters: a label, an image, a button, and an image. %width=45&anchor=figCompo

8.10 Trees

Spec offers also trees. The following script shows how to list all the classes of Pharo using inheritance as shown by Figure **??**.

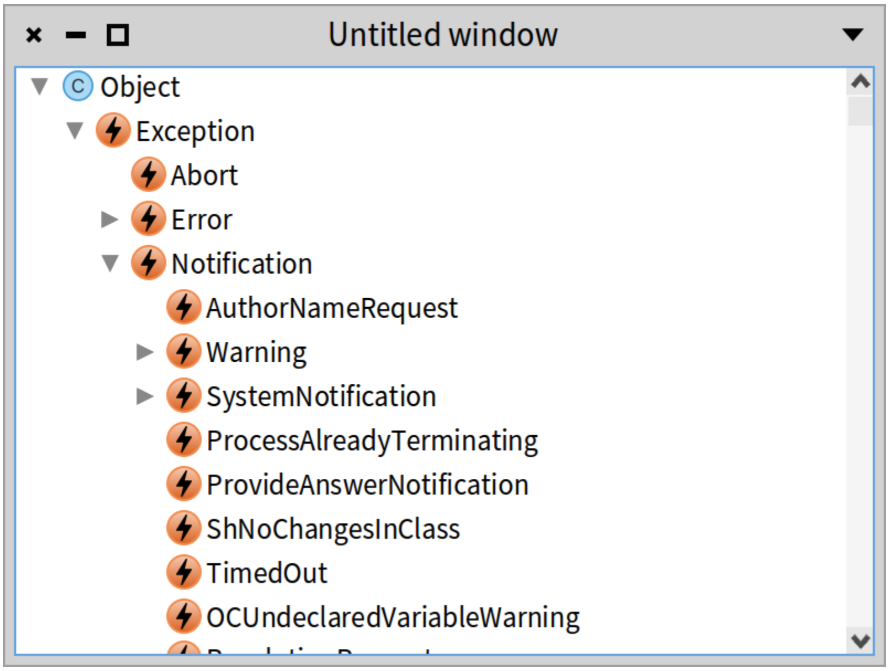


Figure 8-8 A Tree. % width=45&anchor=figTreeExpanded

```
SpTreePresenter new
  roots: { Object };
  children: [ :aClass | aClass subclasses ];
  displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
  display: [ :aClass | aClass name ];
  expandPath: #( 1 1 3 );
  open
```

The message `expandPath:` shows that we can expand a specific item by a path.

The following script shows how to use a dynamic context menu. This is a dynamic menu because its content is recalculated. The dynamic aspect is expressed by a block. Figure **??** shows the result.

```
| tree |
tree := SpTreePresenter new.
tree roots: { Object };
  children: [ :aClass | aClass subclasses ];
  displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
  display: [ :aClass | aClass name ];
  contextMenu: [
```

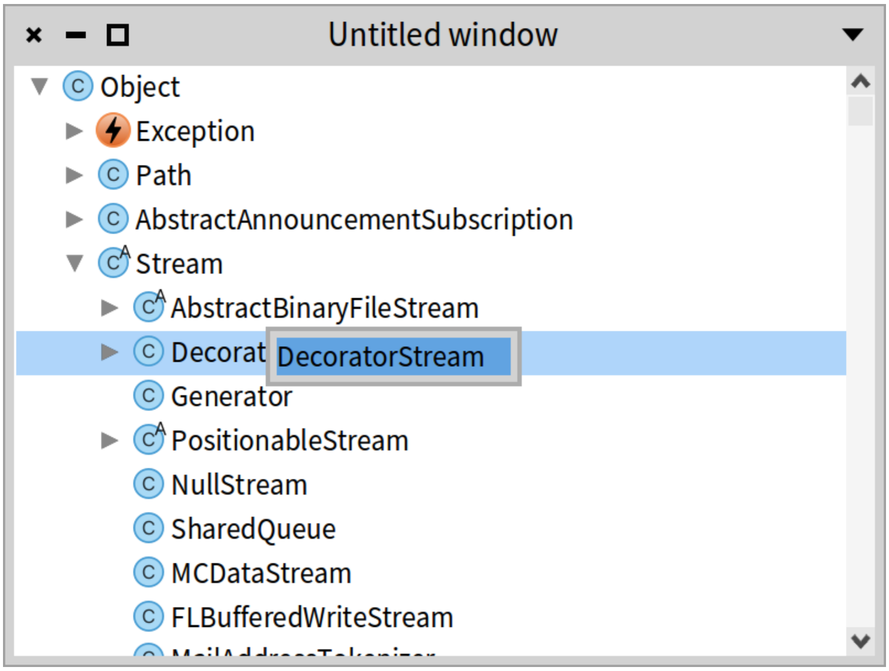


Figure 8-9 A tree with a menu. %width=45&anchor=figTreemenu

```
    SpMenuPresenter new
      addGroup: [ :group |
        group addItem: [ :item | item name: tree selectedItem asString
    ] ] ];
  open
```

The following script shows the use of the message `selectPathByItems:scroll-ToSelection:`, which allows selecting elements by specifying a group of items and asking the tree to scroll to the selection. Figure **??** shows the result.

```
| pathToSpPresenter |
pathToSpPresenter := SpTreePresenter withAllSuperclasses reversed
    allButFirst.
SpTreePresenter new
  roots: { Object };
  children: [ :aClass | aClass subclasses ];
  displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
  display: [ :aClass | aClass name ];
  open;
  selectPathByItems: pathToSpPresenter scrollToSelection: true
```

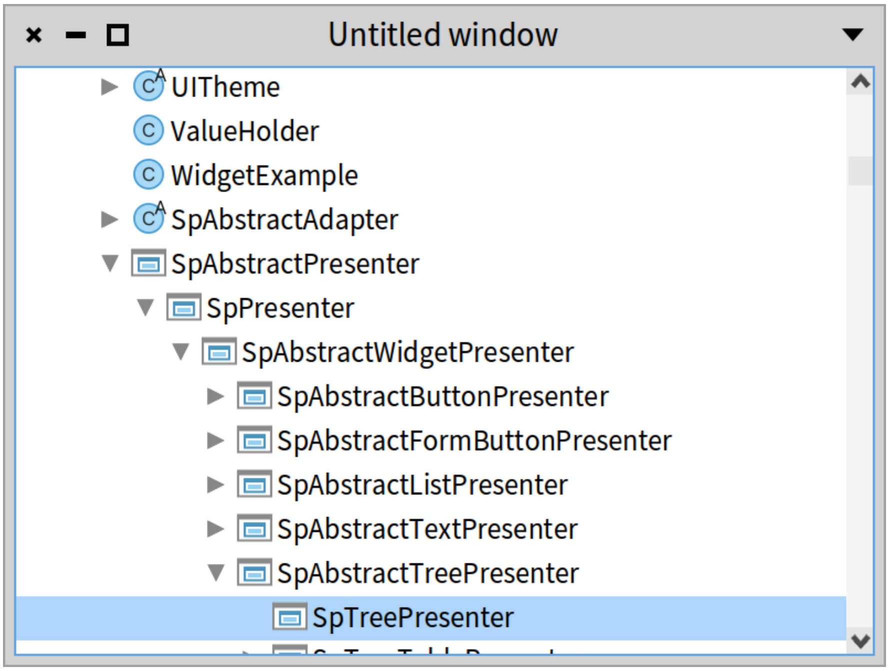


Figure 8-10 A tree with a selected item. % width=50&anchor=figTreeselect

8.11 Tables

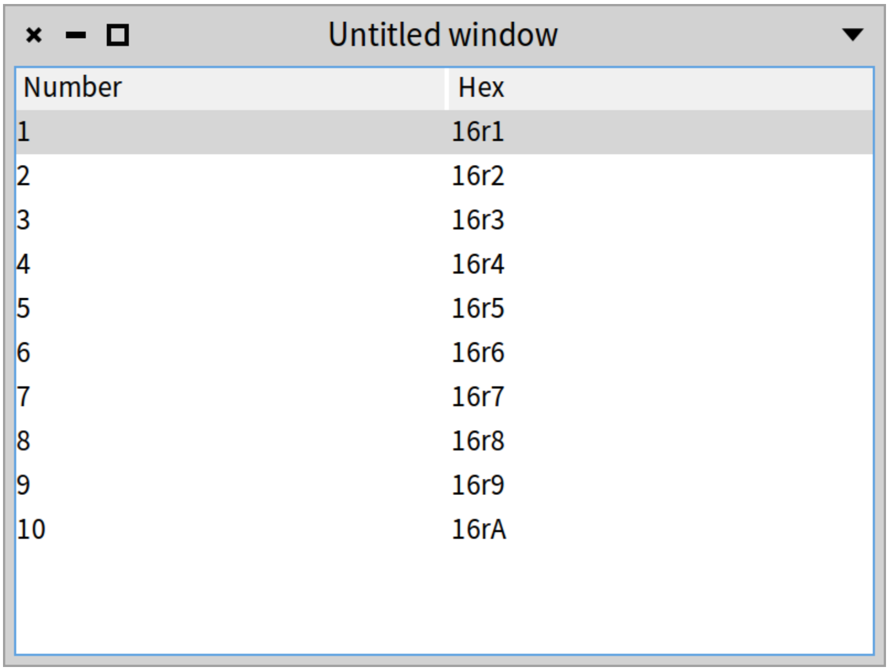
Spec offers tables. A table can have multiple columns and a column can be composed of elementary elements. Tables have different kinds of columns that can be added to a table:

- `SpStringTableColumn` offers cell items that are strings.
- `SpCheckBoxTableColumn` lets us have cells with a checkbox.
- `SpIndexTableColumn` displays the index of the current item.
- `SpDropListTableColumn` lets us have a drop list in cells.
- `SpImageTableColumn` offers cell items with forms (icons, graphics, ...).
- `SpCompositeTableColumn` offers the possibility to compose a column out of different kinds of columns. For instance, it allows one to compose an icon (`SpImageTableColumn`) with a name (`SpStringTableColumn`).

8.12 First table

The following script shows how to define a simple table with two columns as shown in Figure **??**. The message `showColumnHeaders` will display the headers.

```
SpTablePresenter new
  addColumn: (SpStringTableColumn title: 'Number' evaluated:
    #yourself);
  addColumn: (SpStringTableColumn title: 'Hex' evaluated: #hex);
  showColumnHeaders;
  items: (1 to: 10);
  open
```


Untitled window

Number	Hex
1	16r1
2	16r2
3	16r3
4	16r4
5	16r5
6	16r6
7	16r7
8	16r8
9	16r9
10	16rA

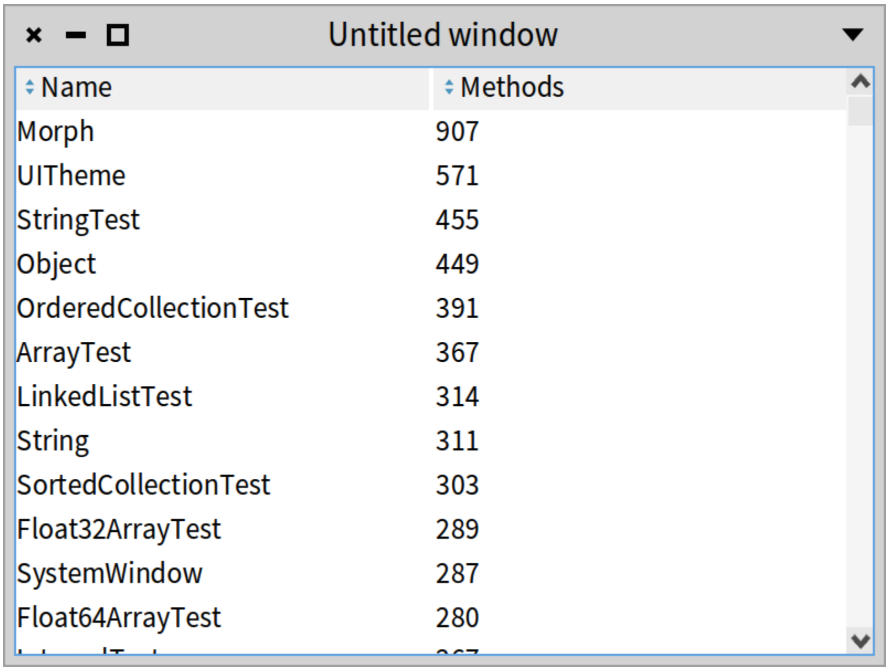
Figure 8-11 A simple table with two columns. % width=50&anchor=figSimpleTable

Add `SpIndexTableColumn title: 'My index'` to the previous table to see the index column in action.

8.13 Sorting headers

The following script presents how to define a table with two sortable columns. Figure **??** shows the result after sorting the second column in descending order.

```
| classNameCompare methodCountSorter |
classNameCompare := [ :c1 :c2 | c1 name < c2 name ].
methodCountSorter := [ :c1 :c2 |
  c1 methodDictionary size threeWayCompareTo: c2 methodDictionary size
    ].

SpTablePresenter new
  addColumn: ((SpStringTableColumn title: 'Name' evaluated: #name)
      compareFunction: classNameCompare);
  addColumn: ((SpStringTableColumn
      title: 'Methods'
      evaluated: [ :c | c methodDictionary size ]) sortFunction:
    methodCountSorter);
  items: Collection withAllSubclasses;
  open
```


Untitled window

Name	Methods
Morph	907
UITheme	571
StringTest	455
Object	449
OrderedCollectionTest	391
ArrayTest	367
LinkedListTest	314
String	311
SortedCollectionTest	303
Float32ArrayTest	289
SystemWindow	287
Float64ArrayTest	280

Figure 8-12 A simple table with two columns that can be sorted. % width=50&anchor=figTableSorting

8.14 Editable tables

The following script shows that table cells can be editable using the messages `beEditable` and `onAcceptEdition:`. The resulting table is shown in Figure **??**.

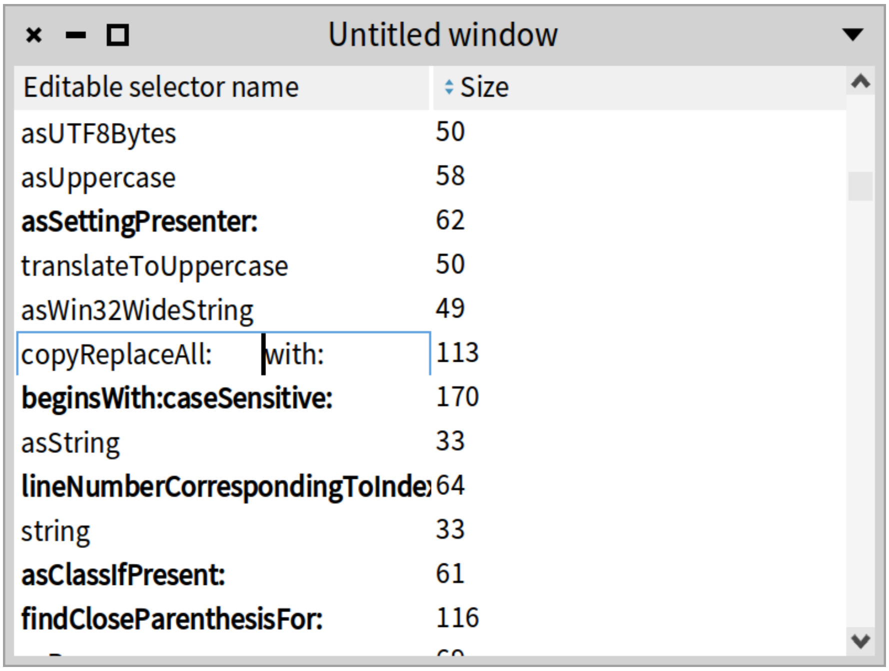
```
| items |
items := String methods.
SpTablePresenter new
  addColumn:
    (SpStringTableColumn new
      title: 'Editable selector name';
      evaluated: [ :m | m selector ];
      displayBold: [ :m | m selector isKeyword ];
      beEditable;
      onAcceptEdition: [ :m :t |
        Transcript
          nextPutAll: t;
          cr;
          endEntry ];
       yourself);
  addColumn:
    (SpStringTableColumn title: 'Size' evaluated: #size)
      beSortable;
      showColumnHeaders;
      items: items;
  open
```

8.15 Tree tables

Spec offers a way to have a tree with extra columns. The class `SpTreeTableP-resenter` encapsulates this behavior. Note that the first column is interpreted as a tree.

The following script shows that the first column will be a tree whose element is composed of an icon and a name: `SpCompositeTableColumn`. Figure **??** shows the window after expanding the root of the tree.

```
SpTreeTablePresenter new
  beResizable;
  addColumn:
    (SpCompositeTableColumn new
      title: 'Classes';
      addColumn:
        (SpImageTableColumn evaluated: [ :aClass |
          self iconNamed: aClass systemIconName ]);
      addColumn:
        (SpStringTableColumn evaluated: [ :each | each name ] );
```



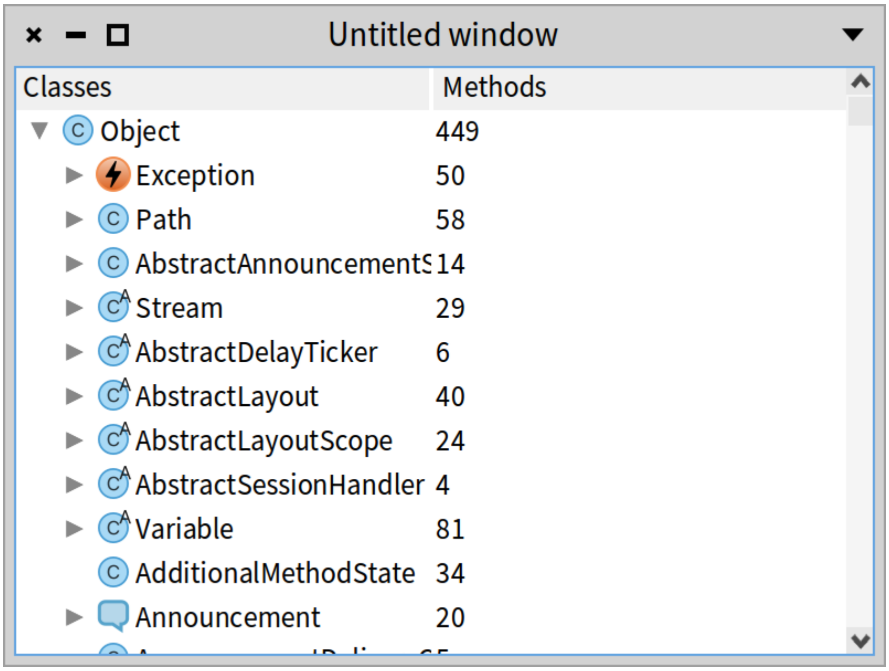
Editable selector name	Size
asUTF8Bytes	50
asUppercase	58
asSettingPresenter:	62
translateToUppercase	50
asWin32WideString	49
copyReplaceAll: with:	113
beginsWith:caseSensitive:	170
asString	33
lineNumberCorrespondingToInde	64
string	33
asClassIfPresent:	61
findCloseParenthesisFor:	116

Figure 8-13 A table with an editable column. % width=50&anchor=figEditableTable

```
      yourself);
  addColumn:
    (SpStringTableColumn new
      title: 'Methods';
      evaluated: [ :class | class methodDictionary size asString ]);
  roots: { Object };
  children: [ :aClass | aClass subclasses ];
  open
```

Sending the messages `width:` and `beExpandable` to the `SpCompositeTableColumn` instance fixes the size of the column.

```
SpCompositeTableColumn new
  title: 'Classes';
  addColumn:
    (SpImageTableColumn evaluated: [ :aClass |
      self iconNamed: aClass systemIconName ]);
  addColumn: (SpStringTableColumn evaluated: #name);
  width: 250;
  beExpandable;
  yourself
```

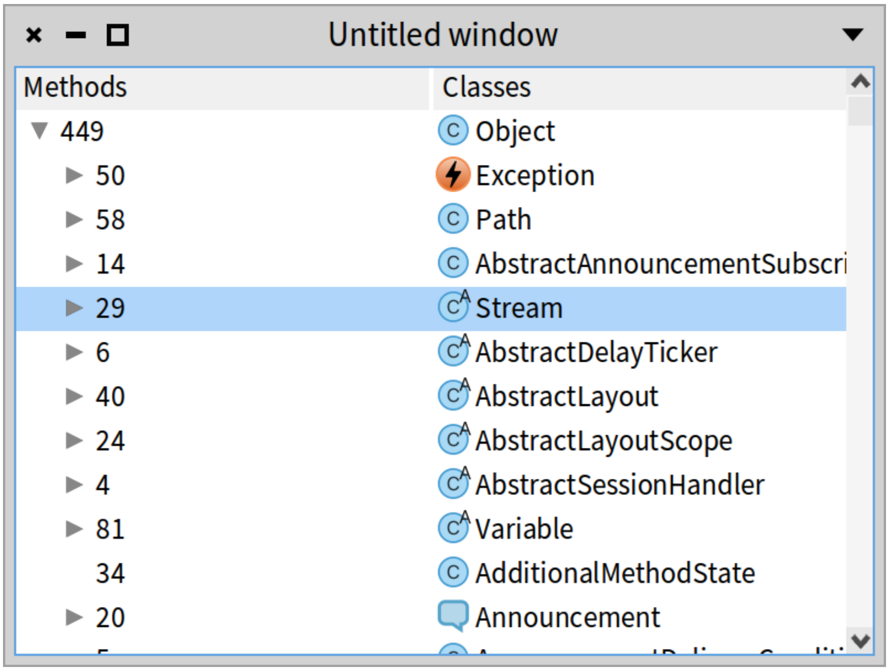


Classes	Methods
▼ Object	449
▶ Exception	50
▶ Path	58
▶ AbstractAnnouncementS	14
▶ Stream	29
▶ AbstractDelayTicker	6
▶ AbstractLayout	40
▶ AbstractLayoutScope	24
▶ AbstractSessionHandler	4
▶ Variable	81
AdditionalMethodState	34
▶ Announcement	20

Figure 8-14 A tree table with two columns: the first one is a composed column with an icon and a string. %width=50&anchor=figTreeTable

You can try the following silly example which results in Figure ??.

```
| compositeColumn |
compositeColumn := SpCompositeTableColumn new title: 'Classes';
  addColumn: (SpImageTableColumn evaluated: [ :aClass |
      self iconNamed: aClass systemIconName ]);
  addColumn: (SpStringTableColumn evaluated: [ :each | each name ] );
  yourself.
SpTreeTablePresenter new
  beResizable;
  addColumn: (SpStringTableColumn new
      title: 'Methods';
      evaluated: [ :class | class methodDictionary size asString ]);
  addColumn: compositeColumn;
  roots: { Object };
  children: [ :aClass | aClass subclasses ];
  open
```



Methods	Classes
▼ 449	Object
▶ 50	Exception
▶ 58	Path
▶ 14	AbstractAnnouncementSubscri
▶ 29	Stream
▶ 6	AbstractDelayTicker
▶ 40	AbstractLayout
▶ 24	AbstractLayoutScope
▶ 4	AbstractSessionHandler
▶ 81	Variable
34	AdditionalMethodState
▶ 20	Announcement

Figure 8-15 A tree table with two columns. % width=50&anchor=figTreeTableSilly

8.16 Conclusion

In this chapter, we presented important containers: lists, component lists, and table presenters.

CHAPTER 9

Managing windows

So far we have described the reuse of SpPresenters, 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 ourselves to opening a window.

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

9.1 A working example

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

```
SpPresenter << #WindowExamplePresenter
  slots: { #minusButton . #plusButton };
  package: 'CodeOfSpec20Book'
```

```
WindowExamplePresenter >> initializePresenters

  plusButton := self newButton.
  minusButton := self newButton.
  plusButton label: '+'.
  minusButton label: '-'
```

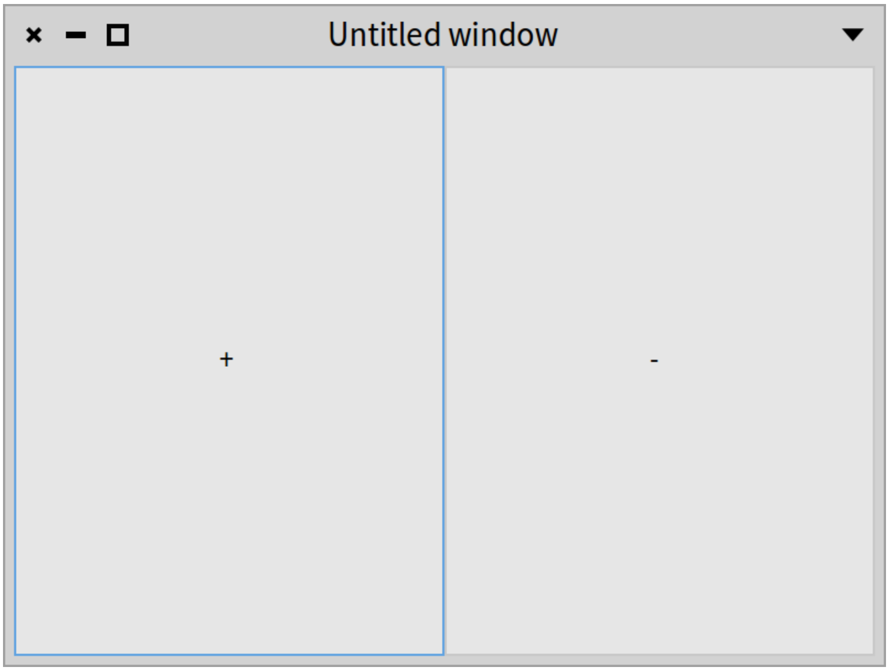


Figure 9-1 A rather simple window on WindowExamplePresenter. %width=50&anchor=windowExample1

```
WindowExamplePresenter >> defaultLayout

  ^ SpBoxLayout newLeftToRight
    add: #plusButton;
    add: #minusButton;
    yourself
```

9.2 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 will show how this is done, including the configuration options specific to dialog boxes. See also Section 9.5 for more information about window decoration.

Opening a window

As we have shown in previous chapters, to open a user interface you have to instantiate the SpPresenter for that interface and send the open message to the instance. That results in the creation of an instance of SpWindowPresenter which points to the window containing the user interface, and showing it in a window on the screen.

We have also seen the openWithLayout: method that takes a layout (instance of SpLayout subclasses) as an argument. Instead of using the default layout, the opened UI will use the layout passed as an argument.

Below we show the two ways we can open a window for our WindowExamplePresenter. The code snippet opens two identical windows as shown in Figure **??**.

```
| presenter |
presenter := WindowExamplePresenter new.
presenter open.
presenter openWithLayout: presenter defaultLayout
```

Opening a dialog box

Spec provides an easy way to open a UI as a simple dialog box with 'Ok' and 'Cancel' buttons. A dialog box does not have icons for resizing and closing, nor a window menu. To open a dialog box, send the message openDialog:

```
| presenter dialog |
presenter := WindowExamplePresenter new.
dialog := presenter openDialog
```

The answer of sending openDialog, assigned to the dialog variable above, is an instance of the SpDialogWindowPresenter class (a subclass of SpWindowPresenter).

The SpDialogWindowPresenter instance can 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.

```
| presenter dialog |
presenter := WindowExamplePresenter new.
dialog := presenter openDialog
  okAction: [ 'okAction' crTrace ];
  cancelAction: [ 'cancelAction' crTrace ]
```

The message canceled sent to dialog will return true if the dialog is closed by clicking on the 'Cancel' button.

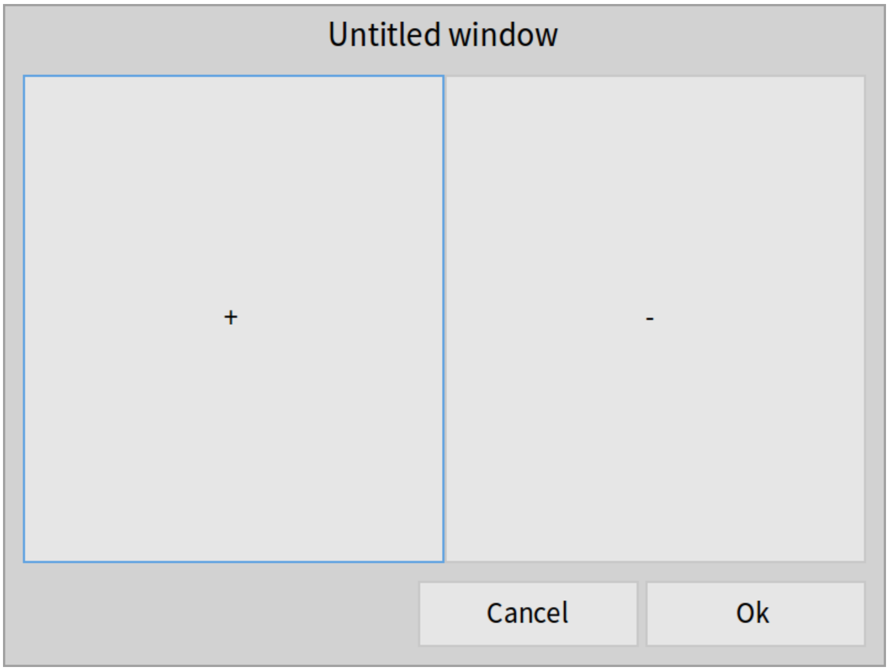


Figure 9-2 A rather simple dialog on WindowExamplePresenter. %width=50&anchor=windowDialog

9.3 Preventing window close

Spec provides a way to check if a window can effectively be closed when the user clicks on the close box. `SpWindowPresenter>>whenWillCloseDo:` takes a block that decides whether the window can be closed. We can change our `WindowExamplePresenter` as follows:

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter whenWillCloseDo: [ :announcement |
    announcement denyClose ]
```

The block has an `announcement` argument. It will be bound to an instance of `SpWindowWillClose`. That class has two interesting methods: `allowClose` and `denyClose`. The code snippet above sends `denyClose` to the announcement. By doing so, we have effectively created an unclosable window!

To be able to close this window, we have to change the implementation of the above method. By default a window can be closed, so the block should only send `denyClose` in case the window cannot be closed. Let's adapt the block to ask whether the user is sure about closing the window.

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter whenWillCloseDo: [ :announcement |
    (self confirm: 'Are you sure that you want to close the window?')
      ifFalse: [ announcement denyClose ] ]
```

Of course, the example method above is extremely simplistic and not very useful. Instead, it should use application-dependent logic of what to check on window close.

9.4 Acting on window close

It is also possible to perform an action whenever a window is closed, both with a plain window or a dialog window.

With a window

When you want to be notified that a window is closed, you should redefine the `initializeWindow:` method in the class of your presenter as follows:

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter whenClosedDo: [ self inform: 'When closed' ]
```

The following snippet programmatically opens and closes a window and you should see the notification triggered on close.

```
| presenter window |
presenter := WindowExamplePresenter new.
window := presenter open.
window close
```

With a dialog window

When you want the same behavior with a dialog window you can either use the mechanism as described previously (i.e. declare your interest in window closing in the method `initializeWindow:`) or configure the dialog presenter returned by the message `openDialog`.

```
| presenter dialog |
presenter := WindowExamplePresenter new.
dialog := presenter openDialog.
dialog
  okAction: [ 'okAction' crTrace ];
  cancelAction: [ 'cancelAction' crTrace ];
  whenClosedDo: [ self inform: 'Bye bye!' ]
```

Action with Window

`withWindowDo:` makes sure that the presenter that scheduled the window still exists or is in a state that makes sense.

```
withWindowDo: [ :window | window title: 'MyTitle' ]
```

9.5 Window size and decoration

Now we focus on sizing a window before and after opening it, and then describe 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, send the `initialExtent:` message to the corresponding `SpWindowPresenter` before opening, for example like this:

```
| windowPresenter |
 windowPresenter := WindowExamplePresenter new asWindow.
 windowPresenter initialExtent: 300@80.
 windowPresenter open
```

The common way to specify the initial size of the window is to use the message `initialExtent:` as follows:

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter initialExtent: 80@100
```

Note that you can also set an initial position using the message `initialPosition:`.

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

```
WindowExamplePresenter >> initializePresenters

  plusButton := self newButton.
  minusButton := self newButton.
  plusButton label: '+'.
  minusButton label: '-'.
  plusButton action: [ self window resize: 500@200].
  minusButton action: [ self window resize: 200@100]
```

You have also `centered`, `centeredRelativeTo:` and `centeredRelativeTo-TopWindow` to help you place the windows relative to world/other windows.

Fixed size

The size of a window can be fixed, so that the user cannot resize it by dragging the sides or corners as follows:

```
| presenter |
presenter := WindowExamplePresenter new open.
presenter 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 `Sp-WindowPresenter` of that window, but the underlying widget library may allow it. Below we show how to get the `SpWindow` of our example and instruct it to remove the different control widgets:

```
| presenter |
presenter := WindowExamplePresenter new open.
presenter window
   removeCollapseBox;
   removeExpandBox;
   removeCloseBox;
   removeMenuBox
```

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

Setting and changing the title

By default, the title of a new window is 'Untitled window'. Of course, this can be changed. The first way is to specialize the method `initializeWindow:` to send the message `title:` to the `windowPresenter` as follows:

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter 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 an argument to the window of the UI. An example is:

```
| presenter |
presenter := WindowExamplePresenter new.
presenter open.
presenter window title: 'I am different!'
```

Setting the about text

The about text of a window can be used by application developers to give a description of the application, and to list its contributors. The about text can be opened by selecting ’About’ from the pop-up menu in the top-right corner of a window, as shown in Figure **??**.

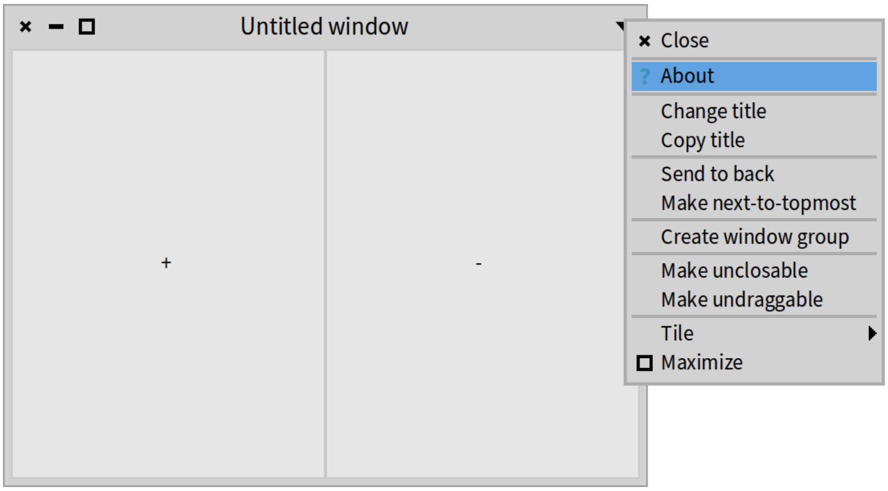
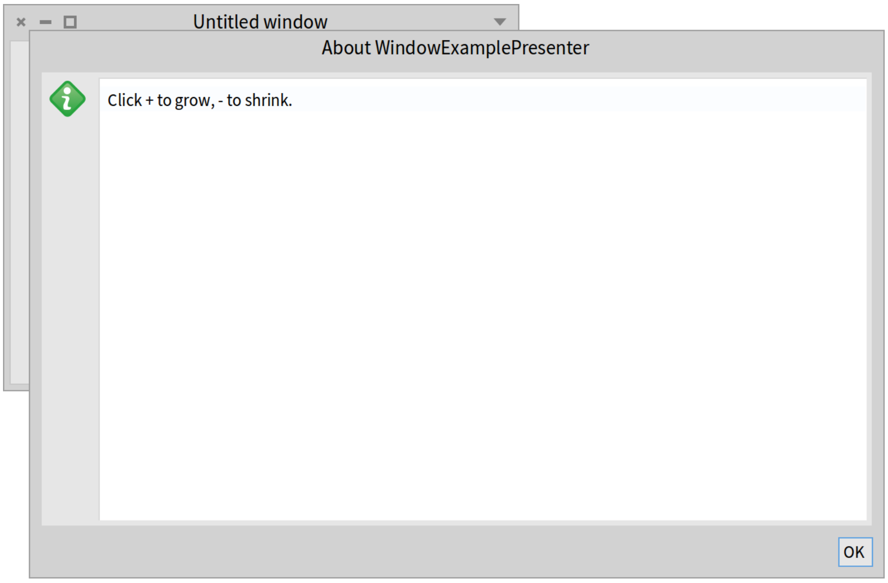


Figure 9-3 Opening the about text of a window.% width=60&anchor=about

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

```
| windowPresenter |
 windowPresenter := WindowExamplePresenter new asWindow.
 windowPresenter aboutText: 'Click + to grow, - to shrink.'.
 windowPresenter open
```

After opening the window with the code snippet above, and after choosing ’About’ from the window menu, the about window opens with the configured about text, as shown in Figure **??**.



Click + to grow, - to shrink.

Figure 9-4 The about text of a window. % width=60&anchor=abouttext

9.6 Getting values from a dialog window

Sending the message `openDialog` to a presenter will return the dialog window itself so you can easily ask it `isOk`. When `isOk` answers `true`, the dialog is in a state to provide the data it has collected from the user.

Let's look at an example. We will open a dialog to select some colors.

Configuring the UI makes up for the largest part of the code below, but the interesting part is at the end. The canceled state is the default state of a dialog so we have to tell the dialog that it is not canceled. We do that in the `okAction` block, where the dialog receives the message `beOk`.

Then in the `whenClosedDo:` block, we send `isOk` to the dialog. If that message answers `true`, it makes sense to process the selection of colors. For the sake of simplicity of this example, we just inspect the selected colors.

```
| selectedColors presenter colorTable dialogPresenter |
selectedColors := Set new.
presenter := SpPresenter new.
colorTable := presenter newTable
  items: (Color red wheel: 10);
  addColumn: (SpCheckBoxTableColumn new
    evaluated: [ :color | selectedColors includes: color ];
    onActivation: [ :color | selectedColors add: color];
    onDeactivation: [ :color | selectedColors remove: color];
    width: 20;
    yourself);
  addColumn: (SpStringTableColumn new
    evaluated: [ :color | '' ];
    displayBackgroundColor: [ :color | color ];
    yourself);
  hideColumnHeaders;
  yourself.
presenter layout: (SpBoxLayout newTopToBottom
  add: colorTable;
  yourself).
dialogPresenter := presenter openDialog.
dialogPresenter
  title: 'Select colors';
  okAction: [ :dialog | dialog beOk ];
  whenClosedDo: [ dialogPresenter isOk
    ifTrue: [ selectedColors inspect ] ]
```

9.7 Little modal dialog presenters

A modal dialog 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.

Spec provides some little predefined dialogs to inform or request information from the users. Most of them inherit from `SpDialogPresenter`. They offer a builder API to configure them.

The simplest dialog is an alert.

```
SpAlertDialog new
  title: 'Inform example';
  label: 'You are seeing an inform dialog!';
  acceptLabel: 'Close this!';
  openModal
```

Confirm dialogs are created as follows:

```
SpConfirmDialog new
  title: 'Confirm example';
  label: 'Are you sure?';
  acceptLabel: 'Sure!';
  cancelLabel: 'No, forget it';
  onAccept: [ :dialog| dialog alert: 'Yes!' ];
  onCancel: [ :dialog| dialog alert: 'No!' ];
  openModal
```

The idiomatic way to use them is to access them via the application of your presenter class:

```
self application newAlert
  title: 'Inform example';
  label: 'You are seeing an inform dialog!';
  acceptLabel: 'Close this!';
  openModal
```

`SpApplication` offers the following API: `newConfirm`, `newAlert`, `newJobList`, `newRequest`, `newSelect`, `newRequestText`.

9.8 Placing a presenter inside a dialog window

Any presenter can be placed in a dialog window by specializing the method `SpAbstractPresenter>>initializeDialogWindow:`, which is implemented like this:

```
initializeDialogWindow: aDialogWindowPresenter
  "Used to initialize the model in the case of the use into a dialog
    window.
   Override this to set buttons other than the default (Ok, Cancel)."

  aDialogWindowPresenter
    addButton: 'Cancel' do: [ :presenter |
      presenter triggerCancelAction.
      presenter close ];
    addDefaultButton: 'Ok' do: [ :presenter |
      presenter triggerOkAction.
      presenter close ]
```

Override this method to define how your presenter will behave when it is open in a dialog window

9.9 Conclusion

In this chapter, we treated the features of Spec that have to do with windows. First we described opening and closing windows as well as how to open a window as a dialog box. That was followed by configuring the window's size and its decorating widgets. After highlighting small yet important details of the window like its title and the about text, the chapter ended with handling dialogs.

CHAPTER 10

Layouts

In Spec, layouts are represented by instances of layout classes. The layout classes encode different positioning of elements such as box, paned, or grid. This chapter presents the available layouts, their definition, and how layouts can be reused when a presenter reuses other presenters.

10.1 Basic principle reminder

Spec expects that layout objects, instances of the layout classes, are associated with a presenter. Each presenter should describe the positioning of its subpresenters.

Contrary to Spec 1.0, where layouts were only defined at the class level, in Spec 2.0, to define the layout of a presenter you can:

- Define the `defaultLayout` method on the instance side
- Use the message `layout:` in your `initializePresenters` method to set an instance of layout in the current presenter.

`defaultLayout` returns a layout and `layout:` sets a layout, for example, an instance of `SpBoxLayout` or `SpPanedLayout`. These two methods are the preferred way to define layouts.

Note that the possibility of defining a class-side accessor e.g. `defaultLayout` remains for those who prefer it.

This new design reflects the dynamic nature of layouts in Spec, and the fact that you can compose them using presenter instances directly, not forcing you to declare subpresenters in instance variables upfront, and then use their names as it was done in Spec 1.0. It is, however, possible that there are cases where you want a layout "template"... so you can still do it.

10.2 A running example

To be able to play with the layouts defined in this chapter, we define a simple presenter named `TwoButtons`.

```
SpPresenter << #TwoButtons
  slots: { #button1 . #button2 };
  package: 'CodeOfSpec20Book'
```

We define a simple `initializePresenters` method as follows:

```
TwoButtons >> initializePresenters

  button1 := self newButton.
  button2 := self newButton.
  button1 label: '1'.
  button2 label: '2'
```

10.3 BoxLayout (SpBoxLayout and SpBoxConstraints)

The class `SpBoxLayout` displays presenters in an ordered sequence of boxes. A box layout can be horizontal or vertical and presenters are ordered left to right and top to bottom respectively. A box layout can be composed of other layouts.

Let us define a first simple layout as follows and whose result is displayed in Figure **??**.

```
TwoButtons >> defaultLayout

  ^ SpBoxLayout newLeftToRight
    add: button1;
    add: button2;
    yourself
```

What we see is that by default a subpresenter expands its size to fit the space of its container.

An element in a vertical box will use all available horizontal space, and fill vertical space according to the rules. This is inversed in a horizontal box.

We can refine this layout to indicate that the subpresenters should not expand to their container using the message `add:expand:`. The result is shown in Figure **??**.

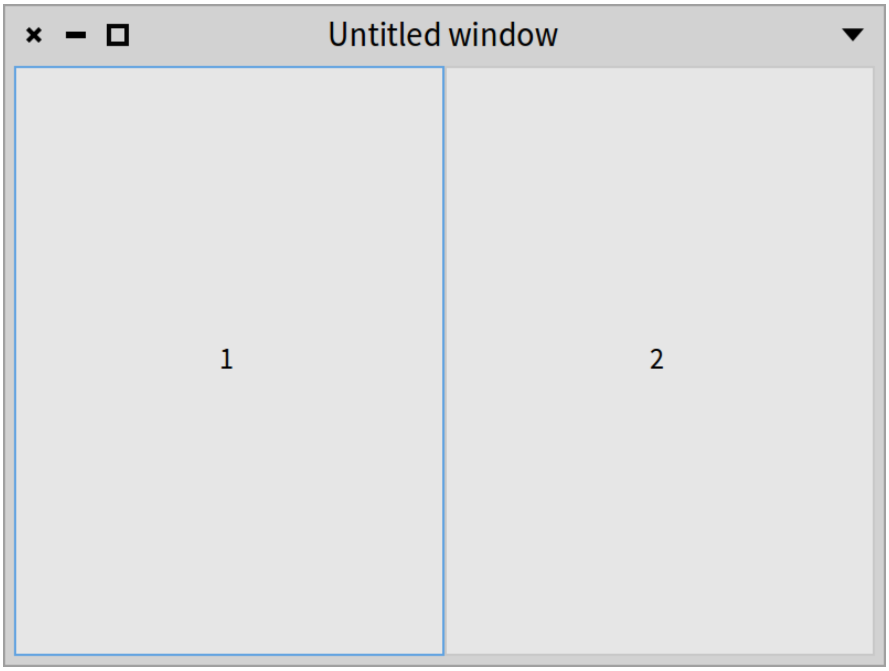


Figure 10-1 Two buttons placed horizontally from left to right. %anchor=TwoButtonsLeftToRight&width=50

```
TwoButtons >> defaultLayout

  ^ SpBoxLayout newLeftToRight
    add: button1 expand: false;
    add: button2 expand: false;
    yourself
```

The full message to add presenters is: `add:expand:fill:padding:`

- `expand:` argument - when true, the new child is to be given extra space allocated to the box. The extra space is divided evenly between all children that use this option.
- `fill:` argument - when true, the space given to a child by the expand option is actually allocated to the child, rather than just padding it. This parameter has no effect if `expand` is set to `false`.
- `padding:` argument - extra space in pixels to put between this child and its neighbors, over and above the global amount specified by the `spacing` property. If a child is a widget at one of the reference ends of the

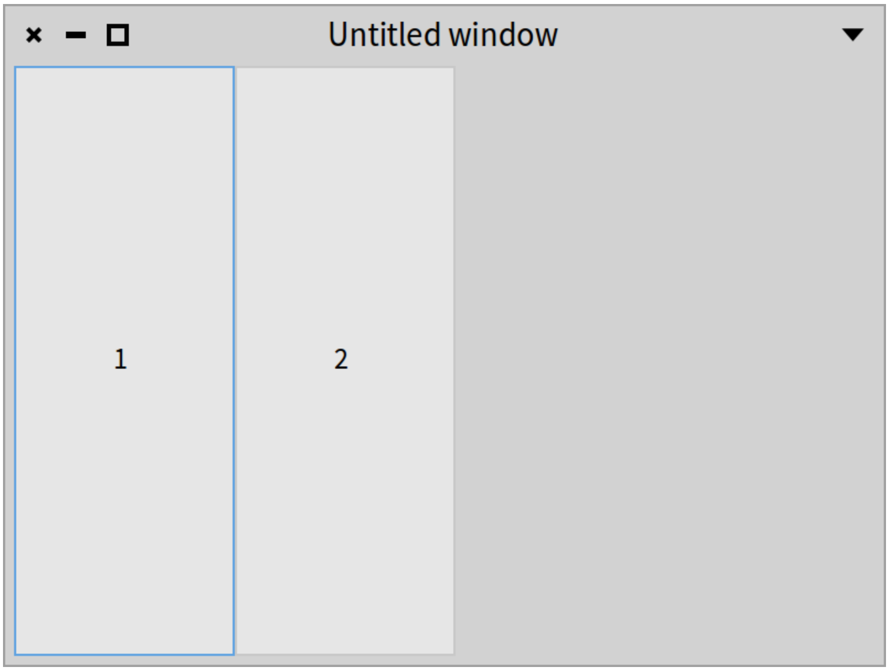


Figure 10-2 Two buttons placed from left to right, but not expanded. %width=50&anchor=TwoButtonsLeftToRightExpanded

box, then padding pixels are also put between the child and the reference edge of the box.

To illustrate this API a bit, we add another button to the presenter and change the `defaultLayout` method as follows. The result is shown in Fig 10-3. We want to stress, however, that it is better not to use a fixed height or padding.

```
TwoButtons >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    spacing: 15;
    add: button1 expand: false fill: true padding: 5;
    add: button2 withConstraints: [ :constraints |
      constraints height: 80; padding: 5 ];
    addLast: button3 expand: false fill: true padding: 5;
    yourself
```

The annotations in the figure indicate the padding in red, the height of `button2` in blue, and the spacing in green. Note that the padding of `button2` is included in the height of the button.

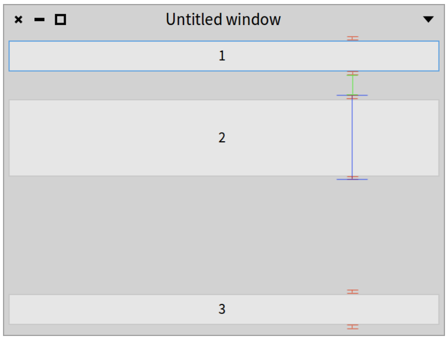


Figure 10-3 Three buttons placed from top to bottom playing with padding and fill options.

The `defaultLayout` method sends the message `withConstraints: [ :constraints | constraints height: 80; padding: 5 ]`. This message allows setting constraints when the often used messages `add:`, `add:expand:`, and `add:expand:fill:padding:` do not cover your particular use case. The `constraints` argument of the block is an instance of the `SpBoxConstraints` class.

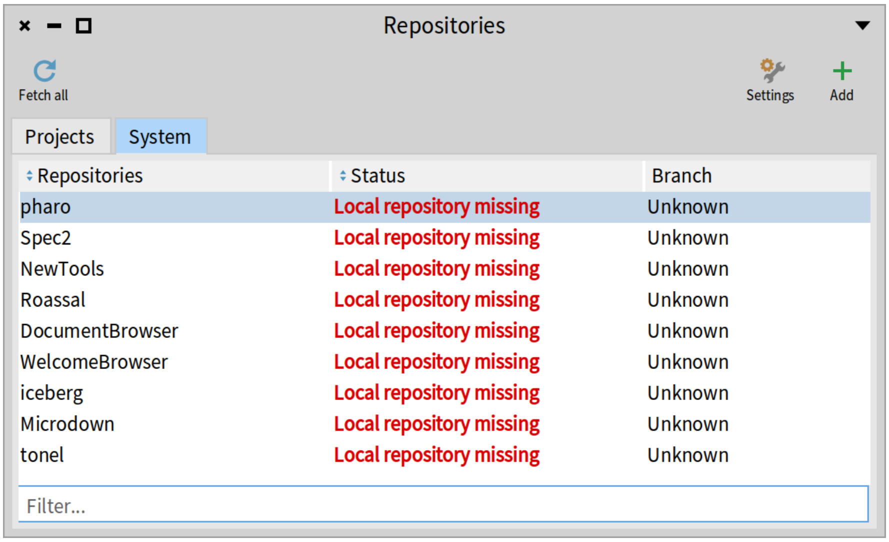
The `defaultLayout` method adds button `button3` to the box layout with `addLast:expand:fill:padding:`. For every method starting with `add:`, the `SpBoxLayout` class provides a similar method starting with `addLast:`.

A box layout has two parts: a ”start” and an ”end”. Messages starting with `add:`, add subpresenters to the ”start”. Messages starting with `addLast:`, add subpresenters to the ”end”. As you can see in Figure 10-3, there is a large gap between `button2` and `button3`. For vertical box layouts, the ”start” part of a box layout aligns to the top side of the box. The ”end” part aligns to the bottom side of the box. The gap between the ”start” and the ”end” is all the excess space not used by the subpresenters.

In this example with three buttons, the usefulness of the ”start” and ”end” parts is not very clear. But it is very handy for button bars with buttons on the left side and on the right side, such as in the Repositories browser of Iceberg, as you can see in Figure **??**. The bar has one button on the left side and two buttons on the right side.

10.4 Box layout alignment

A box layout can be configured with horizontal and vertical alignment of the children. These are the horizontal alignment options, which are messages that



Repositories	Status	Branch
pharo	Local repository missing	Unknown
Spec2	Local repository missing	Unknown
NewTools	Local repository missing	Unknown
Roassal	Local repository missing	Unknown
DocumentBrowser	Local repository missing	Unknown
WelcomeBrowser	Local repository missing	Unknown
iceberg	Local repository missing	Unknown
Microdown	Local repository missing	Unknown
tonel	Local repository missing	Unknown

Figure 10-4 Buttons on the left side and on the right side. % width=60&anchor=Repositories

can be sent to a `SpBoxLayout` instance:

- hAlignStart
- hAlignCenter
- hAlignEnd

These are the vertical layout options:

- vAlignStart
- vAlignCenter
- vAlignEnd

Let’s see how this works in a small example. We will create a presenter with 9 subpresenters, which we will call ”tiles”, layed out in 3 rows with 3 columns. Each subpresenter displays two label presenters with labels ’One’ and ’Two’. The presenter class defines nine instance variables. The names refer to the position of the content inside each tile.

```
SpPresenter << #AlignmentExample
  slots: {
         #northWest .
         #north .
         #northEast .
```

```
      #west .
      #center .
      #east .
      #southWest .
      #south .
      #southEast };
  package: 'CodeOfSpec20Book'
```

As always, `initializePresenters` binds the instance variables that hold the subpresenters. It uses a helper method `newTile:` to create the tiles.

```
AlignmentExample >> initializePresenters

  northWest := self newTile: [ :tileLayout |
    tileLayout vAlignStart; hAlignStart ].
  north := self newTile: [ :tileLayout |
    tileLayout vAlignStart; hAlignCenter ].
  northEast := self newTile: [ :tileLayout |
    tileLayout vAlignStart; hAlignEnd ].
  west := self newTile: [ :tileLayout |
    tileLayout vAlignCenter; hAlignStart ].
  center := self newTile: [ :tileLayout |
    tileLayout vAlignCenter; hAlignCenter ].
  east := self newTile: [ :tileLayout |
    tileLayout vAlignCenter; hAlignEnd ].
  southWest := self newTile: [ :tileLayout |
    tileLayout vAlignEnd; hAlignStart ].
  south := self newTile: [ :tileLayout |
    tileLayout vAlignEnd; hAlignCenter ].
  southEast := self newTile: [ :tileLayout |
    tileLayout vAlignEnd; hAlignEnd ]
```

Note that the block argument of the `newTile:` message has a `titleLayout` argument, which is bound to an instance of `SpBoxLayout`. Inside the nine blocks, the alignment messages that we saw earlier are sent to configure the alignment inside the tiles. For instance, for the top-left tile called ”northWest”, `vAlignStart` is sent to align to the top side of the tile, and `hAlignStart` is sent to align to the left side of the tile.

```
AlignmentExample >> newTile: alignmentBlock

  | tileLayout |
  tileLayout := SpBoxLayout newTopToBottom
    add: self newLabelOne;
    add: self newLabelTwo;
    yourself.
  alignmentBlock value: tileLayout.
  ^ SpPresenter new
```

```
    layout: tileLayout;
    addStyle: 'tile';
    yourself
```

`newTile:` uses two other helper methods:

```
AlignmentExample >> newLabelOne

  ^ self newLabel
    label: 'One';
    yourself
```

```
AlignmentExample >> newLabelTwo

  ^ self newLabel
    label: 'two';
    yourself
```

The layout of the window is defined with:

```
AlignmentExample >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    spacing: 5;
    add: (self rowWithAll: { northWest . north . northEast });
    add: (self rowWithAll: { west . center . east });
    add: (self rowWithAll: { southWest . south . southEast });
    yourself
```

It answers a vertical box layout with three rows. It applies a spacing of 5 pixels between the rows. It sends `rowWithAll:` three times to create horizontal box layouts with three subpresenters each. `rowWithAll:` applies the same spacing of 5 pixels between the tiles in a row.

```
AlignmentExample >> rowWithAll: tiles

  | row |
  row := SpBoxLayout newLeftToRight
    spacing: 5;
    yourself.
  tiles do: [ :tile | row add: tile ].
  ^ row
```

For demonstration purposes, we apply a stylesheet to display tiles with a white background and a black border.

```
AlignmentExample >> application

  ^ SpApplication new
    addStyleSheetFromString: '.application [
      .tile [
        Container { #borderWidth: 2, #borderColor: #black },
        Draw { #backgroundColor: #white } ]
    ]';
    yourself
```

Now we have all the code we need to open the window with:

```
AlignmentExample new open
```

The result is shown in Figure **??**. Each tile displays the label presenters at another location. The label presenters are positioned vertically.

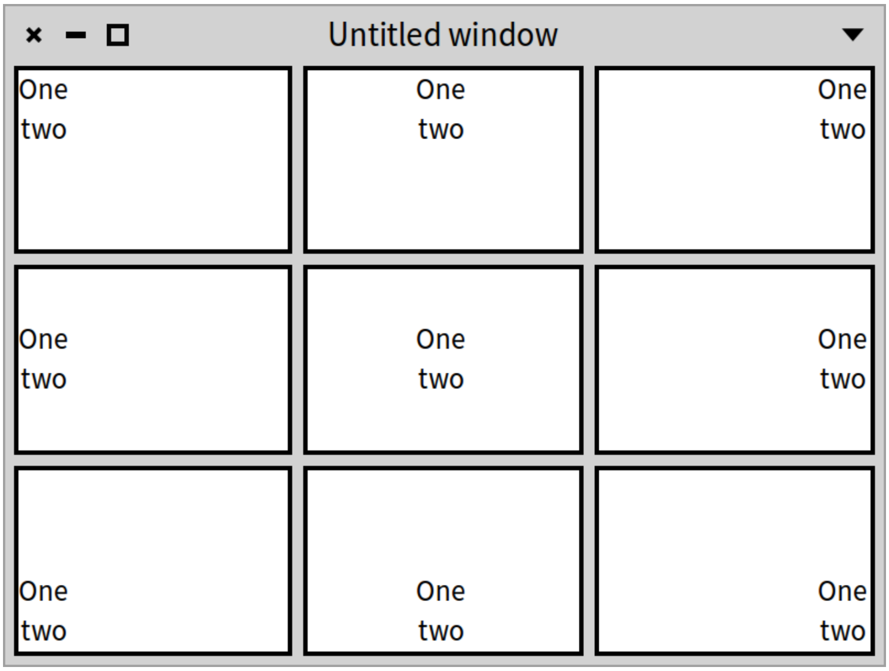


Figure 10-5 Nine tiles with different alignment options. %width=60&anchor=AlignmentExampleWithVerticalTiles

Let's see what happens when we put the label presenters in a horizontal box layout.

```
AlignmentExample >> newTile: alignmentBlock

  | tileLayout |
  tileLayout := SpBoxLayout newLeftToRight
    add: self newLabelOne;
    add: self newLabelTwo;
    yourself.
  alignmentBlock value: tileLayout.
  ^ SpPresenter new
    layout: tileLayout;
    addStyle: 'tile';
    yourself
```

Figure **??** shows the result of opening the window again. Now the labels are positioned horizontally.

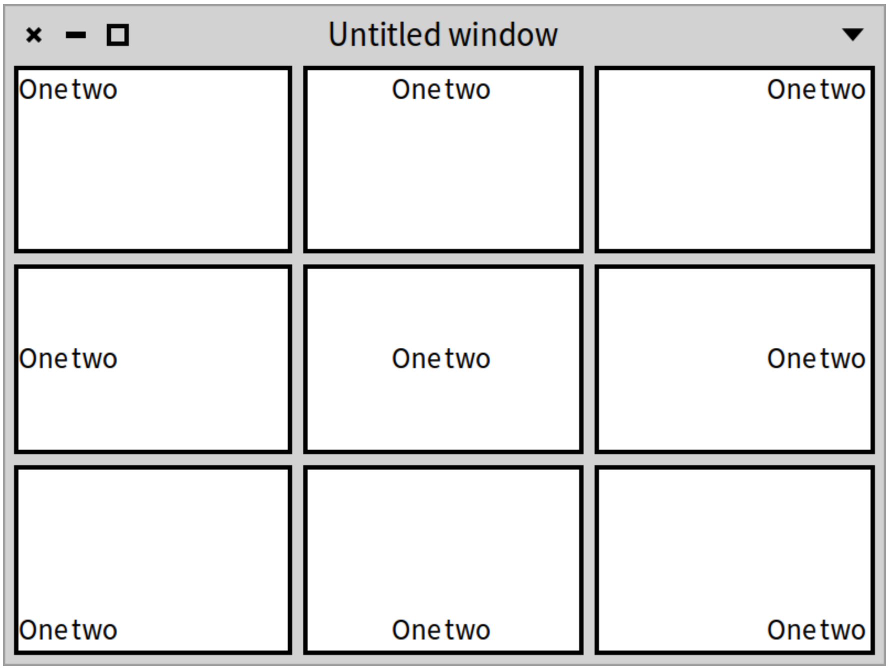


Figure 10-6 Nine tiles with the labels in a vertical box layout.%width=60&anchor=AlignmentExampleWithHorizontalTiles

10.5 Example setup for layout reuse

Before presenting some of the other layouts, we show an important aspect of Spec presenter composition: a composite can declare that it wants to reuse a presenter using a specific layout of a presenter.

Consider our artificial example of a two-button UI. Let us use two layouts as follows. We define two class methods returning different layouts. Note that we could define such methods on the instance side to. We define them on the class side to be able to get the layouts without an instance of the class.

```
TwoButtons class >> buttonRow

  ^ SpBoxLayout newLeftToRight
    add: #button1;
    add: #button2;
    yourself
```

```
TwoButtons class >> buttonColumn

  ^ SpBoxLayout newTopToBottom
    add: #button1;
    add: #button2;
    yourself
```

Note that when we define the layout at the class level, we use a symbol whose name is the corresponding instance variable. Hence we use `#button2` to refer to the presenter stored in the instance variable `button2`.

10.6 Opening with a layout

The message `openWithLayout:` lets you specify the layout you want to use when opening a presenter. Here are some examples:

- `TwoButtons new openWithLayout: TwoButtons buttonRow` places the buttons in a row.
- `TwoButtons new openWithLayout: TwoButtons buttonColumn` places them in a column.

We define a `defaultLayout` method which invokes one of the previously defined methods so that the presenter can be opened without giving a layout.

```
TwoButtons >> defaultLayout

  ^ self class buttonRow
```

10.7 Better design

We can do better and define two instance level methods to encapsulate the layout configuration.

```
TwoButtons >> beColumn

  self layout: self class buttonColumn
```

```
TwoButtons >> beRow

  self layout: self class buttonRow
```

Then we can write the following script:

```
TwoButtons new
  beColumn;
  open
```

10.8 Specifying a layout when reusing a presenter

Having multiple layouts for a presenter implies that there is a way to specify the layout to use when a presenter is reused. This is simple. We use the method `layout:`. Here is an example. We create a new presenter named `ButtonAndListH`.

```
SpPresenter << #ButtonAndListH
  slots: { #buttons . #list };
  package: 'CodeOfSpec20Book'
```

```
ButtonAndListH >> initializePresenters

  buttons := self instantiate: TwoButtons.
  list := self newList.
  list items: (1 to: 10)
```

```
ButtonAndListH >> initializeWindow: aWindowPresenter

  aWindowPresenter title: 'SuperWidget'
```

```
ButtonAndListH >> defaultLayout

  ^ SpBoxLayout newLeftToRight
    add: buttons;
    add: list;
    yourself
```

This `ButtonAndListH` class results in a SuperWidget window as shown in Figure **??**. It reuses the `TwoButtons` widget and places all three widgets in a horizontal order because the `TwoButtons` widget uses the `buttonRow` layout method by default.

Alternatively, we can create `ButtonAndListV` class as a subclass of `ButtonAndListH` and only change the `initializePresenters` method as below. It specifies that the reused `buttons` widget should use the `buttonColumn` layout method, and hence results in the window shown in Figure **??**.

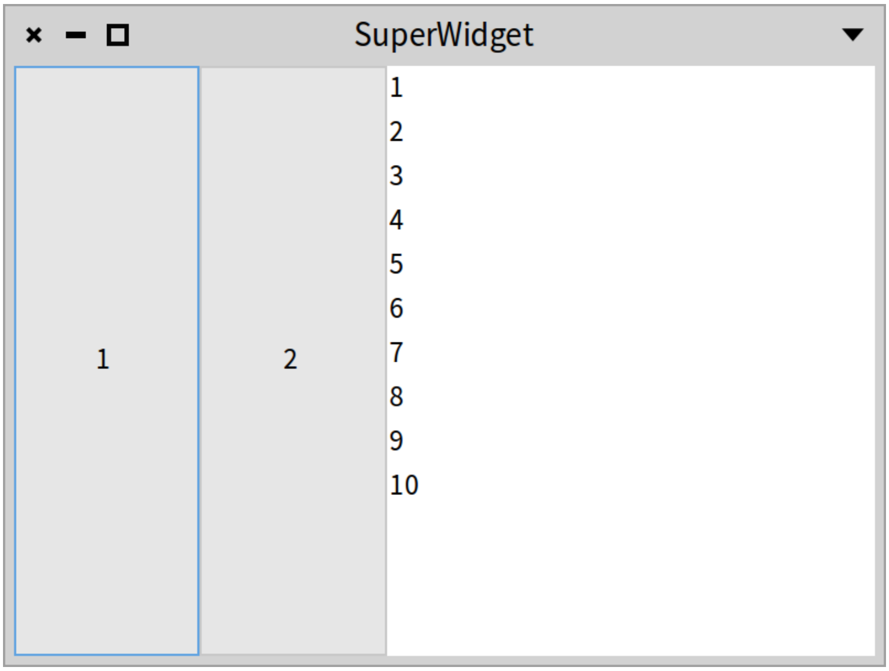


Figure 10-7 Buttons placed horizontally.% width=50&anchor=ButtonAndListH

```
ButtonAndListH << #ButtonAndListV
  slots: {};
  package: 'CodeOfSpec20Book'
```

```
ButtonAndListV >> initializePresenters

  super initializePresenters.
  buttons beColumn
```

Alternative to declare subcomponent layout choice

The alternative is to define a new method `defaultLayout` and to use the `add:layout:` message. We define a different presenter.

```
ButtonAndListH << #ButtonAndListV2
  slots: {};
  package: 'CodeOfSpec20Book'
```

We define a new `defaultLayout` method as follows:

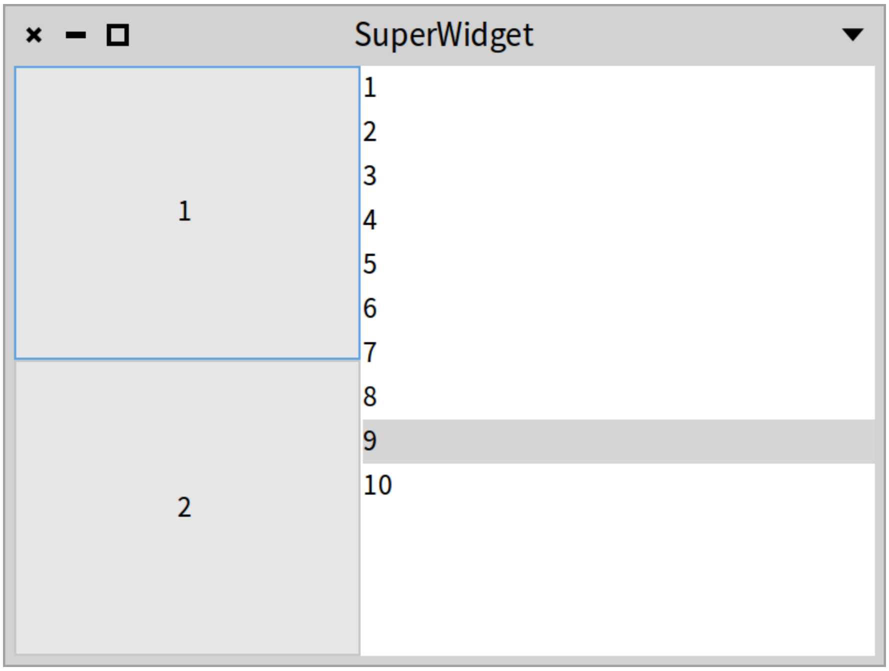


Figure 10-8 Buttons placed vertically. % width=50&anchor=ButtonAndListV

```
ButtonAndListV2 >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    add: buttons layout: #buttonColumn;
    add: list;
    yourself
```

Note the use of the message `add:layout:` with the selector of the method returning the layout configuration: `#buttonColumn`. This is normal since we cannot access the state of a subcomponent at this moment. Let's open a window with:

```
ButtonAndListV2 new open
```

That opens the window shown in Figure **??**.

Dynamically changing a layout

It is possible to change the layout of a presenter dynamically, for example from an inspector. Open the presenter with:

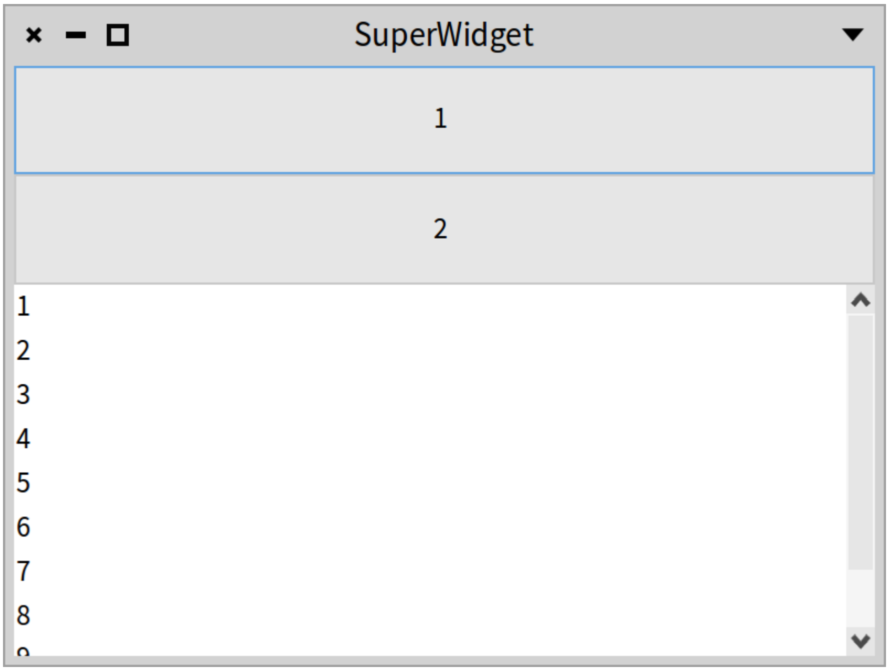


Figure 10-9 Buttons and list placed vertically. % width=50&anchor=ButtonAndListV2

```
ButtonAndListV new inspect open
```

That opens an inspector on the presenter, and a window with the buttons placed vertically as shown in Figure **??**.

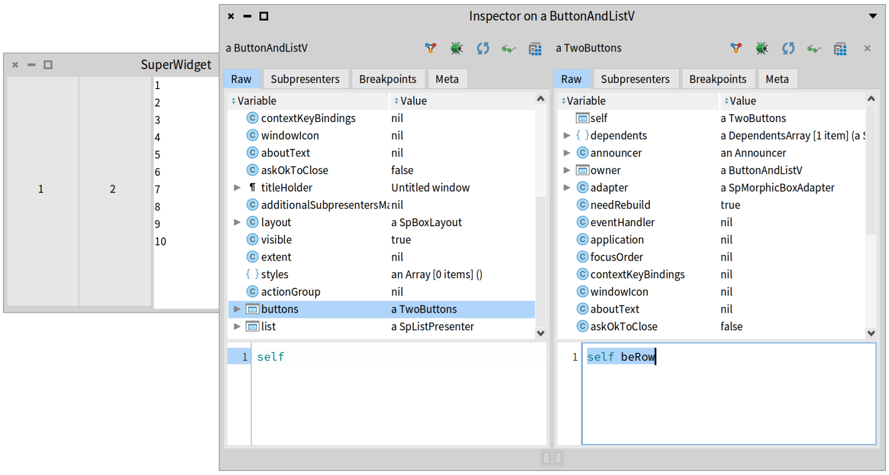
Then select the 'buttons' instance variable in the inspector and do `self beRow`. The result is shown Figure **??**.

10.9 Grid layout (SpGridLayout)

The class `SpGridLayout` arranges subpresenters in a grid according to certain layout properties such as:

- A position is mandatory (`columnNumber@rowNumber`)
- A span can be added if desired (`columnExtension@rowExtension`)

The following example opens a window with a grid layout with several widgets, as shown in Figure **??**.



Variable	Value
contextKeyBindings	nil
windowIcon	nil
aboutText	nil
askOkToClose	false
titleHolder	Untitled window
additionalSubpresentersM:	nil
layout	a SpBoxLayout
visible	true
extent	nil
styles	an Array [0 items] ()
actionGroup	nil
buttons	a TwoButtons
list	a SpListPresenter

```
self
```

Variable	Value
self	a TwoButtons
dependents	a DependentsArray [1 item] (a S
announcer	an Announcer
owner	a ButtonAndListV
adapter	a SpMorphicBoxAdapter
needRebuild	true
eventHandler	nil
application	nil
focusOrder	nil
contextKeyBindings	nil
windowIcon	nil
aboutText	nil
askOkToClose	false

```
self beRow
```

Figure 10-10 Tweaking and playing interactively with layouts from the inspector.%width=100&anchor=InteractiveTweaking

```
SpPresenter << #GridExample
  slots: { #promptLabel . #nameText . #suggestionsText . #submitButton
    };
  package: 'CodeOfSpec20Book'
```

```
GridExample >> initializePresenters

  promptLabel := self newLabel
    label: 'Please enter your name and your suggestions.';
    yourself.
  nameText := self newTextInput.
  suggestionsText := self newText.
  submitButton := self newButton
    label: 'Submit';
    yourself
```

```
GridExample >> defaultLayout

  ^ SpGridLayout new
    add: #promptLabel at: 1@1 span: 3@1;
    add: 'Name:' at: 1@2;
    add: #nameText at: 2@2 span: 2@1;
    add: 'Suggestions:' at: 1@3;
    add: #suggestionsText at: 2@3 span: 2@1;
    add: #submitButton at: 2@4 span: 1@1;
    yourself
```

The layout defines a grid with three columns. The prompt 'Please enter your name and your suggestions.' spans the three columns. The labels of the two fields are put in the first column. The fields span the second and the third column. The button is put in the second column. The second field is a multi-line text field. That is why it is higher than the first field, which is a single-line text field.

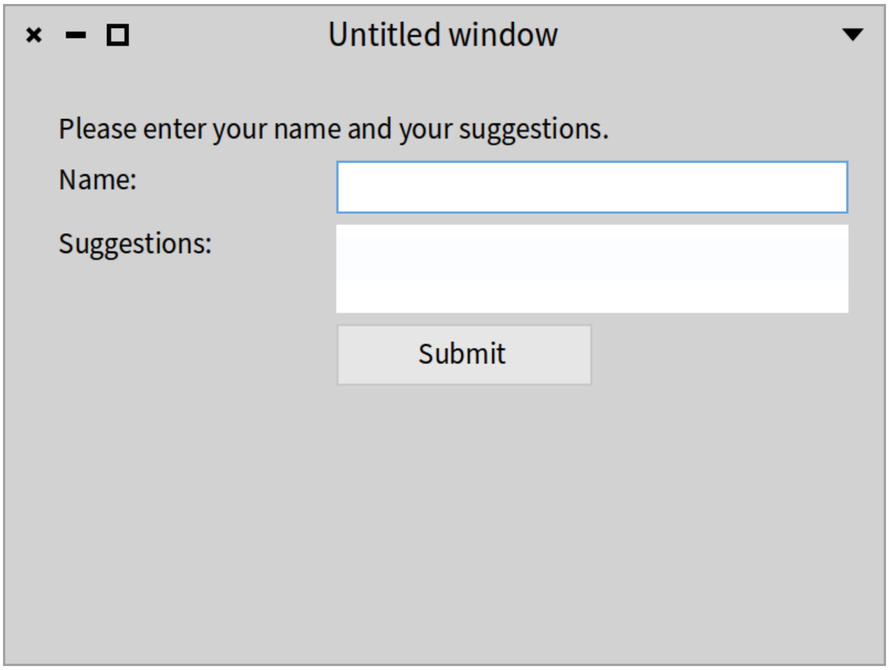


Figure 10-11 A simple grid for a small form. % width=60&anchor=GridExample

Here is a list of options:

- `columnHomogeneous`: Whether presenters in a column will have the same size.
- `rowHomogeneous`: Whether presenters a row will have the same size.
- `colSpacing:`: The horizontal space between cells.
- `rowSpacing:`: The vertical space between cells.

The `defaultLayout` method of the example maybe hard to read, especially when the grid contains a lot of presenters. The reader has to compute the positions and the spans of the subpresenters. We can use a `SpGridLayoutBuilder` to make grid building easier. The class is not to be used directly. Instead send `build:` to a `SpGridLayout`. Below is an alternative `defaultlayout` method that produces the same result as before. By putting all presenters of one row on one line, it is clear that there are four rows, and it is clear which subpresenters are part of the same row.

```
GridExample >> defaultLayout

  ^ SpGridLayout build: [ :builder |
    builder
      add: #promptLabel span: 3@1; nextRow;
      add: 'Name:'; add: #nameText span: 2@1; nextRow;
      add: 'Suggestions:'; add: #suggestionsText span: 2@1; nextRow;
      nextColumn; add: #submitButton ]
```

10.10 Paned layout (SpPanedLayout)

A paned layout is like a box layout, but restricted to two children, which are the "panes". It places children in a vertical or horizontal fashion and adds a splitter in between, that the user can drag to resize the panes. `positionOfSlider:` indicates the original position of the splitter. It can be nil (then it defaults to 50%), or it can be a percentage (e.g. 70 percent), a `Float` (e.g. 0.7), or a `Fraction` (e.g. 7/10).

Let's look at this simple example:

```
SpPresenter << #PanedLayoutExample
  slots: { #leftList . #rightList };
  package: 'CodeOfSpec20Book'
```

```
PanedLayoutExample >> initializePresenters

  leftList := self newList
    items: (1 to: 10);
    yourself.
  rightList := self newList
    items: ($a to: $z);
    yourself
```

```
PanedLayoutExample >> defaultLayout

  ^ SpPanedLayout newLeftToRight
    positionOfSlider: 70 percent;
    add: #leftList;
    add: #rightList;
    yourself
```

Let's open the presenter with:

```
PanedLayoutExample new open
```

Figure **??** shows the result. The left list takes 70% of the width of the window and the right list takes 30%.

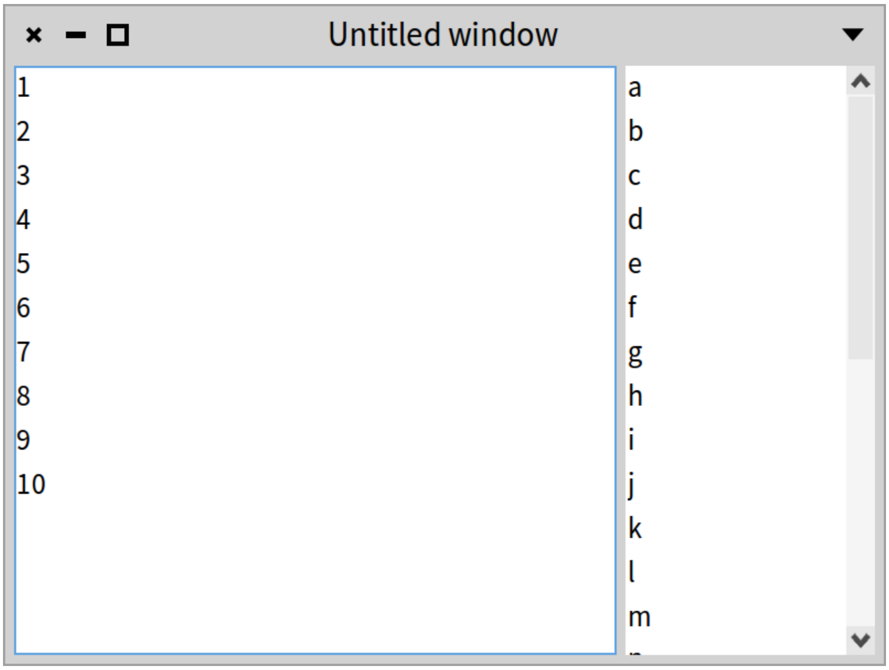


Figure 10-12 A paned layout with two lists. % width=50&anchor=PanedLayoutExample

10.11 Overlay layout (SpOverlayLayout)

An overlay layout allows overlaying one presenter by other presenters.

As an example, we will create a presenter that shows a button labeled 'Inbox', with a red indicator overlayed in the top-right corner. A use case could be indicating that there are unread messages in the inbox.

```
SpPresenter << #OverlayLayoutExample
  slots: { #button . #indicator };
  package: 'CodeOfSpec20Book'
```

`initializePresenters` creates the button and the indicator. The latter is a `SpRoassalPresenter`. We use a helper method to answer the shape that should be shown.

```
OverlayLayoutExample >> initializePresenters

  button := self newButton
    label: 'Inbox';
    yourself.
  indicator := (self instantiate: SpRoassalPresenter)
    script: [ :view | view addShape: self indicatorShape ];
    yourself
```

```
OverlayLayoutExample >> indicatorShape

  ^ RSBox new
      extent: 10@10;
      color: Color red;
      yourself
```

To make the structure of the layout clear, we have three methods. The `defaultLayout` is the layout of the window. For demonstration purposes, we put the button in the middle of the window. The button's dimensions are 50 by 50 pixels.

```
OverlayLayoutExample >> defaultLayout

  | buttonVBox |
  buttonVBox := SpBoxLayout newTopToBottom
      vAlignCenter;
      add: self buttonLayout height: 50;
      yourself.
  ^ SpBoxLayout newLeftToRight
      hAlignCenter;
      add: buttonVBox width: 50;
      yourself
```

The `defaultLayout` method sends `buttonLayout` to fetch the overlay layout for the button and the indicator. The `child` is the presenter that we want to overlay with the indicator. It is possible to add multiple overlays. In this example, we have only one, which is defined by `indicatorLayout`. Note that `addOverlay:withConstraints:` is used to configure where the overlay presenter should be displayed. We choose to display it in the top-right corner, by sending `vAlignStart` (top) and `hAlignEnd` (right).

```
OverlayLayoutExample >> buttonLayout

  ^ SpOverlayLayout new
      child: button;
      addOverlay: self indicatorLayout
        withConstraints: [ :constraints |
          constraints vAlignStart; hAlignEnd ];
      yourself
```

The `indicatorLayout` method defines the layout for the indicator. To apply a vertical and a horizontal padding, we have to wrap a vertical box layout with a horizontal box layout. We could have wrapped a horizontal box layout with a vertical box layout to achieve the same result. We apply a padding of 2 pixels so that the indicator does not overlap the border of the button.

```
OverlayLayoutExample >> indicatorLayout

  | counterVBox |
  counterVBox := SpBoxLayout newTopToBottom
      add: indicator withConstraints: [ :constraints |
        constraints height: 12; padding: 2 ];
      yourself.
  ^ SpBoxLayout newLeftToRight
      add: counterVBox withConstraints: [ :constraints |
        constraints width: 12; padding: 2 ];
      yourself
```

With all these methods in place, we can open the presenter.

```
OverlayLayoutExample new open.
```

That opens the window shown in Figure **??**.

10.12 Conclusion

Spec offers several predefined layouts. Probably new ones will be added but in a compatible way. An important closing point is that layouts can be dynamically composed. It means that you are able to design applications that can adapt to specific conditions.

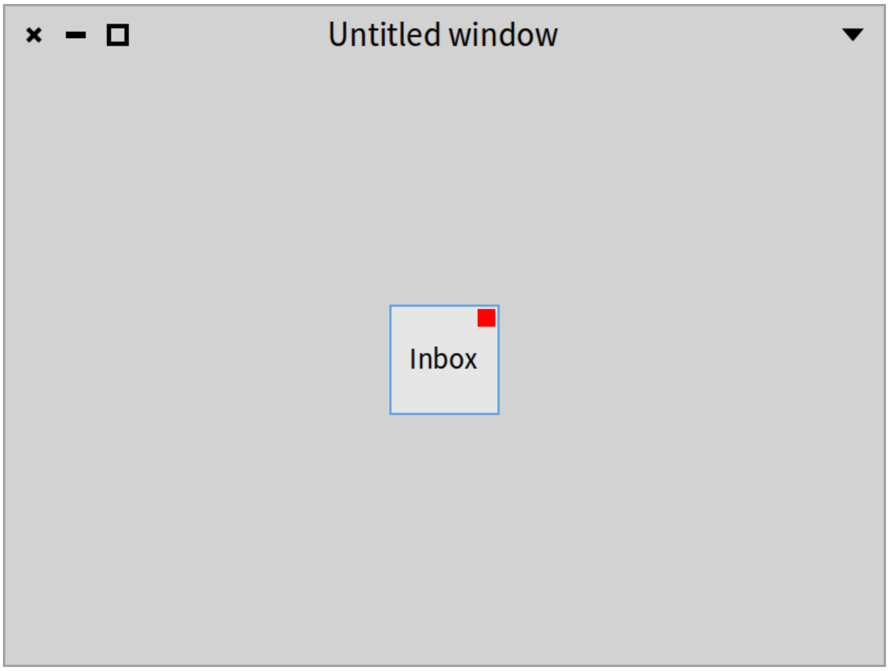


Figure 10-13 An overlay layout with a button and a Roassal box. %width=50&anchor=OverlayLayoutExample

CHAPTER 11

Dynamic presenters

Contrary to Spec 1.0, in Spec 2.0 all the layouts are dynamic. It means that you can change the displayed elements on the fly. It is a radical improvement from Spec 1.0 where most of the layouts were static and building dynamic widgets was cumbersome.

In this chapter, we will show that presenters can be dynamically composed using layouts. We will show a little interactive session. Then we will build a little browser with dynamic aspects.

11.1 Layouts as simple as objects

Building dynamic applications using Spec is simple. In fact, any layout in Spec is dynamic and composable. Let's explore how that works. We start with the following code snippet:

```
presenter := SpPresenter new.
presenter application: SpApplication new.
```

For this presenter, we will use the `SpPanedLayout` which can receive two presenters (or layouts) and place them in one half of the window. If you want to see all the available layouts in Spec, you can check the package `Spec2-Layout`.

```
presenter layout: SpPanedLayout newTopToBottom.
presenter open.
```

Of course, as shown in Figure **??**, we are going to see an empty window because we did not put anything in the layout.

Figure 11-1 An empty layout. % width=40&anchor=layout1

Now, without closing the window, we can dynamically edit the layout of the main presenter. We will add a button presenter by executing the following lines:

```
button1 := presenter newButton.
presenter layout add: button1.
button1 label: 'I am a button'.
```

Now we can add another button. There is no need to close and reopen the window. Everything updates dynamically and without the need of rebuilding the window. As we have instantiated the layout with `newTopToBottom`, the presenters will be laid out vertically. See Figure **??**.

```
button2 := presenter newButton.
presenter layout add: button2.
button2 label: 'I am another button'.
```

We can put an icon in the first button. See Figure **??**.

```
button1 icon: (button1 iconNamed: #smallDoIt).
```

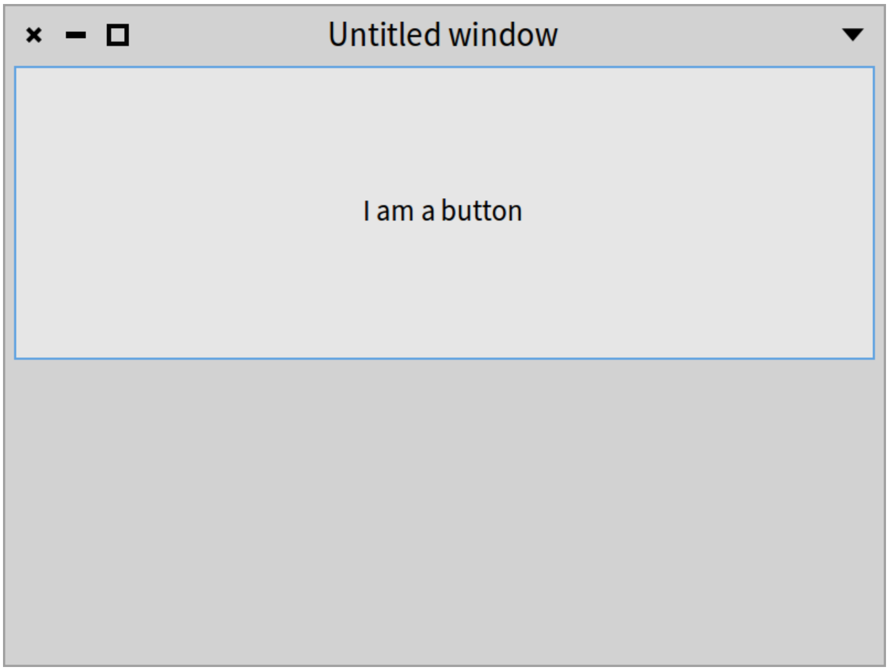


Figure 11-2 Paned layout with one button. %width=40&anchor=layout2

Or we can delete one of the buttons from the layout, as shown in Figure **??**.

```
presenter layout remove: button2.
```

What you see here is that all the changes happen simply by creating a new instance of a given layout and sending messages to it. It means that programs can define complex logic for the dynamic behavior of a presenter.

11.2 Creating a presenter that dynamically adds buttons with random numbers

We will create a presenter in which we will add and remove buttons dynamically. We will create a new class called `DynamicButtons`.

```
SpPresenter << #DynamicButtons
  slots: { #addButton . #removeButton . #text };
  package: 'CodeOfSpec20Book'
```

In `initializePresenters`, we add a button. When we click on it, it adds a new button to the layout. We also want a button that will remove the last button

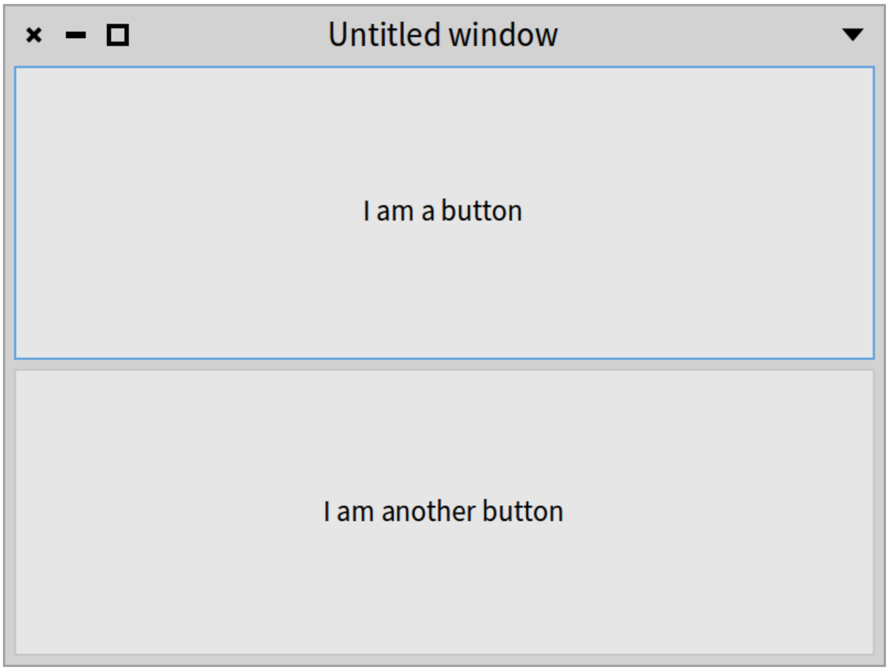


Figure 11-3 Paned layout with two buttons. % width=40&anchor=layout3

that was added, if any. Finally, we add a read-only text presenter that cannot be removed.

```
DynamicButtons >> initializePresenters

  addButton := self newButton.
  addButton
    action: [ self addToLayout ];
    label: 'Add a presenter to the layout';
    icon: (self iconNamed: #smallAdd).

  removeButton := self newButton.
  removeButton
    action: [ self removeFromLayout ];
    label: 'Remove a presenter from the layout';
    icon: (self iconNamed: #smallDelete);
    disable.

  text := self newText.
  text
    text: 'I am a text presenter.
```

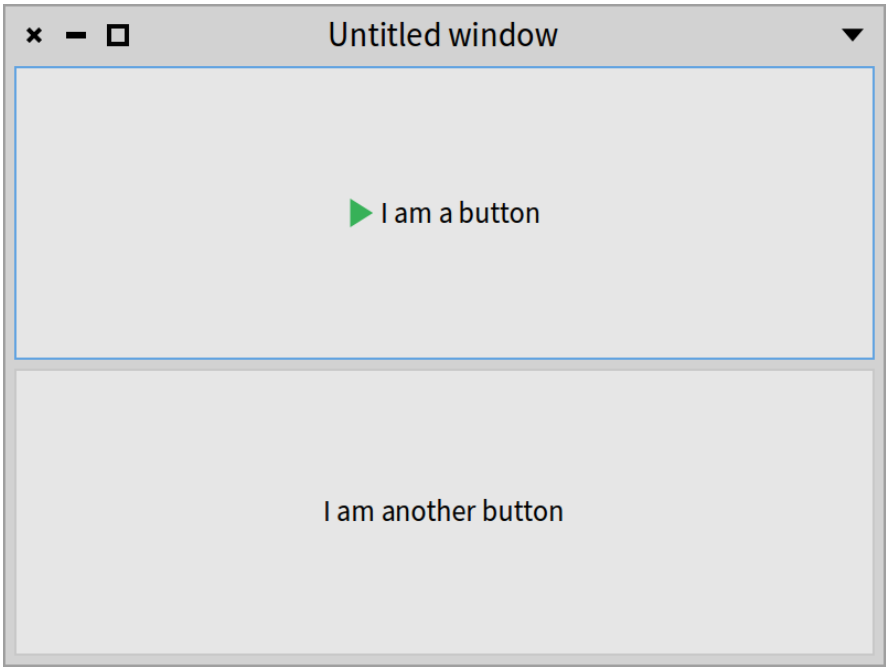


Figure 11-4 Paned layout with two buttons, one with an icon. % width=40&anchor=layout4

```
    I will not be removed';
    beNotEditable
```

Now we have to implement the methods `addToLayout` and `removeFromLayout` used in the action blocks of the buttons. Those methods, as their names indicate, add and remove presenters dynamically.

Let's start with the `addToLayout` method. We will add a new button to the layout. The label of the new button is a random number. We enable the remove button so that the newly added button can be removed.

```
DynamicButtons >> addToLayout

  | randomButtonName newButton |
  removeButton enable.
  randomButtonName := 'Random number: ', (Random new nextInteger:
    1000) asString.
  newButton := self newButton
    label: randomButtonName;
    icon: (self iconNamed: #smallObjects);
```

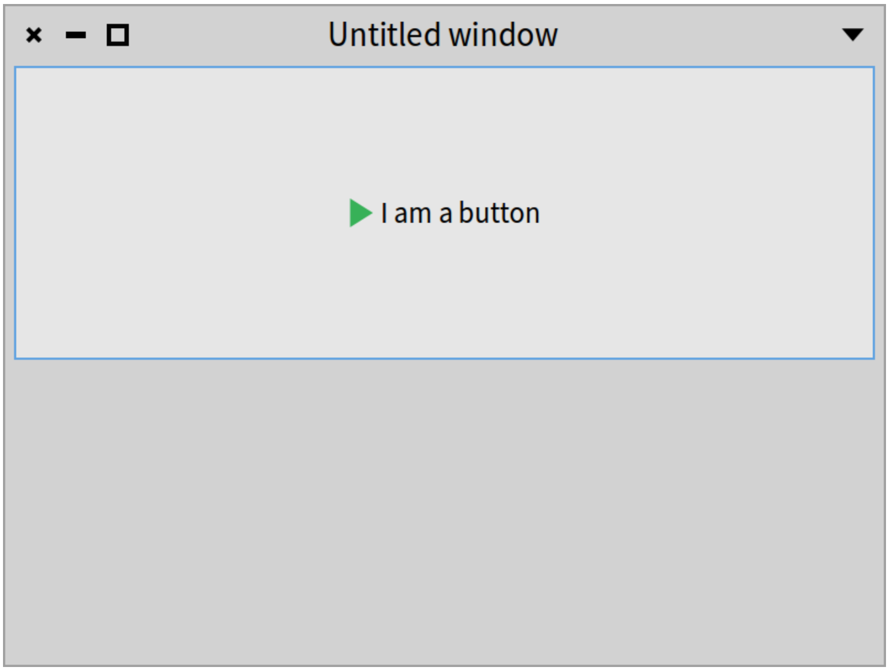


Figure 11-5 Removing a button. % width=40&anchor=layout5

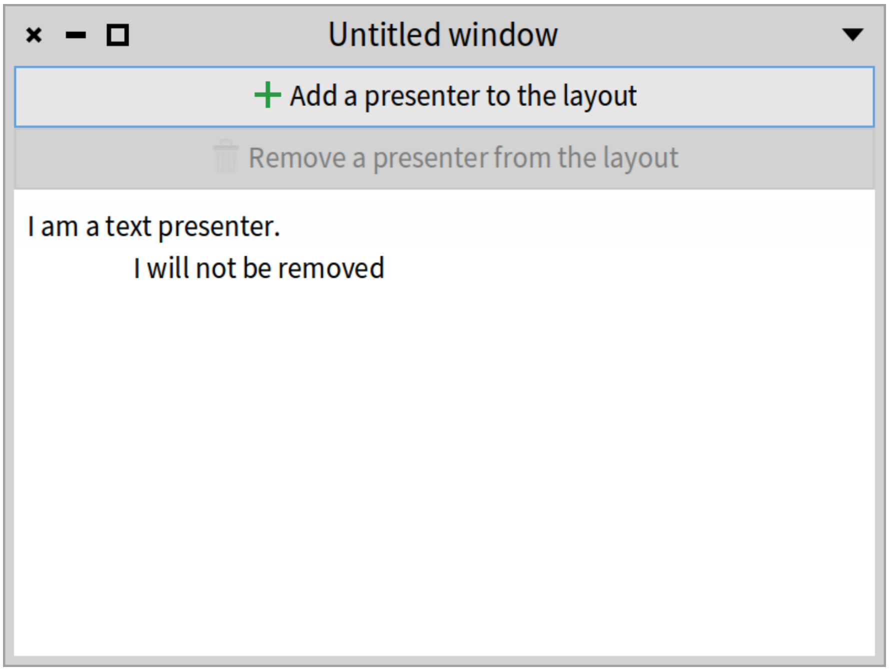
```
    yourself.
  self layout add: newButton expand: false
```

For removing a button from the layout, we will first check if there is a button that we can remove. If yes, we will just remove the last button. Then, if there are no more buttons left to remove, we will disable the remove button.

```
DynamicButtons >> removeFromLayout

  self layout remove: self layout presenters last.
  self layout presenters last = text ifTrue: [ removeButton disable ]
```

The only thing that is still missing is the default layout.

```
DynamicButtons >> defaultLayout

  ^ SpBoxLayout newTopToBottom
    add: addButton expand: false;
    add: removeButton expand: false;
    add: text;
    yourself
```


Untitled window
Add a presenter to the layout
Remove a presenter from the layout

I am a text presenter.

I will not be removed

Figure 11-6 A presenter that dynamically adds buttons. % width=40&anchor=layout6

After opening the window with the following code snippet, we see the window shown in Figure **??**.

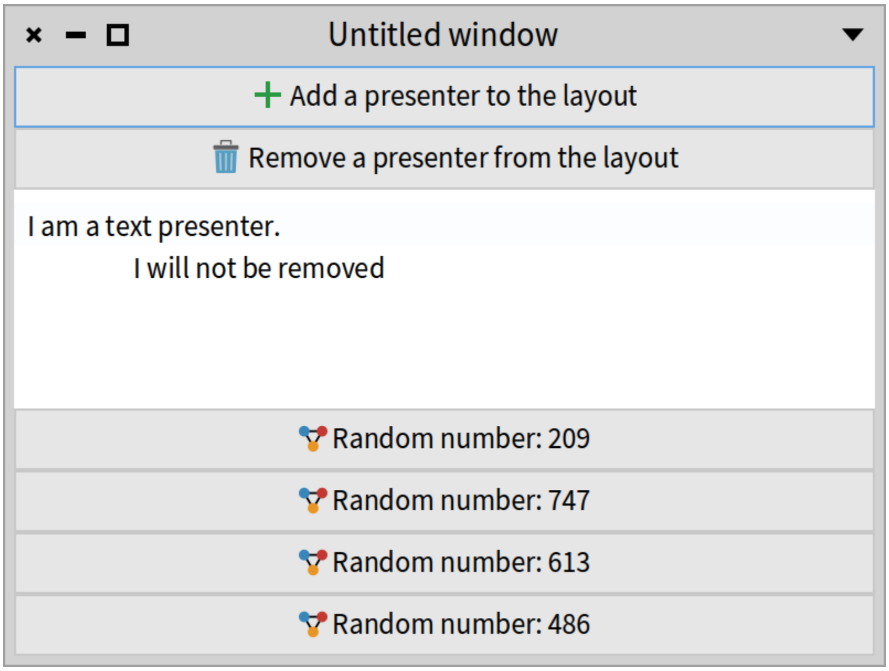
```
DynamicButtons new open
```

Figure **??** shows what the window looks like after clicking the add button four times.

11.3 Building a little dynamic browser

With all of the knowledge gained so far, we are going to build a new mini version of the System Browser as shown in Figure **??**. We want to have:

- A tree that shows all the system classes.
- A list that shows all methods of the selected class.
- A text presenter that shows the code of a selected method.
- A button.



I am a text presenter.
I will not be removed

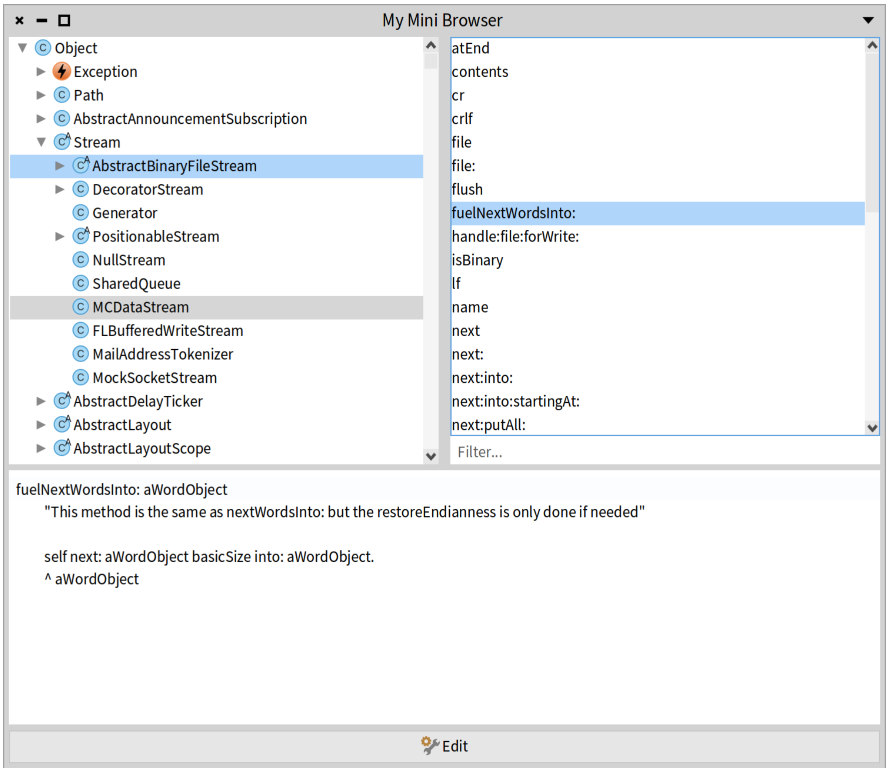
Figure 11-7 Adding random buttons. % width=40&anchor=layout7

Initially, the code of the method will be in "Read-only" mode. When we press the button, we are switching to "Edit" mode.

Let's get started.

```
SpPresenter << #MyMiniBrowser
  slots: { #classTree . #code . #methodList . #button };
  package: 'CodeOfSpec20Book'
```

The `initializePresenters` method instantiates the tree presenter class. We want the tree presenter to show all the classes that are present in the Pharo image. We know that (almost) all subclasses inherit from `Object`, so that is going to be the only root of the tree. To get the children of a tree node, we can send the message `subclasses` to a class. We want each of the tree nodes to have a nice icon. We can fetch the icon of a class with the message `systemIconName`. Finally, we want to "activate" the presenter with only one click instead of two.



```
fuelNextWordsInto: aWordObject
	"This method is the same as nextWordsInto: but the restoreEndianness is only done if needed"

	self next: aWordObject basicSize into: aWordObject.
	^ aWordObject
```

Figure 11-8 The mini browser in action. % width=60&anchor=layout8

```
MyMiniBrowser >> initializePresenters

  classTree := self newTree
    activateOnSingleClick;
    roots: { Object };
    children: [ :each | each subclasses ];
    displayIcon: [ :each | self iconNamed: each systemIconName ];
    yourself.
```

For the methods, we want to use a filtering list, so that we can search for method selectors. Also, we want to display only the selector of the method and sort the methods in an ascending way.

```
methodList := self newFilteringList display: [ :method |
  method selector ].
methodList listPresenter sortingBlock:
  [ :method | method selector ] ascending.
```

We said that, initially, the code is going to be in “Read-only” mode. The label of the button is going to be “Edit” to say that if we click on the button, we will change to “Edit” mode. We also want to have a nice icon.

```
button := self newButton
  label: 'Edit';
  icon: (self iconNamed: #smallConfiguration);
  yourself.
```

As the initial behavior will be read-only mode, the code will be a text presenter that is not editable.

```
code := self newText.
code beNotEditable
```

Here is the complete code of the method:

```
MyMiniBrowser >> initializePresenters

  classTree := self newTree
    activateOnSingleClick;
    roots: { Object };
    children: [ :each | each subclasses ];
    displayIcon: [ :each | self iconNamed: each systemIconName ];
    yourself.
  methodList := self newFilteringList display: [ :method |
    method selector ].
  methodList listPresenter sortingBlock:
    [ :method | method selector ] ascending.
  button := self newButton
    label: 'Edit';
    icon: (self iconNamed: #smallConfiguration);
    yourself.
  code := self newText.
  code beNotEditable
```

Opening the presenter with the code below, opens the window shown in Figure **??**.

```
MyMiniBrowser new open
```

11.4 Placing elements visually

We initialized our presenters, but we did not indicate how they need to be displayed.

We want the upper part of the layout to have the classes and the methods shown in a horizontal way, like in the System Browser. To achieve that, we will create

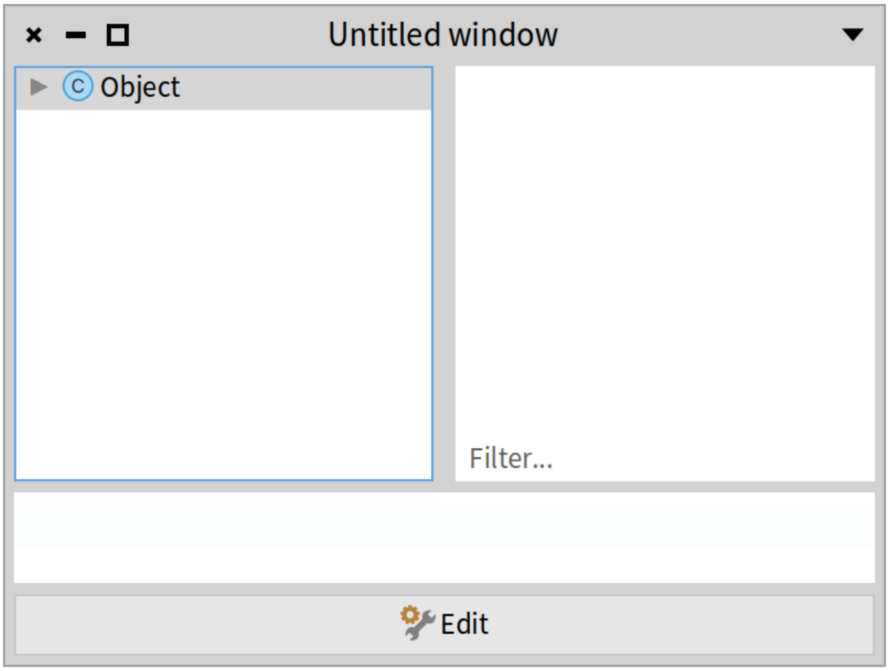


Figure 11-9 A little browser in read-only mode. %width=60&anchor=layout9

another left-to-right layout, with a spacing of 10 pixels between the classes and the methods.

We will add that layout to our main layout, which is a top-to-bottom layout. We add the code and the button under the classes and the methods. We do not want the code to expand. In addition, we want a separation of 5 pixels for this layout.

```
MyMiniBrowser >> defaultLayout

  | classesAndMethodsLayout |
  classesAndMethodsLayout := SpBoxLayout newLeftToRight.
  classesAndMethodsLayout
    spacing: 10;
    add: classTree;
    add: methodList.
  ^ SpBoxLayout newTopToBottom
    spacing: 5;
    add: classesAndMethodsLayout;
    add: code;
    add: button expand: false;
    yourself
```

11.5 Connecting the flow

So far so good, but we did not add any behavior to the presenters. We have to implement the `connectPresenters` method.

When we click on a class in the tree, we want to update the items of the method list with the methods of the selected class. When we click on a method, we want to update the text of the code with the source code of the method.

```
MyMiniBrowserPresenter >> connectPresenters

  classTree whenActivatedDo: [ :selection |
    methodList items: selection selectedItem methods ].
   methodList listPresenter
    whenSelectedDo: [ :selectedMethod |
      code text: selectedMethod ast formattedCode ].
  button action: [ self buttonAction ]
```

For now, we define the method `buttonAction` to do nothing.

```
MyMiniBrowserPresenter >> buttonAction
```

11.6 Toggling Edit/Read-only mode

When we click on the button we want several things. That is why it is better to create a separate method.

1. We want to change the label of the button to alternate between “Edit” and “Read only”.
2. We want to change the presenter of the code. If the Mini Browser is in read-only mode, we want to have a text presenter that is not editable. If the Mini Browser is in edit mode, we want to have a code presenter that applies syntax coloring to the code and shows the line numbers. But always the code is going to have the same text (the code of the selected method).

```
MyMiniBrowserPresenter >> buttonAction

  | newCode |
  button label = 'Edit'
    ifTrue: [
      button label: 'Read only'.
      newCode := self newCode
        beForMethod: methodList selectedItem;
        text: methodList selectedItem ast formattedCode;
      yourself ]
    ifFalse: [
      button label: 'Edit'.
      newCode := self newText
        text: methodList selectedItem ast formattedCode;
        beNotEditable;
        yourself ].

  self layout replace: code with: newCode.
  code := newCode
```

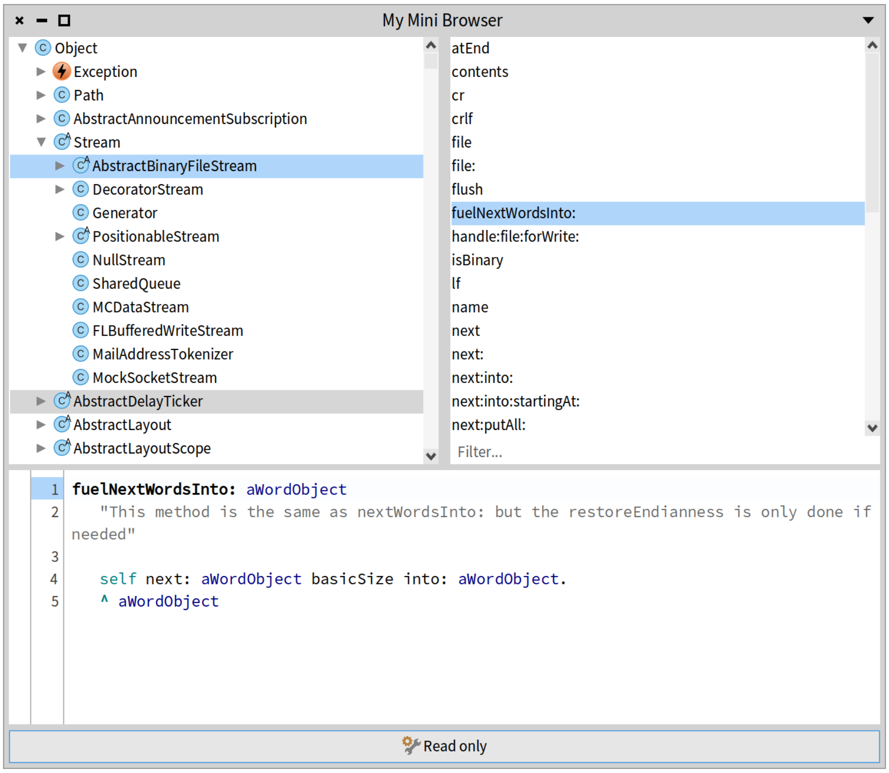
As a last detail, because we love details, we do not want “Untitled window” as the window title and we want a default extent. We define the `initializeWindow:` method.

```
MyMiniBrowserPresenter >> initializeWindow: aWindowPresenter

  aWindowPresenter
    title: 'My Mini Browser';
    initialExtent: 750@650
```

Voilà! We have a new minimal version version of the System Browser with a read-only mode. When we run `MyMiniBrowser new open`, and we select a class and a method, and we press the ’Edit’ button, we see the window in Figure **??**.

11.7 Conclusion

With Spec we can build applications ranging from very simple to very sophisticated. The dynamic layouts allow changing layouts on the fly. Layouts can be configured in multiple ways, so have a look at their classes and the available examples. Spec has lots of presenters that are ready to be used. Start digging into the code to see which presenters are available, and to learn their API.



```
fuelNextWordsInto: aWordObject
   "This method is the same as nextWordsInto: but the restoreEndianness is only done if
needed"

   self next: aWordObject basicSize into: aWordObject.
   ^ aWordObject
```

Figure 11-10 Our little browser in edit mode. % width=60&anchor=layout10

CHAPTER 12

Menu and menuBar

status: definitively needed

Soon in the best theater...

12.1 Menu

`contextMenu: self todoListContextMenu` sets the context menu to what is defined in the method `todoListContextMenu`. Let us study right now.

```
TodoListPresenter >> initializePresenters
    | addButton |
    todoListPresenter := self newTable
    addColumn: ((SpCheckBoxTableColumn evaluated: [:task | task
     isDone])
            width: 20;
            onActivation: [ :task | task done: true ];
            onDeactivation: [ :task | task done: false ];
            yourself);
    addColumn: (SpStringTableColumn
            title: 'Title'
            evaluated: [:task | task title);
    contextMenu: self todoListContextMenu;
    yourself.
```

```
TodoListPresenter >> todoListContextMenu

    ^ self newMenu
        addItem: [ :item | item
```

```
                        name: 'Edit...';
                        action: [ self editSelectedTask ] ];
        addItem: [ :item | item
                        name: 'Remove';
                        action: [ self removeSelectedTask ] ]
```

12.2 Menu Bar

12.3 ToolBar

How to create one that we can resue across components?

Attention un `SpToolBarButton` Family should be contained in a `SpToolBar`

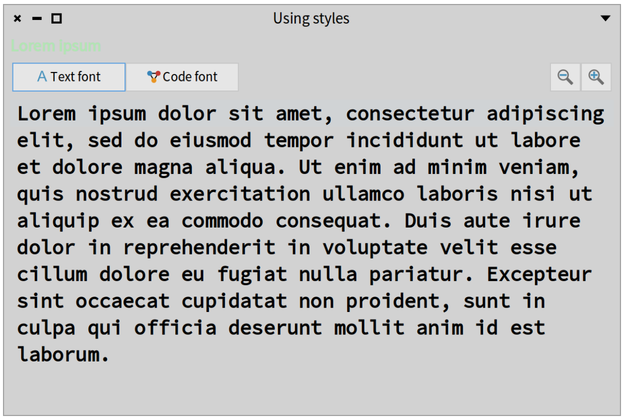
Homemade toolbar with SpButton

CHAPTER 13

Styling applications

In this chapter, we will describe how to use custom styles in Spec applications. First we present styles and then we will build a little editor like the one displayed hereafter.

We will show that an application in Spec manages styles and lets you adapt the look of a presenter as shown in Figure 13-1.



```
Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure
dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur
sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est
laborum.
```

Figure 13-1 Building a little styling editor.

We give some basis before showing how to effectively use styles to enhance the look and feel of an application.

13.1 How do styles work?

Styles in Spec work like CSS. They are stylesheets in which the properties for displaying a presenter are defined. Properties such as colors, width, height, font, and others. As a general principle, it is better to use styles instead of fixed constraints, because your application will be more responsive.

TODO Pay attention. A stylesheet does not cover all aspects of a widget.

13.2 About stylesheets

Spec first collects the style for the presenter, then collects the styles for its sub-presenters. 'application' is the default root level.

A defined stylesheet always has a root element and this root element has to be called `'.application'`.

Each style follows a cascading style, starting from `.application` like

```
.application.label.header
.application.link
.application.checkBox
```

There are two ways to express stylesheets: one for Morphic expressed using an extended version of STON, and CSS for GTK.

13.3 STON notation

Morphic styles can be declared using STON. STON is a textual object notation. It is described in a dedicated chapter in the *Enterprise Pharo* book available at https://books.pharo.org.

Each style element can use specific properties defined by associated classes:

- Geometry: `SpGeometryStyle`
- Draw: `SpDrawStyle`
- Font: `SpFontStyle`
- Container: `SpContainerStyle`
- Text: `SpTextStyle`

Example:

```
Geometry { #hResizing: true }
Draw { #color:  Color { #red: 1, #green: 0, #blue: 0, #alpha: 1}}
Draw { #color: #blue}
Font { #name: "Lucida Grande", #size: 10, #bold: true }
Container { #borderColor: Color { #rgb: 0, #alpha: 0 }, #borderWidth:
    2, #padding: 5 },
```

You can define your style globally, and add it to your specific presenter with the `addStyle:` message, for example `addStyle: 'section'`.

This message is specific to the `SpAbstractMorphicAdapter` backend. Here are two examples of stylesheets.

```
styleSheet
 ^ SpStyleSTONReader fromString: '
.application [
 Font { #name: "Source Sans Pro", #size: 10 },
 Geometry { #height: 25 },
 .label [
  Geometry { #hResizing: true },
  .headerError [Draw { #color:  Color{ #red: 1, #green: 0, #blue: 0,
    #alpha: 1}}  ],
  .headerSuccess [Draw { #color: Color{ #red: 0, #green: 1, #blue: 0,
    #alpha: 1}}  ],
  .header [
   Draw { #color: Color{ #rgb: 622413393 }},
   Font { #name: "Lucida Grande", #size: 10, #bold: true } ],
  .shortcut [
   Draw { #color: Color{ #rgb: 622413393 } },
   Font { #name: "Lucida Grande", #size: 10 } ],
  .fixed [
   Geometry { #hResizing: false, #width: 100 } ],
  .dim [
   Draw { #color : Color{ #rgb: 708480675 } } ]
 ],
 …
'
```

The next one extends the default stylesheet.

```
styleSheet
 ^ SpStyle defaultStyleSheet, (SpStyleSTONReader
  fromString:
   '
.application [
 Draw { #backgroundColor: #lightRed},
 .section [
   Draw { #color: #green, #backgroundColor: #lightYellow},
   Font {  #name: "Verdana", #size: 12, #italic: true, #bold: true}],
 .disabled [ Draw { #backgroundColor: #lightGreen} ],
 .textInputField [ Draw { #backgroundColor: #blue} ],
 .label [
  Font {  #name: "Verdana", #size: 10, #italic: false, #bold: true},
  Draw { #color: #red, #backgroundColor: #lightBlue} ] ]
')
```

13.4 Anatomy of a style

The styles in Spec format are similar to CSS. We have to write the styles as a string and then parse it as a STON file.

Here is an example that we will explain step by step.

```
'.application [
    .lightGreen [ Draw { #color: #B3E6B5 } ],
    .lightBlue [ Draw { #color: #lightBlue } ] ]'
```

`SpPropertyStyle` has 5 subclasses: `SpContainerStyle`, `SpDrawStyle`, `Sp-FontStyle`, `SpTextStyle`, and `SpGeometryStyle`. These subclasses define the 5 types of properties that exist. On the class side, the method `stonName` indicates the name that we must put in the STON file. stonName above is unclear

- `SpDrawStyle` modifies the properties related to the drawing of the presenter, such as the color and the background color.
- `SpFontStyle` manipulates properties related to fonts.
- `SpGeometryStyle` is for sizes, like width, height, minimum height, etc.
- `SpContainerStyle` is for the alignment of the presenters, usually with property is changed on the main presenter, which is the one that contains and arranges the other ones.
- `SpTextStyle` controls the properties of the `SpTextInputFieldPresen-ter`.

If we want to change the color of a presenter, we need to create a string and use the `SpDrawStyle` property, which STON name is `Draw` as shown below. For setting the color, we can use either the hexadecimal code of the color or the sender of `Color` class.

```
'.application [
    .lightGreen [ Draw { #color: #B3E6B5 } ],
    .lightBlue [ Draw { #color: #lightBlue } ] ]'
```

Now we have two styles: `lightGreen` and `lightBlue` that can be applied to any presenter.

13.5 Environmental variables

We can also use environmental variables to get the values of the predefined colors and fonts of the current theme. For example, we can create two styles for changing the font of the text of a presenter:

```
'.application [
    .codeFont [ Font { #name: EnvironmentFont(#code) } ],
    .textFont [ Font { #name: EnvironmentFont(#default) } ]
]'
```

Also we can change the styles for all the presenters by default. For instance, we can display all the text in bold by default.

```
'.application [
    Font { #bold: true }
]'
```

unclear what is EnvironmentFont vs Font What are the environmental variables

13.6 Defining an application

To use styles we need to associate the main presenter with an application. The class `SpApplication` already has default styles. To not redefine all the properties for all the presenters, we can concatenate the default styles (`SpStyle defaultStyleSheet`) with our own.

To parse a string into a STON we use the class `SpStyleVariableSTONReader`.

```
presenter := SpPresenter new.
app := SpApplication new.
presenter application: app.

styleSheet := SpStyle defaultStyleSheet,
    (SpStyleVariableSTONReader fromString:
    '.application [
         Font { #bold: true },
            .red [ Draw { #color: #red } ],
            .bgGray [ Draw { #backgroundColor: #E2E2E2 } ],
        .blue [ Draw { #color: #blue } ]
]' ).

app styleSheet: styleSheet.
```

Now we can add styles to a presenter as follows, and whose result is shown in Figure 13-2.

```
label := presenter newLabel.
presenter layout: (SpBoxLayout newTopToBottom
    add: label;
    yourself).

label label: 'I am a label'.
label addStyle: 'red'.
label addStyle: 'bgGray'.

presenter open
```

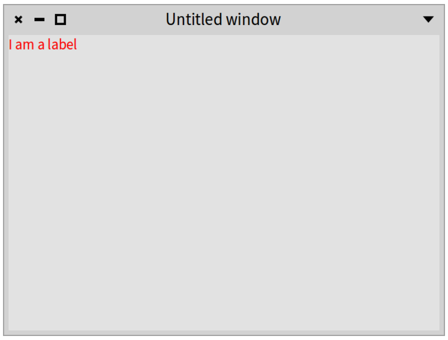


Figure 13-2 The stylesheet had been applied to the label.

13.7 Dynamically applying styles

We can also remove and add styles at runtime as shown in the following snippet whose result is displayed in Figure 13-3.

```
label removeStyle: 'red'.
label removeStyle: 'bgGray'.
label addStyle: 'blue'.
```

13.8 Now using classes

Until now we just wrote scripts. Now we want to show how we can use styles using presenter classes. To properly use styles, it is better to define a custom application as a subclass of `SpApplication`. How do we associate an application to a presenter?

```
SpApplication << #CustomStylesApplication
  slots: {};
  package: 'CodeOfSpec20Book'
```

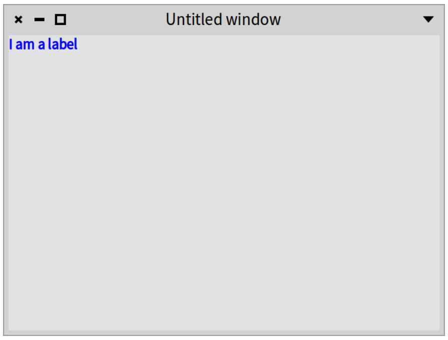


Figure 13-3 After changing the style.

In the class, we override the method `styleSheet` to return our custom stylesheet concatenated with the default one.

```
CustomStylesApplication >> styleSheet

  | customStyleSheet |
  customStyleSheet := SpStyleVariableSTONReader fromString:
    '.application [
      Font { #bold: true },
      .lightGreen [ Draw { #color: #B3E6B5 } ],
      .lightBlue [ Draw { #color: #lightBlue } ],
      .container [ Container { #padding: 4, #borderWidth: 2 } ],
      .bgOpaque [ Draw { #backgroundColor: EnvironmentColor(#base) } ],
      .codeFont [ Font { #name: EnvironmentFont(#code) } ],
      .textFont [ Font { #name: EnvironmentFont(#default) } ],
      .bigFontSize [ Font { #size: 20 } ],
      .smallFontSize [ Font { #size: 14 } ],
      .icon [ Geometry { #width: 30 } ],
      .buttonStyle [ Geometry { #width: 110 } ],
      .labelStyle [
        Geometry { #height: 25 },
        Font { #size: 12 } ]
    ]'.
  ^ SpStyle defaultStyleSheet , customStyleSheet
```

We can use different properties in the same style. For example, in `labelStyle` we are setting the height of the presenter to 25 scaled pixels and the font size to 12 scaled pixels. Also, we are using `EnvironmentColor(#base)` for obtaining the default background color according to the current theme, because the color will change according to the theme that is used in the image.

13.9 Defining a presenter for the editor

For the main presenter, we will build a mini text viewer in which we will be able to change the size and font of the text that we are viewing.

```
SpPresenter << #CustomStyles
  slots: { #text . #label . #zoomOutButton . #textFontButton .
    #codeFontButton . #zoomInButton };
  package: 'CodeOfSpec20Book'
```

In the initializePresenters method we will first initialize the presenters and then set the styles for the presenters.

```
CustomStyles >> initializePresenters

  self instantiatePresenters.
  self initializeStyles
```

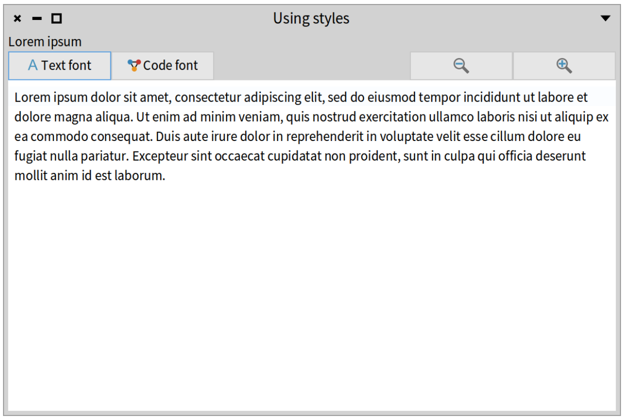
```
CustomStyles >> instantiatePresenters

  zoomInButton := self newButton.
  zoomInButton icon: (self iconNamed: #glamorousZoomIn).
  zoomOutButton := self newButton.
  zoomOutButton icon: (self iconNamed: #glamorousZoomOut).
  codeFontButton := self newButton.
  codeFontButton
    icon: (self iconNamed: #smallObjects);
    label: 'Code font'.
  textFontButton := self newButton.
  textFontButton
    icon: (self iconNamed: #smallFonts);
    label: 'Text font'.
  text := self newText.
  text
    beNotEditable
    clearSelection;
    text: String loremIpsum.
  label := self newLabel.
  label label: 'Lorem ipsum'
```

```
CustomStyles >> defaultLayout

  | buttonbar |
  buttonbar := SpBoxLayout newLeftToRight
    add: textFontButton expand: false;
    add: codeFontButton expand: false;
    addLast: zoomOutButton expand: false;
    addLast: zoomInButton expand: false;
    yourself.
  ^ SpBoxLayout newTopToBottom
    add: label expand: false;
    add: buttonbar expand: false;
    add: text;
    yourself
```

Finally, we change the window title and size:

```
CustomStyles >> initializeWindow: aWindowPresenter

  aWindowPresenter
    title: 'Using styles';
    initialExtent: 600 @400
```

Without setting the custom styles nor using our custom application in the presenter, we obtain Figure 13-4, assuming that the "Pharo Light" theme is in effect:



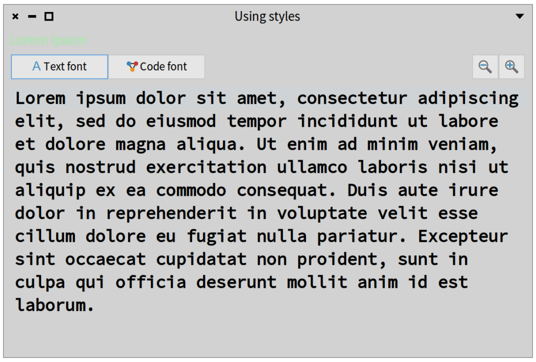
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 13-4 Styling.

13.10 Initializing styles

We do not want the black background color for the text presenter. We would like to have a sort of multi-line label. We want the zoom buttons to be smaller as they only have icons. We want to have the option to change the size and font of the text inside the text presenter. Finally, we want to change the color of the label, change its height and make it a little bit bigger.

```
CustomStyles >> initializeStyles
    "Change the height and size of the label and the color as
    ligthgreen"

  label addStyle: 'labelStyle'.
  label addStyle: 'lightGreen'.
  "The default font of the text will be the code font and the font
    size will be the small one."
  text addStyle: 'codeFont'.
  text addStyle: 'smallFontSize'.
  "Change the background color."
  text addStyle: 'bgOpaque'.
  "Use a smaller width for the zoom buttons"
  zoomInButton addStyle: 'icon'.
  zoomOutButton addStyle: 'icon'.
  codeFontButton addStyle: 'buttonStyle'.
  textFontButton addStyle: 'buttonStyle'.
  "As this presenter is the container, set to self the container style
    to add a padding and border width."
  self addStyle: 'container'
```



```
Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure
dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur
sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est
laborum.
```

Figure 13-5 Styled editor.

Finally, we have to override the `start` method in the application. We are going to set the application of the presenter and run the presenter from the application.

```
CustomStylesApplication >> start

  (self new: CustomStyles) open
```

Now, when we run `CustomStylesApplication new start` we will obtain Figure 13-5.

13.11 Wiring buttons

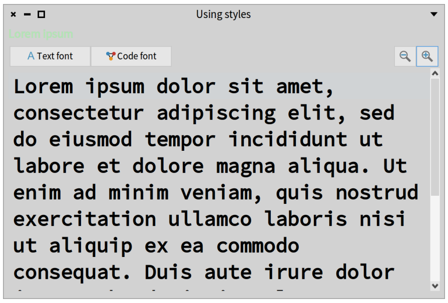
The only thing missing is to add the behavior of the buttons.

For example, if we click on the zoom-in button we want to remove the `smallFontStyle` and add the `bigFontSize`. When we click on the text font button, we want to remove the style `codeFont` and add the `textFont` style.

This is what we have to do in the `connectPresenters` method:

```
CustomStyles >> connectPresenters

  zoomInButton action: [
    text removeStyle: 'smallFontSize'.
    text addStyle: 'bigFontSize' ].
  zoomOutButton action: [
    text removeStyle: 'bigFontSize'.
    text addStyle: 'smallFontSize' ].
  codeFontButton action: [
    text removeStyle: 'textFont'.
    text addStyle: 'codeFont' ].
  textFontButton action: [
    text removeStyle: 'codeFont'.
    text addStyle: 'textFont' ]
```

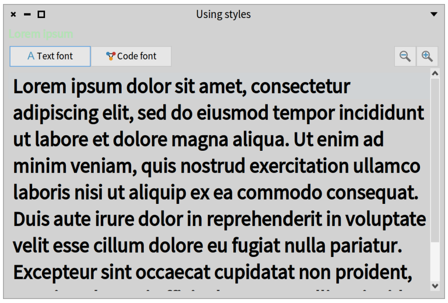
When we click on the the zoom-in button, the size of the text changes as shown in Figure 13-6.



```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed
do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi
ut aliquip ex ea commodo
consequat. Duis aute irure dolor
```

Figure 13-6 Zoomed styled editor.

When we click the ”Text font” button, the font of the text changes as shown in Figure 13-7.



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident,

Figure 13-7 Styled editor with other font.

13.12 Spec implementation details

You can ask an adapter for its style name using the message `styleName`

```
SpMorphicLabelAdapter styleName
> Label
```

13.13 Conclusion

Using styles in Spec is great. It makes it easier to have a consistent design as we can add the same style to several presenters. If we want to change some style, we only edit the stylesheet. Also, the styles automatically scale if we change the font size of all the images. These are the main reasons why in Spec we have the notion of an application. We can dynamically change how a presenter looks.

CHAPTER 14

Using transmissions and ports (Draft)

Transmissions are a more compact way to connect presenters than events as shown previously.

14.1 What are transmissions?

Transmissions are a way to connect presenters, thinking about the “flow” of information more than the way it is displayed. Each presenter defines **output ports** (ports to send information) and **input ports** (ports to receive information). There are at least one default input port and one default output port. A transmission connects a presenter’s output port with a presenter’s input port.

For example, think on an overview-detail (O->D) relationship, when you navigate the elements in the overview O, you want to see the detail D. This is typically solved by showing a list with list elements and a form with the detail of an element. In Spec, this will be declared more or less like this:

```
list := self newList.
detail := self newText.
```

- Input ports define the transmission destination points of a presenter. They handle an incoming transmission and transmit them properly to the target presenter.
- An output port defines origin actions (and the possible data associated to such action) to transmit to a destination (input) port. It also defines the

transformations to apply to the output data before giving them to the input port.

14.2 Transmitting from an output port to an input port

A transmission connects a presenter's output port with a presenter's input port as shown in the following example:

```
list transmitTo: detail.
```

This connects the list presenter default output port with the detail presenter default input port.

14.3 Transforming a transmission

The object transmitted from a presenter output port can be inadequate for the input port. To solve this problem a transmission offers the possibility to transform the transmitted object.

This is as simple as using the `transform:` protocol:

SD: is the method called trasmitTo:transform: or juts transmitTo:

```
list
    transmitTo: detail
    transform: [ :aValue | aValue asString ].
```

14.4 Transmitting from an output port to an arbitrary input receiver

It is possible that the user requires to listen an output port, but instead transmitting the value to another presenter, other operation is needed.

todo christophe I do not get the previous sentence

There is the `transmitDo:` protocol to handle this situation:

```
list transmitDo: [ :aValue | aValue crTrace ].
```

14.5 Acting after a transmission

Sometimes, after a transmission happens, the user needs to react to modify something given the new status achieved by the presenter (like, pre-selecting something). The `postTransmission:` protocol allows you to handle that situation.

```
list
    transmitTo: detail
    postTransmission: [ :fromPresenter :toPresenter :value |
        "something to do here"
        toPresenter enabled: value isEmptyOrNil not ].
```

CHAPTER 15

Integration of Athens in Spec

This chapter was originally written by Renaud de Villemeur. We thank him for his contribution. It shows how you can integrate vector graphic drawing within Spec components.

15.1 Introduction

There are two different computer graphics: vector and raster graphics. Raster graphics represent images as a collection of pixels. Vector graphics is the use of geometric primitives such as points, lines, curves, or polygons to represent images. These primitives are created using mathematical equations.

Both types of computer graphics have advantages and disadvantages. The advantages of vector graphics over raster are:

- smaller size,
- ability to zoom indefinitely,
- moving, scaling, filling, and rotating do not degrade the quality of an image.

Ultimately, pictures on a computer are displayed on a screen with a specific display dimension. However, while raster graphic doesn't scale very well when the resolution differs too much from the picture resolution, vector graphics are rasterized to fit the display they will appear on. Rasterization is the technique of taking an image described in a vector graphics format and transforming it into a set of pixels for output on a screen.

Note. You have the same concept when doing 3D programming with an API like OpenGL. You describe your scene with points, vertices, etc, and in the end, you rasterize your scene to display it on your screen.

Morphic is the way to do graphics with Pharo. However, most existing canvases are pixel-based, and not vector-based. This can be an issue with current IT ecosystems, where the resolution can differ from machine to machine (desktop, tablet, phones, etc).

Enter Athens, a vector-based graphic API. Under the hood, it uses the Cairo graphic library for the rasterization phase.

When you integrate Athens with Spec, you'll use its rendering engine to create your picture. It is transformed into a `Form` and displayed on the screen.

15.2 Hello world in Athens

We will see how to use Athens directly integrated with Morphic. This is why we create a `Morph` subclass. Figure **??** shows the display of such a morph. It will be the class we will use for all our experiments.

First, we define a class which inherits from `Morph`:

```
Morph << #AthensHello
  slots: { #surface };
  package: 'CodeOfSpec20Book'
```

During the initialization phase, we create an Athens surface:

```
AthensHello >> initialize

  super initialize.
  self extent: self defaultExtent.
  surface := AthensCairoSurface extent: self extent
```

where `defaultExtent` is simply defined as

```
AthensHello >> defaultExtent

  ^ 400@400
```

The `drawOn:` method, mandatory in `Morph` subclasses, asks Athens to render its drawing and it will then display it in a Morphic canvas as a `Form` (a bitmap picture)

```
AthensHello >> drawOn: aCanvas

  self renderAthens.
  surface displayOnMorphicCanvas: aCanvas at: bounds origin
```

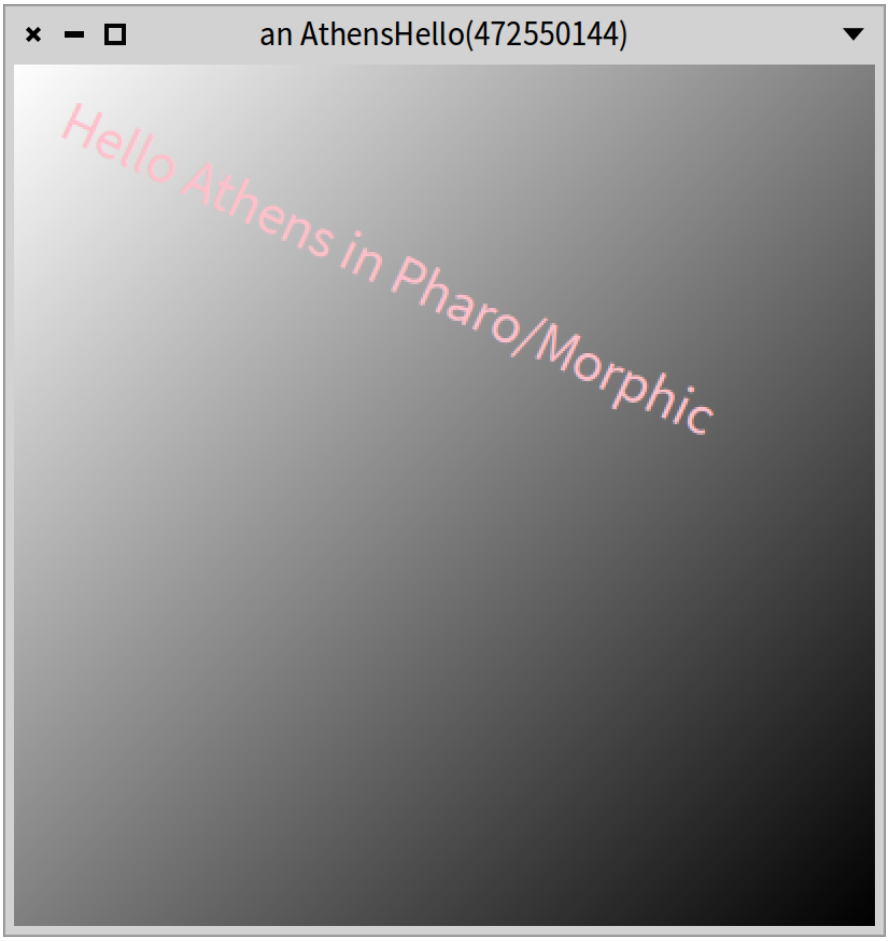


Figure 15-1 AthensHello new openInWindow. %width=60&label=athens

Our actual Athens code is located in the `renderAthens` method, and the result is stored in the `surface` instance variable.

```
AthensHello >> renderAthens

  | font |
  font := LogicalFont familyName: 'Arial' pointSize: 10.
  surface drawDuring: [ :canvas |
    surface clear.
    canvas setPaint: ((LinearGradientPaint from: 0@0  to: self extent)
     colorRamp: {  0 -> Color white. 1 -> Color black }).
    canvas drawShape: (0@0 extent: self extent).
    canvas setFont: font.
    canvas setPaint: Color pink.
    canvas pathTransform translateX: 20 Y: 20 + (font
     getPreciseAscent); scaleBy: 2; rotateByDegrees: 25.
    canvas drawString: 'Hello Athens in Pharo/Morphic' ]
```

Open the morph in a window with:

```
AthensHello new openInWindow
```

15.3 Handling resizing

You can already create the window and see a nice gradient with a greeting text. However, you will notice that when resizing the window, the Athens content is not resized. To fix this, we need one extra method.

```
AthensHello >> extent: aPoint

  | newExtent |
  newExtent := aPoint rounded.
  (bounds extent closeTo: newExtent) ifTrue: [ ^ self ].
  bounds := bounds topLeft extent: newExtent.
  surface := AthensCairoSurface extent: newExtent.
  self layoutChanged.
  self changed
```

Congratulations, you have now created your first morphic window where content is rendered using Athens.

15.4 Using the morph with Spec

Now that we have a morph, we can use it in a presenter as follows.

```
SpPresenter << #AthensHelloPresenter
  slots: { #morphPresenter };
  package: 'CodeOfSpec20Book'
```

We define a basic layout so that Spec knows where to place it.

```
AthensHelloPresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
     add: morphPresenter;
     yourself
```

In `initializePresenters` we wrap the morph in a `SpMorphPresenter`.

```
AthensHelloPresenter >> initializePresenters

  morphPresenter := self instantiate: SpMorphPresenter.
  morphPresenter morph: AthensHello new
```

When we open the presenter it displays the morph:

```
AthensHelloPresenter new open
```

15.5 Direct integration of Athens with Spec

We can also achieve a direct integration without relying on a specific Morph creation.

We first create a presenter named `AthensExamplePresenter`. This is the presenter that will support the actual rendering using Athens.

```
SpPresenter << #AthensExamplePresenter
  slots: { #paintPresenter };
  package: 'CodeOfSpec20Book'
```

We define a simple layout to place the `paintPresenter`.

```
AthensExamplePresenter >> defaultLayout

  ^ SpBoxLayout newTopToBottom
     add: paintPresenter;
     yourself
```

This presenter wraps an `AthensPresenter` as follows:

```
AthensExamplePresenter >> initializePresenters

  paintPresenter := self instantiate: SpAthensPresenter.
  paintPresenter surfaceExtent: 600@400.
  paintPresenter drawBlock: [ :canvas | self render: canvas ]
```

It configures the `AthensPresenter` to draw with the `render:` message.

```
AthensExamplePresenter >> render: canvas

  canvas
    setPaint:
      (canvas surface
        createLinearGradient: {
          0 -> Color white.
          1 -> Color black }
        start: 0@0
        stop: canvas surface extent).
  canvas drawShape: (0 @ 0 extent: canvas surface extent)
```

Executing `AthensExamplePresenter new open` produces Figure **??**.

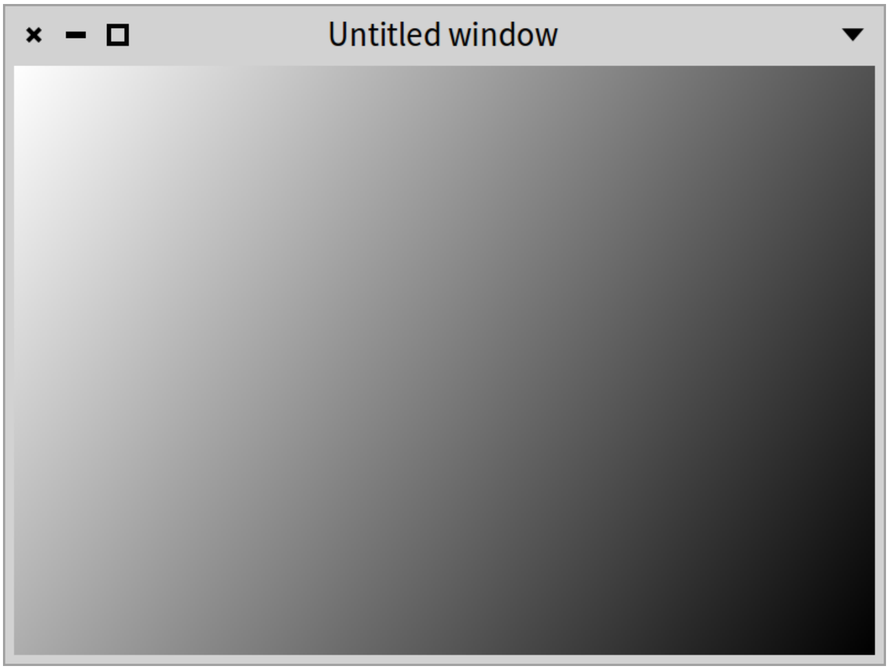


Figure 15-2 A Spec application with an Athens presenter. % width=60&label=athens2

This example is simple because we did not cover the rendering that may have to be invalidated if something changes, but it shows the key aspect of the architecture.

15.6 Conclusion

This chapter illustrated clearly that Spec can take advantage of canvas-related operations such as those proposed by Athens to open the door to specific visuals.

CHAPTER 16

Customizing your Inspector

status: should do another pass status: spellchecked

An Inspector is a tool that is used to look at and interact with objects. In Pharo, inspecting an object means opening this tool and interacting with your object. It is a key tool when developing in Pharo. It allows one to navigate the object structure, look at the state of the variables, change their value, or send messages. An inspector can show other information and you can extend it to display the information that is best suited for you. This is what we will see in this chapter.

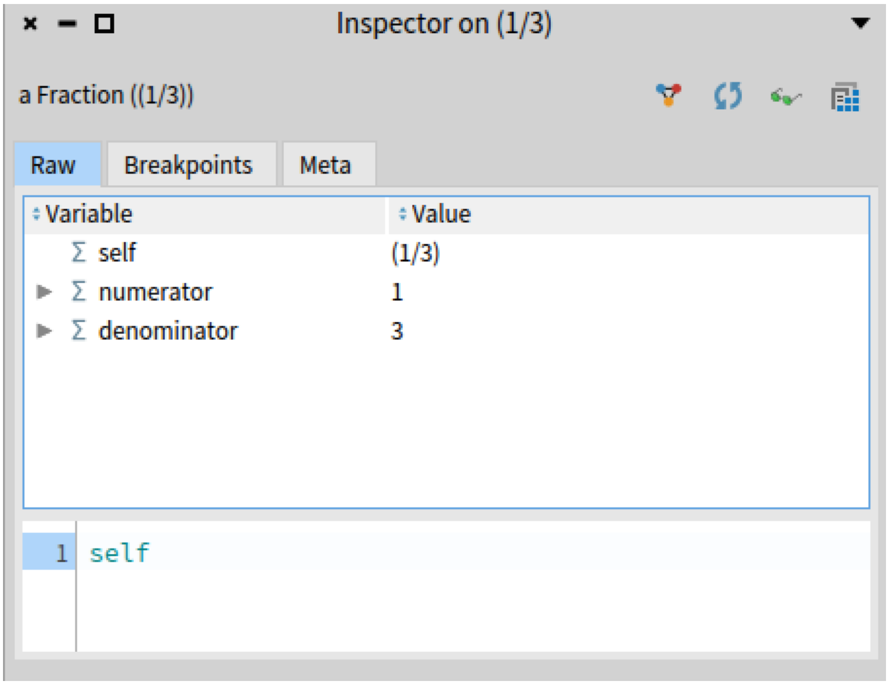
16.1 A first look at the inspector

You can inspect the result of an execution by selecting the code and using the shortcut `Cmd/ctrl + i` or `right-click + "Do & inspect it"`. This will execute the code and open an inspector on the result.

By inspecting `1/3` we get the inspector shown in Figure **??**.

There are three areas in an inspector. They are highlighted in Figure **??**.

1. This text starts with the **class** of the inspected object. Here we have an instance of Fraction.
2. This is the raw view on the object. It shows the internal state of the object. Here the fraction has a `numerator` instance variable holding the value `1`, and a `denominator` instance variable holding the value `3`.
3. The last area is the evaluator. In this area, you can write expressions and evaluate them like you would in the playground. In an evaluator, `self`


Inspector on (1/3)

a Fraction ((1/3))

Variable	Value
self	(1/3)
numerator	1
denominator	3

```
self
```

Figure 16-1 An inspector with the Raw tab selected. % width=60&anchor=InspectorWithRawTab

refers to the inspected object. This object can be seen in the raw view above the evaluator showing the value of the `self` variable.

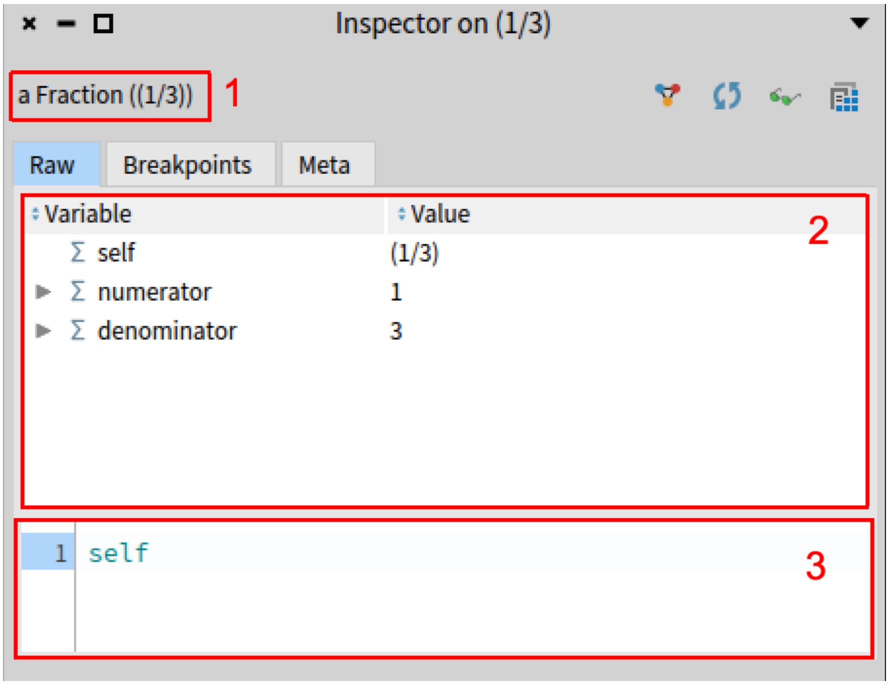
The raw view on the object is a tree list. By clicking the small grey triangle on the left side, you can unfold the state of the object held by the instance variable. By clicking on an instance variable, you open a new inspector pane.

This is recursive: if you click on more variables, more panes will open. By default, only the last two panes are visible at any time. You can use the small rectangles at the bottom of the window to navigate between panes.

KDH: adding a GIF file does not work. We need an alternative or remove this part

16.2 The inspector toolbar

Each inspector has the toolbar shown in Figure 16-3:



Variable	Value
self	(1/3)
numerator	1
denominator	3

```
self
```

Figure 16-2 Inspector areas. % width=60&anchor=InspectorWithThreeAreas

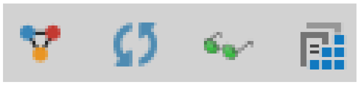


Figure 16-3 Toolbar.

- The **triangle button** is related to object-centric debugging. It allows putting breakpoints on state access (read and/or write) of a specific object.
- The **circling arrows** button allows refreshing the current view of the object. Fields of an object are not updated live, so if the object is modified from elsewhere, the new values will only show after this button is used.
- The **green glasses** button opens another inspector window on the current object.
- The **last button** allows opening a browser on the class of the inspected object. It can be used to check for available methods to use in the evaluator.

16.3 The Breakpoints tab: managing breakpoints

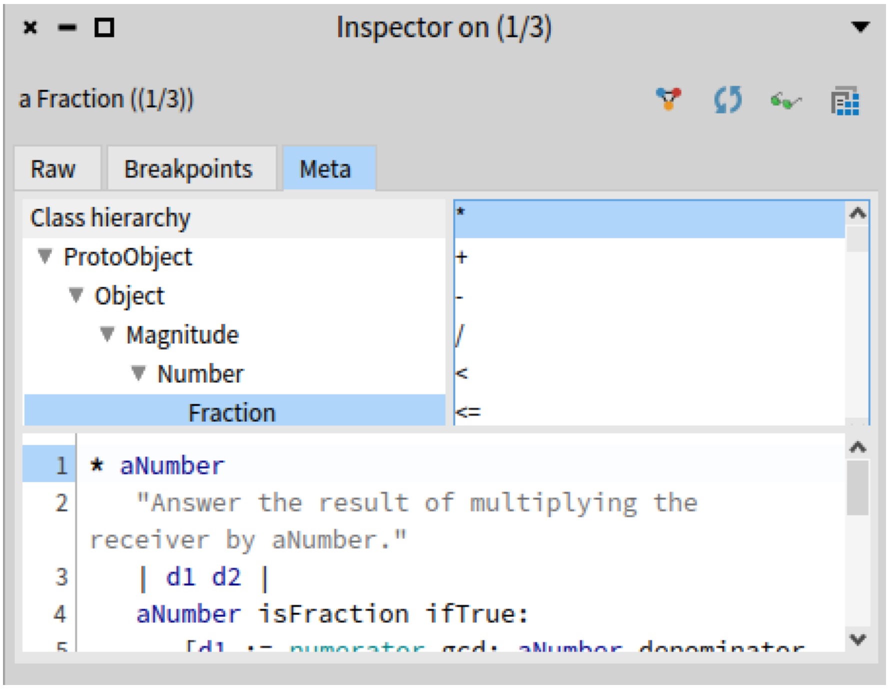
KDH: this section was/is missing.

The following animation shows how to put a breakpoint on writing an instance variable, the breakpoints listing on the current object, and how to deactivate one.

KDH: adding a GIF file does not work. We need an alternative or remove this part

16.4 The Meta tab: class hierarchy and searching methods

The `Meta` tab is the last one that is available for most objects. See Figure **??**. On the left, it shows the hierarchy of the current object's class. On the right, it shows the available methods. Clicking on parent classes in the hierarchy will show methods implemented in this class on the right. Selecting a method will display its source code at the bottom of the tab.

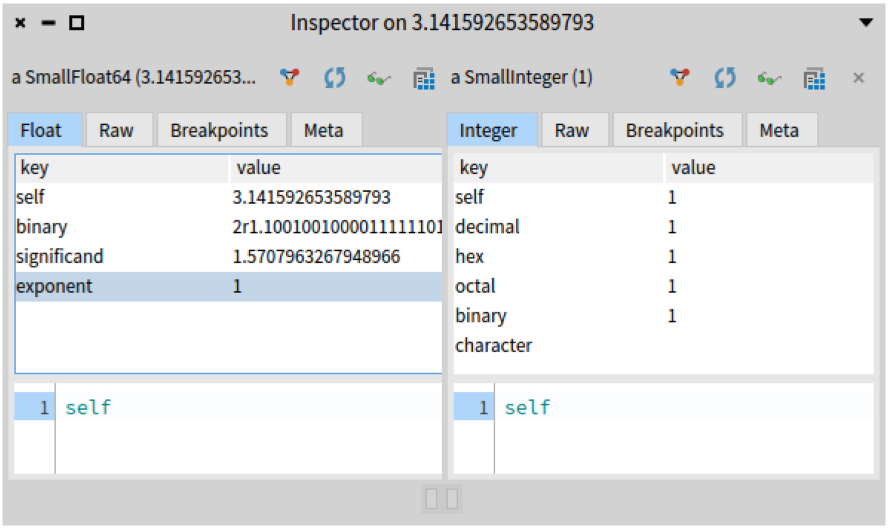


```
* aNumber
	"Answer the result of multiplying the receiver by aNumber."
	| d1 d2 |
	aNumber isFraction ifTrue:
```

Figure 16-4 Meta tab. % width=60&anchor=InspectorMetaTab

16.5 Creating custom tabs

If you used the inspector a bit, you may have noticed that some objects have additional tabs showing up in the inspector. For example, both `Floats` and `Integers` have their first tabs showing different representations of numbers, as shown in Figure **??**.



key	value
self	3.141592653589793
binary	2r1.1001001000011111101
significand	1.5707963267948966
exponent	1

```
self
```

key	value
self	1
decimal	1
hex	1
octal	1
binary	1
character	

```
self
```

Figure 16-5 Inspecting numbers. % width=60&anchor=InspectorForNumbers

Another example is the `FileReference` class. When a file reference is inspected, according to the type of the file, different tabs show up with relevant information.

Creating a new tab is as simple as reusing existing Spec presenters or defining new ones for your specific case. For example, you can define a tab displaying a specific Roassal visualization.

The following sections will explain how to add a few additional tabs to instances of `OrderedCollection`. This class already has a custom tab showing the list of its items which is defined by its superclass `Collection`.

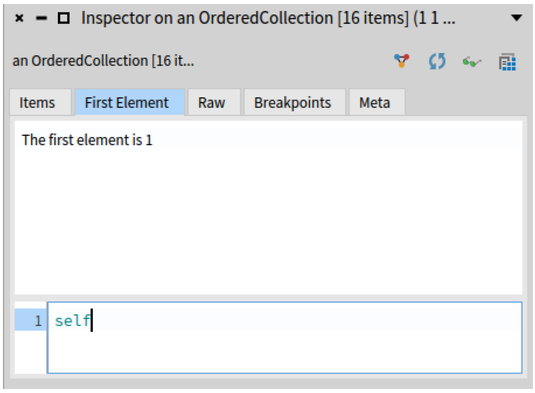
16.6 Adding a tab with text

Let's add a first tab containing a text describing the first element of the collection. Define the following method:

```
OrderedCollection << inspectionFirstElement

  <inspectorPresentationOrder: 1 title: 'First Element'>

  ^ SpTextPresenter new
    text: 'The first element is ', self first asString;
    beNotEditable;
    yourself
```

`<inspectorPresentationOrder: 1 title: 'First Element'>` is a pragma that is detected when creating an inspector on an object. When creating an inspector on an instance of `OrderedCollection`, this method will now be used to generate a tab. The title of the tab will be `First Element`, it will have position 1 in the order of tabs.

The content of the tab is returned by the method. Here we are creating a text presenter (`SpTextPresenter`) with the content we want and we specify that it should not be editable. This gives us the result shown in Figure 16-6.


Inspector on an OrderedCollection [16 items] (1 1 ...
an OrderedCollection [16 it...
Items
First Element
Raw
Breakpoints
Meta

The first element is 1

```
self
```

Figure 16-6 First element tab.

Notice that our new tab is in the second position. This is because in `Collection<<inspectionItems:` (the method defining the Items tab) the order parameter is 0.

16.7 Adding a tab with a table and conditions on when to display it

Let's create a new tab that will display a table if the collection contains only numbers. It will show each number and the result of multiplying that number with 2.

First let's create the tab with the table:

```
OrderedCollection << inspectionMultipliedByTwo

  <inspectorPresentationOrder: 10 title: 'Multiply by 2'>

  | itemColumn multipliedByTwoColumn |
  itemColumn := SpStringTableColumn
    title: 'Item'
    evaluated: #yourself.
  itemColumn width: 30.
  multipliedByTwoColumn := SpStringTableColumn
    title: 'Multiply by 2'
    evaluated: [ :each | each * 2 ].
  ^ SpTablePresenter new
      addColumn: itemColumn;
      addColumn: multipliedByTwoColumn;
      items: self;
      beResizable;
      yourself
```

When we inspect a collection of numbers we see the tabs shown in Figure **??**.

However if the collection contains elements that are not numbers, the tab crashes and looks like a red rectangle. By defining a method with the name `<name of the method defining the tab>Context:` we can specify when we want to activate a given tab. For example:

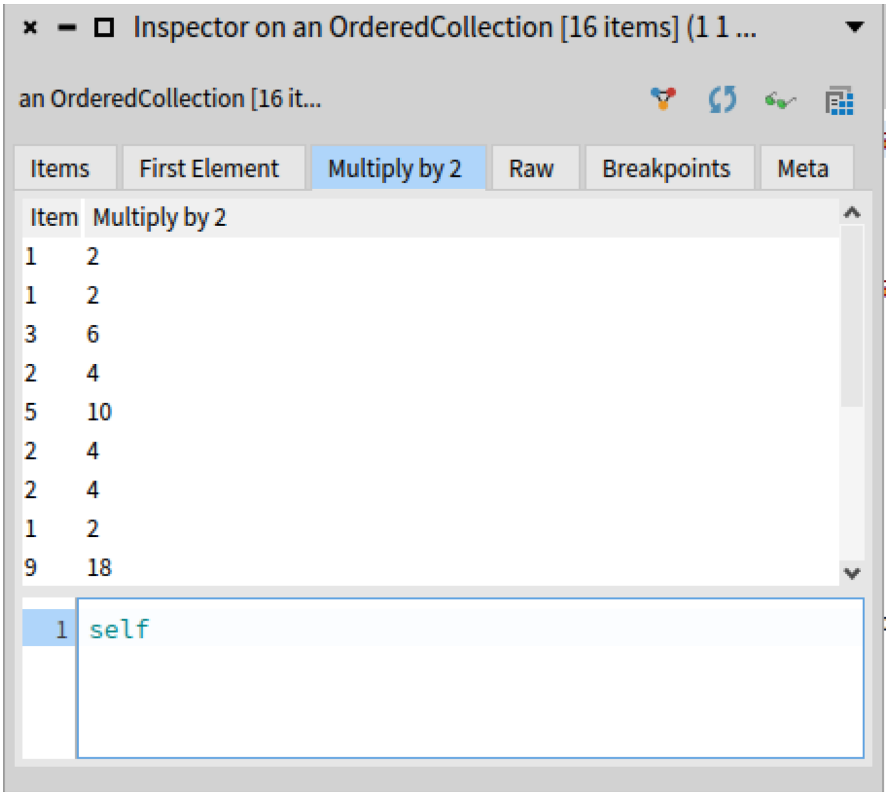
```
OrderedCollection << inspectionMultipliedByTwoContext: aContext

  ^ aContext active: self containsOnlyNumbers
```

```
OrderedCollection << containsOnlyNumbers

  ^ self allSatisfy: [ :each | each isNumber ]
```

These two methods will ensure that the tab will be displayed only when there are only numbers in the collection.

16.8 Adding a raw view of a specific element of the collection and removing the evaluator

We can also add a tab showing the raw view of the max value:


Inspector on an OrderedCollection [16 items] (1 1 ...
an OrderedCollection [16 it...
Items
First Element
Multiply by 2
Raw
Breakpoints
Meta

Item	Multiply by 2
1	2
1	2
3	6
2	4
5	10
2	4
2	4
1	2
9	18

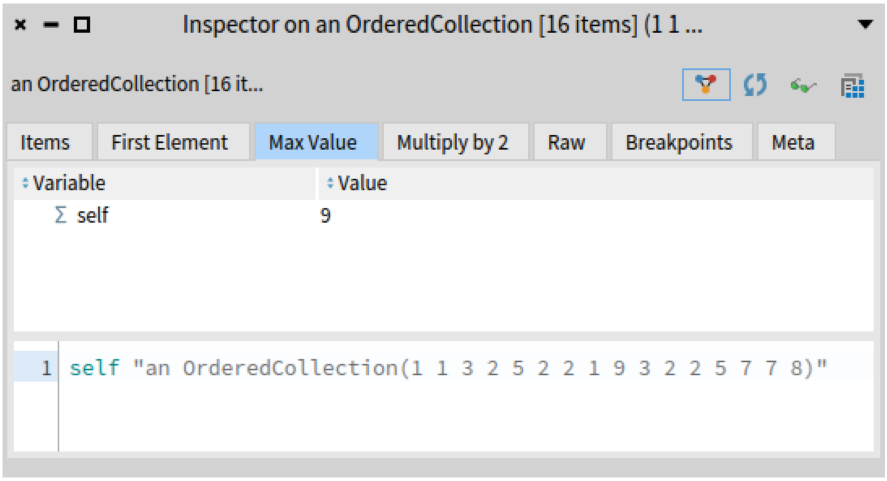
```
self
```

Figure 16-7 Multiplied by 2 tab. % width=60&anchor=InspectorMultipliedByTwoTab

```
OrderedCollection << inspectionMaxValue

  <inspectorPresentationOrder: 5 title: 'Max Value'>

  ^ StRawInspectionPresenter on: self max
```

```
OrderedCollection << inspectionMaxValueContext: aContext

  ^ aContext active: self containsOnlyIntegers
```

However as we can see in Figure **??**, the `self` in the evaluator does not match the `self` in the max value, which is confusing. So we will hide the evaluator.


Inspector on an OrderedCollection [16 items] (1 1 ...

an OrderedCollection [16 it...

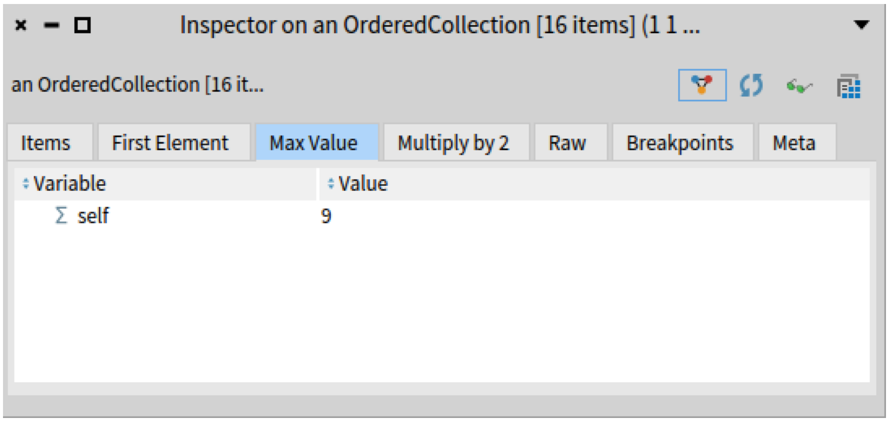
Variable	Value
Σ self	9

```
self "an OrderedCollection(1 1 3 2 5 2 2 1 9 3 2 2 5 7 7 8)"
```

Figure 16-8 Inspect max value tab. %width=60&anchor=InspectorMaxValueTab

```
OrderedCollection << inspectionMaxValueContext: aContext

  aContext withoutEvaluator.
  ^ aContext active: self containsOnlyIntegers
```

By reinspecting the same collection we see the inspector in Figure **??**.


Inspector on an OrderedCollection [16 items] (1 1 ...

an OrderedCollection [16 it...

Variable	Value
Σ self	9

Figure 16-9 Removing the evaluator. % width=60&anchor=InspectorWithoutEvaluator

16.9 Adding Roassal charts

As said above, Roassal allows one to build visualizations. The library includes some common graphs like a histogram. Let's add a histogram of the values if there are only numbers in the collection. Roassal 3 visualizations can be embedded in a presenter by sending the `asPresenter` message to an instance of `RSBuilder`. In the code below, `RSHistogramPlot` is a subclass of `RSBuilder`.

```
OrderedCollection << inspectionIntegerHistogram

  <inspectorPresentationOrder: -1 title: 'Histogram'>

  | plot |
  plot := RSHistogramPlot new x: self.
  ^ plot asPresenter
```

```
OrderedCollection << inspectionIntegerHistogramContext: aContext

  aContext active: self containsOnlyIntegers.
  aContext withoutEvaluator.
```

By inspecting `{ 1 . 1 . 3 . 2 . 5 . 2. 2 . 1. 9. 3 . 2. 2. 5 . 7 . 7 . 8 } asOrderedCollection` we see the inspector shown in Figure **??**.

16.10 Conclusion

In this chapter, we presented briefly the inspector and how you can specialize its tabs and evaluator to shape the way you can see and interact with your objects. We presented how to define conditional tabs, as well as embed visualizations.

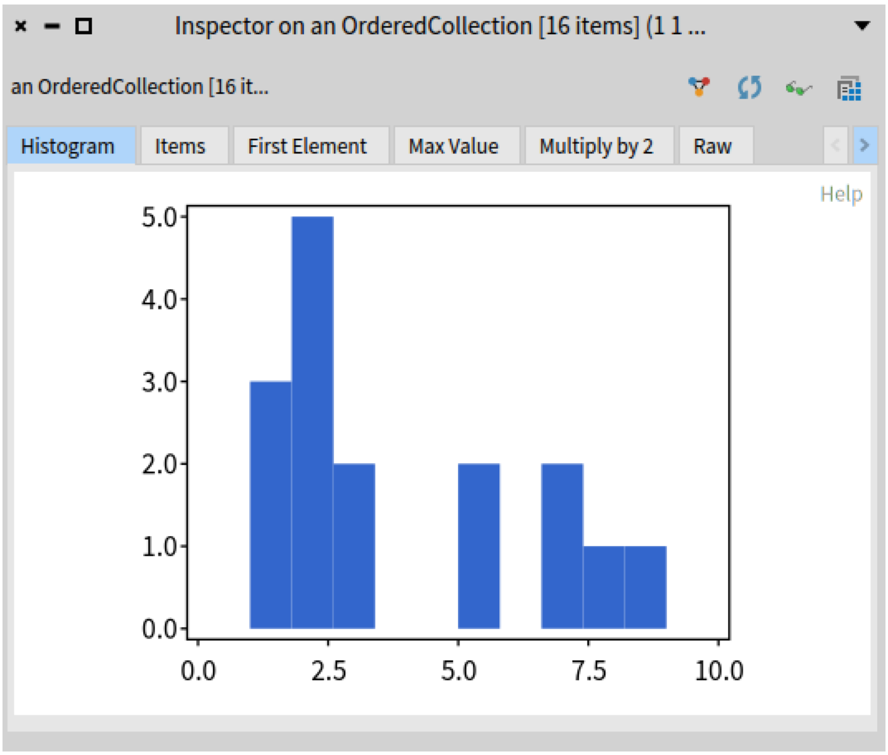


Figure 16-10 Histogram tab. %width=60&anchor=histogram

Part III

Working with Commands

CHAPTER 17

A simple contact book

In this chapter, we develop a simple model for a contact book. Then we define a user interface. This example will be used later in the book as an example to explain concepts such as commands, applications, and windows.

Now it is more of a replay of the concepts previously mentioned. We start by implementing classes modeling the domain and then we will add a basic graphical user interface to obtain a little application as shown in Figure **??**.

17.1 Contact book model

The model for the domain of our example is composed of two classes: Contact and ContactBook as shown in Figure **??**.

Contact

The class modeling a contact is defined as follows.

```
Object << #Contact
    slots: {#name . #phone};
    package: 'ContactBook'
```

It just defines a `printOn:` method and a couple of accessors (not shown in the text).

```
Contact >> printOn: aStream

        super printOn: aStream.
        aStream nextPut: $(.
```

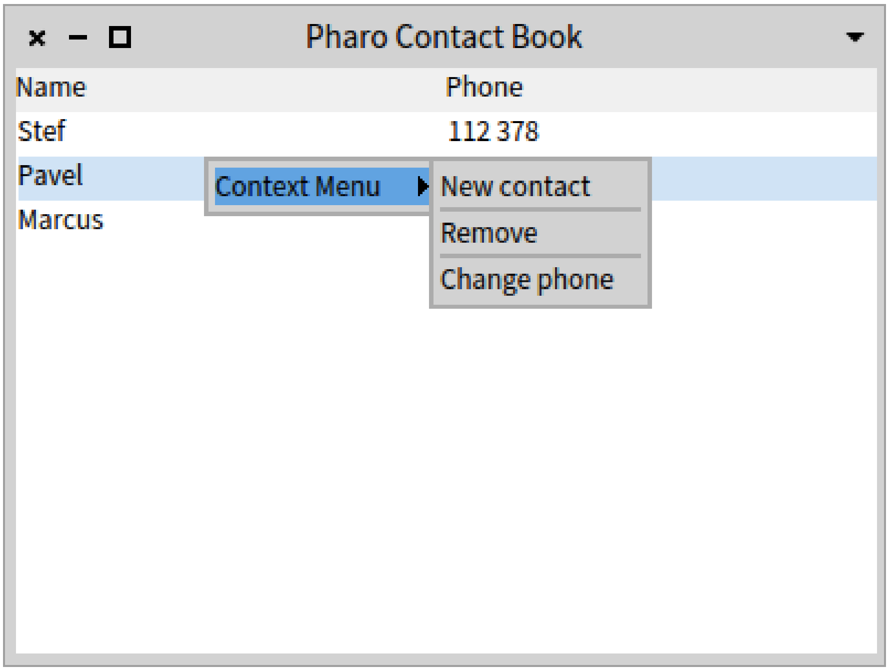


Figure 17-1 A rudimentary contact book application. % width=60&anchor=overview

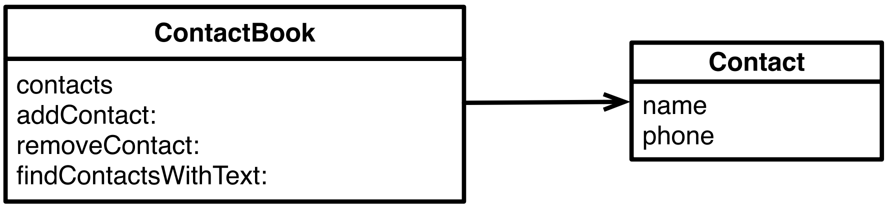


Figure 17-2 A simple model for the contact book. %width=60&anchor=contact-model

```
        aStream nextPutAll: name.
        aStream nextPut: $).
```

```
Contact >> hasMatchingText: aString

    ^ name includesSubstring: aString caseSensitive: false
```

```
Contact class >> name: aNameString phone: aPhoneString

    ^ self new
        name: aNameString;
        phone: aPhoneString;
        yourself
```

ContactBook

Now we define the class modeling the contact book. As for the contact class, it is simple and quite straightforward.

```
Object << #ContactBook
    slots: { #contacts };
    package: 'ContactBook'
```

```
ContactBook >> initialize

    super initialize.
    contacts := OrderedCollection new
```

We add the possibility to add and remove a contact

```
ContactBook >> addContact: aContact

    contacts add: aContact
```

```
ContactBook >> removeContact: aContact

    contacts remove: aContact
```

```
ContactBook >> addContact: newContact after: contactAfter

    contacts add: newContact after: contactAfter
```

We add a simple testing method in case one wants to write some tests (which we urge you to do).

```
ContactBook >> includesContact: aContact

    ^ contacts includes: aContact
```

And now we add a method to create a contact and add it to the contact book.

```
ContactBook >> add: contactName phone: phone

    | contact |
    contact := Contact new name: contactName; phone: phone.
    self addContact: contact.
    ^ contact
```

Finally, some facilities to query the contact book.

```
ContactBook >> findContactsWithText: aText

    ^ contacts select: [ :e | e hasMatchingText: aText ]
```

```
ContactBook >> size

    ^ contacts size
```

```
ContactBook >> contents

    ^ contacts
```

Pre-filling up the contact book

Since we want to have some contacts and we want to keep them without resorting to a database or file, we set some class instance variables.

We define a class instance variable `coworkers` and define a class method accessor as follows:

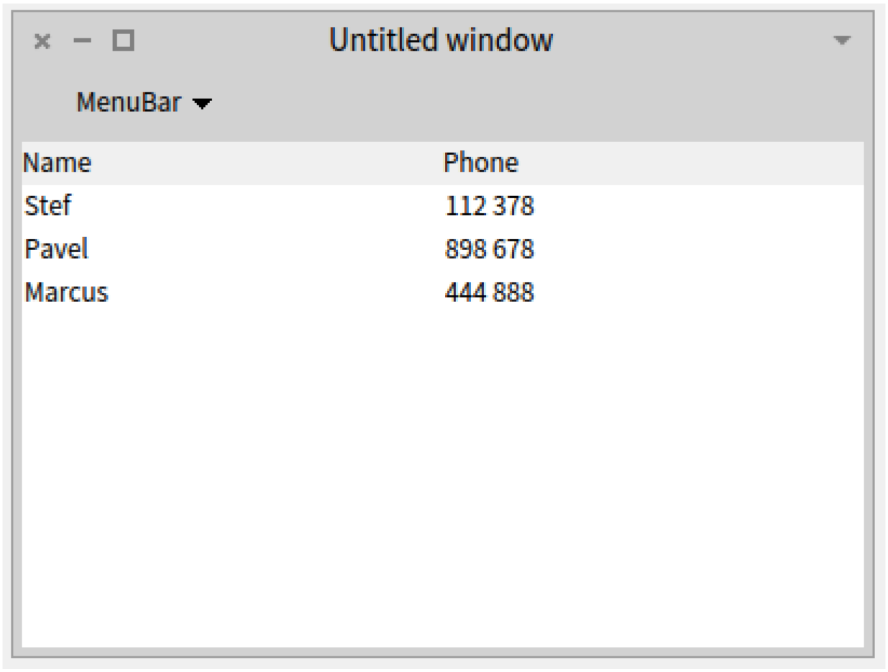
```
ContactBook class >> coworkers

    ^coworkers ifNil: [
        coworkers := self new
            add: 'Stef' phone: '112 378';
            add: 'Pavel' phone: '898 678';
            add: 'Marcus' phone: '444 888';
            yourself]
```

We add one method to be able to reset them if necessary. The `<script>` pragma tells the system browser to add a small button to execute `reset` method easily.

```
ContactBook class >> reset

    <script>
    coworkers := nil
```

17.2 A simple graphical user interface

Now we define the graphical user interface (GUI) to expose the model to the user. The targeted GUI is shown in Figure **??**.



Name	Phone
Stef	112 378
Pavel	898 678
Marcus	444 888

Figure 17-3 A rudimentary contact book application. %width=60&anchor=firstFullUI

We define the class `ContactBookPresenter`. It holds a reference to a contact book and it is structured around a table.

```
SpPresenter << #ContactBookPresenter
    slots: { #table . #contactBook};
    package: 'ContactBook'
```

We define an accessor for the contact book and the table.

```
ContactBookPresenter >> contactBook

    ^ contactBook
```

```
ContactBookPresenter >> table: anObject

    table := anObject
```

```
ContactBookPresenter >> table

    ^ table
```

Initializing the model

We specialize the method `setModelBeforeInitialization:` that is invoked by the framework to assign the `contactBook` instance variable to the object passed during the execution of the expression `(ContactBookPresenter on: ContactBook coworkers) open`.

```
ContactBookPresenter >> setModelBeforeInitialization: aContactBook

    super setModelBeforeInitialization: aContactBook.
    contactBook := aContactBook
```

Layout

```
ContactBookPresenter >> defaultLayout

    ^ SpBoxLayout newVertical add: #table; yourself
```

Widget initialization

We initialize the table to display two columns for the name and the phone. The respective accessor messages will be sent to the elements to fill up the columns. Finally, the table content is set using the contact book contents.

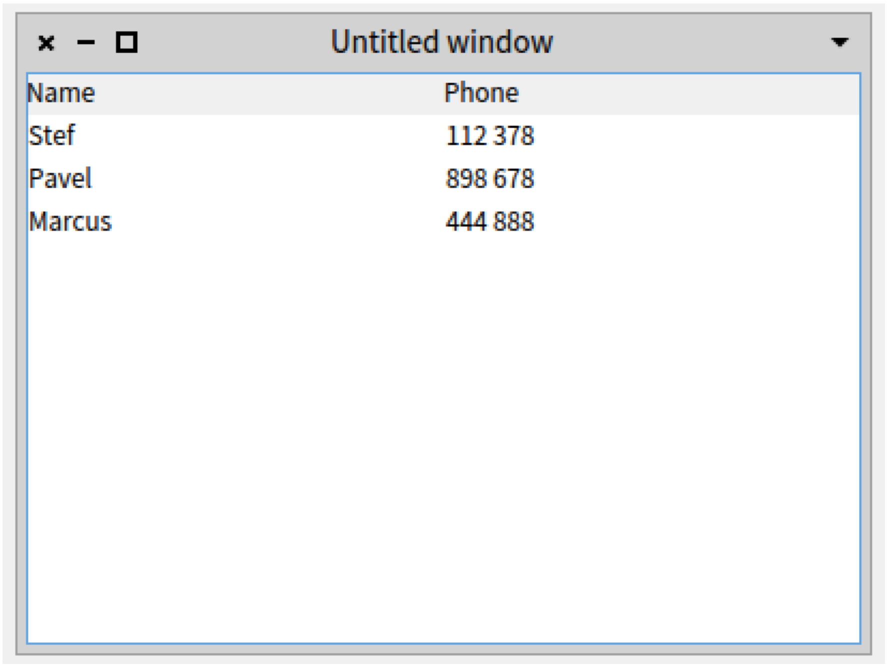
```
ContactBookPresenter >> initializePresenters

    table := self newTable.
    table
        addColumn: (StringTableColumn title: 'Name' evaluated: #name);
        addColumn: (StringTableColumn title: 'Phone' evaluated:
    #phone).
    table items: contactBook contents.
```

Now we can open the UI by executing the snippet `(ContactBookPresenter on: ContactBook coworkers) open`.

We define a class method to be able to easily re-execute the setup.

```
ContactBookPresenter class >> coworkersExample

    <example>
    ^ (self on: ContactBook coworkers) open
```

You should obtain the GUI as shown in Figure **??**.



Name	Phone
Stef	112 378
Pavel	898 678
Marcus	444 888

Figure 17-4 First version of the GUI without menus and toolbar. % width=60&anchor=firstMenuToolbar

Interacting with user

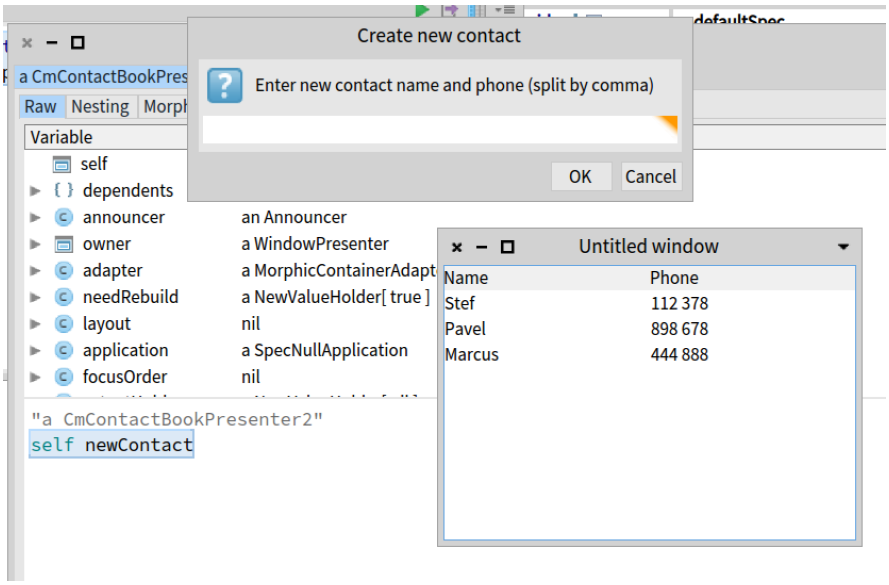
We now implement the method that will open a window to ask the user to create a new contact for the contact book.

```
ContactBookPresenter >> newContact
    | rawData split |
    rawData := self
        request: 'Enter new contact name and phone (split by comma)'
        initialAnswer: ''
        title: 'Create new contact'.
    split := rawData splitOn: $,.
    (split size = 2 and: [ split allSatisfy: [ :each | each isNotEmpty
    ]])
        ifFalse: [ SpInvalidUserInput signal: 'Please enter contact
    name and phone (split by comma)'  ].

    ^ Contact new
        name: split first;
        phone: split second;
        yourself
```

To test it, we can get access to the presenter with

```
(ContactBookPresenter on: ContactBook coworkers)
    open presenter inspect
```

and you can send the `newContact` message to open the GUI shown in Figure ??.



Variable	
self	
dependents	
announcer	an Announcer
owner	a WindowPresenter
adapter	a MorphicContainerAdapt
needRebuild	a NewValueHolder[true]
layout	nil
application	a SpecNullApplication
focusOrder	nil

```
"a CmContactBookPresenter2"
self newContact
```

Untitled window

Name	Phone
Stef	112 378
Pavel	898 678
Marcus	444 888

Figure 17-5 Playing inside the inspector. % width=80&anchor=inspector

Some extra methods

We will also define the methods `isContactSelected` and `selectedContact` to know if a contact is currently selected and to return it. It will help us later to add a contact just after the currently selected contact.

```
ContactBookPresenter >> isContactSelected

    ^ table selectedItems isNotEmpty
```

```
ContactBookPresenter >> selectedContact

    ^ table selection selectedItem
```

17.3 Conclusion

Now we have a little contact book manager that we can use to explain other topics.

CHAPTER 18

Commander: A powerful and simple command framework

Commander was a library originally developed by Denis Kudriashov. Commander 2.0 is the second iteration of that library. It was designed and developed by Julien Delplanque and Stéphane Ducasse. Note that Commander 2.0 is not compatible with Commander but it is really easy to migrate from Commander to Commander 2.0. We describe Commander 2.0 in the context of Spec. From now on, when we mention Commander we refer to Commander 2.0. In addition, we show how to extend Commander to other needs.

18.1 Commands

Commander models application actions as first-class objects following the Command design pattern. With Commander, you can express commands and use them to generate menus and toolbars, but also to script applications from the command line.

Every action is implemented as a separate command class (subclass of `CmCommand`) with an `execute` method and the state required for execution.

We will show later that for a UI framework, we need more information such as an icon and shortcut description. In addition, we will present how commands can be decorated with extra functionality in an extensible way.

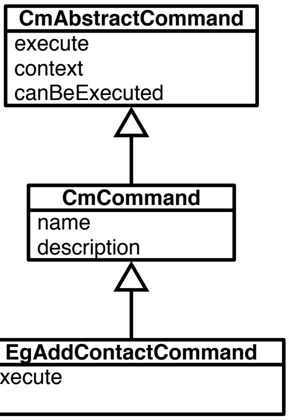


Figure 18-1 A simple command and its hierarchy.

18.2 Defining commands

A command is a simple object instance of a subclass of the class `CmCommand`. It has a description, a name (this name can be either static or dynamic as we will show later on). In addition, it has a context from which it extracts information to execute itself. In its basic form, there is no more than that.

Let us have a look at examples. We will define some commands for the ContactBook application and illustrate how they can be turned into menus and a menubar.

18.3 Adding some convenience methods

For convenience reasons, we define a common superclass of all the commands of the contact book application named `ContactBookCommand`.

```
CmCommand << #ContactBookCommand
    package: 'ContactBook'
```

We define a simple helper method to make the code more readable

```
ContactBookCommand >> contactBookPresenter

    ^ self context
```

For the same reason, we define another helper to access the contact book and the selected item.

```
ContactBookCommand >> contactBook

    ^ self contactBookPresenter contactBook
```

```
ContactBookCommand >> selectedContact

    ^ self contactBookPresenter selectedContact
```

Using the helper method `isContactSelected` we defined in the previous chapter, the method `hasSelectedContact` can be implemented as:

```
ContactBookCommand >> hasSelectedContact

    ^ self contactBookPresenter isContactSelected
```

Adding the Add Contact command

We define a subclass to define the add a contact command.

```
ContactBookCommand << #AddContactCommand
    package: 'ContactBook'
```

```
AddContactCommand >> initialize
    super initialize.
    self
        basicName: 'New contact';
        basicDescription: 'Creates a new contact and adds it to the
    contact book.'
```

```
AddContactCommand >> execute

    | contact |
    contact := self contactBookPresenter newContact.
    self hasSelectedContact
        ifTrue: [ self contactBook addContact: contact after: self
    selectedContact ]
        ifFalse: [ self contactBook addContact: contact ].
    self contactBookPresenter updateView
```

We define the method `updateView` to refresh the contents of the table.

```
ContactBookPresenter >> updateView
    table items: contactBook contacts
```

Now in an inspector on an instance of `ContactBookPresenter`, we can simply execute the command as follows:

```
(AddContactCommand new context: self) execute
```

Executing the command should ask you to give a name and a phone number and the new contact will be added to the list.

We can also execute the following snippet.

```
| presenter command |
presenter := ContactBookPresenter on: ContactBook coworkers.
command := AddContactCommand new context: presenter.
command execute
```

18.4 Adding the Remove Contact command

Now we define now another command to remove a contact. This example is interesting because it does not involve any UI interaction. It shows that a command is not necessarily linked to UI interaction.

```
ContactBookCommand << #RemoveContactCommand
    package: 'ContactBook'
```

```
RemoveContactCommand >> initialize
    super initialize.
    self
        name: 'Remove';
        description: 'Removes the selected contact from the contact
     book.'
```

This command definition illustrates how we can control when a command should or should not be executed. The method canBeExecuted allows specifying such a condition.

```
RemoveContactCommand >> canBeExecuted
    ^ self context isContactSelected
```

The method execute is straightforward.

```
RemoveContactCommand >> execute
    self contactBook removeContact: self selectedContact.
    self contactBookPresenter updateView
```

The following test validates the correct execution of the command.

```
ContactCommandTest >> testRemoveContact

    self assert: presenter contactBook size equals: 3.
    presenter table selectIndex: 1.
    (RemoveContactCommand new context: presenter) execute.
    self assert: presenter contactBook size equals: 2
```

18.5 Turning commands into menu items

Now that we have our commands, we would like to reuse them and turn them into menus. In Spec, commands that are transformed into menu items are structured into a tree of command instances. The class method `buildCommandsGroupWith:forRoot:` of `SpPresenter` is a hook to let presenters define the root of the command instance tree.

A command is transformed into a command for Spec using the message `forSpec`.We will show later that we can add UI-specific information to a command such as an icon and a shortcut.

The method `buildCommandsGroupWith:forRoot:` registers commands to which the presenter instance is passed as context. Note that here we just add plain commands, but we can also create groups. Later in this chapter we will also specify a menu bar in this method.

```
ContactBookPresenter class >>
    buildCommandsGroupWith: presenter
    forRoot: rootCommandGroup

    rootCommandGroup
        register: (AddContactCommand forSpec context: presenter);
        register: (RemoveContactCommand forSpec context: presenter)
```

Now we have to attach the root of the command tree to the table. This is what we do with the new line in the `initializePresenters` method. Notice that we have full control and as we will show we could select a subpart of the tree (using the message `/`) and define it as root for a given component.

```
ContactBookPresenter >> initializePresenters
    table := self newTable.
    table
        addColumn: (SpStringTableColumn title: 'Name' evaluated:
    #name);
        addColumn: (SpStringTableColumn title: 'Phone' evaluated:
    #phone).
    table contextMenu: [ self rootCommandsGroup beRoot asMenuPresenter
    ].
    table items: contactBook contacts.
```

When reopening the interface with `(ContactBookPresenter on: ContactBook coworkers) open`, you should see the menu items as shown in Figure 18-2. As we will show later, we could even replace a menu item with another one, changing its name, or icon in place.

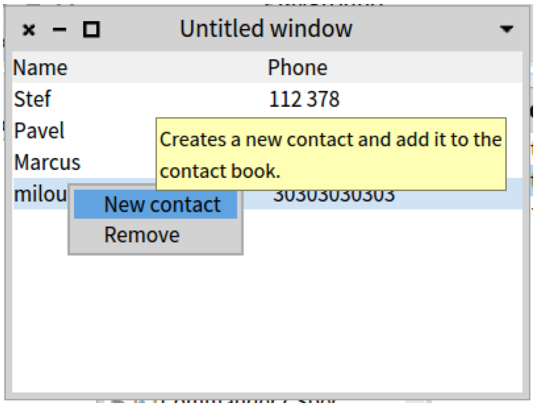


Figure 18-2 With two menu items with groups.

18.6 Introducing groups

Commands can be managed in groups and such groups can be turned into corresponding menu item sections. The key hook method is the class method named `buildCommandsGroupWith: presenterInstance forRoot:`.

Here we give an example of such grouping. Note that the message `asSpecGroup` is sent to a group. We create two methods, each creating a simple group, one for adding, and one for removing contracts.

```
ContactBookPresenter class >> buildAddingGroupWith: presenter

    ^ (CmCommandGroup named: 'Adding') asSpecGroup
        description: 'Commands related to contact addition.';
        register: (AddContactCommand forSpec context: presenter);
        beDisplayedAsGroup;
        yourself
```

```
ContactBookPresenter class >> buildRemovingGroupWith: presenter

    ^ (CmCommandGroup named: 'Removing') asSpecGroup
        description: 'Commands related to contact removal.';
        register: (RemoveContactCommand forSpec context: presenter);
        beDisplayedAsGroup;
        yourself
```

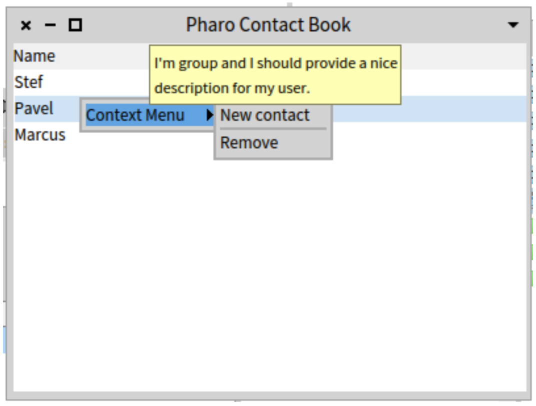
We group the previously defined groups together under the contextual menu:

```
ContactBookPresenter class >> buildContextualMenuGroupWith: presenter

    ^ (CmCommandGroup named: 'Context Menu') asSpecGroup
        register: (self buildAddingGroupWith: presenter);
        register: (self buildRemovingGroupWith: presenter);
        yourself
```

Finally, we revisit the hook `buildCommandsGroupWith:forRoot:` to register the last group to the root command group.

```
ContactBookPresenter class >>
    buildCommandsGroupWith: presenter
    forRoot: rootCommandGroup

    rootCommandGroup
        register: (self buildContextualMenuGroupWith: presenter)
```

When reopening the interface with `(ContactBookPresenter on: ContactBook coworkers) open`, you should see the menu items inside a `'Context Menu'` as shown in Figure 18-3.



I'm group and I should provide a nice description for my user.

Figure 18-3 With a context menu.

To show that we can also select part of the command tree, we select the `'Context Menu'` group and declare it as the root of the table menu. Then you will not see the `'Context Menu'` anymore.

```
ContactBookPresenter >> initializePresenters

    table := self newTable.
    table
        addColumn: (SpStringTableColumn title: 'Name' evaluated:
    #name);
```

```
        addColumn: (SpStringTableColumn title: 'Phone' evaluated:
    #phone).
    table contextMenu: [ (self rootCommandsGroup / 'Context Menu')
    beRoot asMenuPresenter ].
    table items: contactBook contacts
```

Here we see that by sending the slash message (`/`), we can select the group in which we want to add a menu iten.

18.7 Extending menus

Building menus is nice, but sometimes we need to add a menu to an existing one. Commander supports this via a dedicated pragma, called `<extensionCommands>` that identifies extensions.

Imagine that we have new functionality that we want to add to the contact book and that this behavior is packaged in another package, here, `ContactBook-Extensions`. First, we will define a new command and second, we will show how we can extend the existing menu to add a new menu item.

```
ContactBookCommand << #ChangePhoneCommand
    package: 'ContactBook-Extensions'
```

```
ChangePhoneCommand >> initialize

    super initialize.
    self
        name: 'Change phone';
        description: 'Change the phone number of the contact.'
```

```
ChangePhoneCommand >> execute

    self selectedContact phone: self contactBookPresenter newPhone.
    self contactBookPresenter updateView
```

We extend `ContactBookPresenter` with the method `newPhone` to let the presenter decide how a user should provide a new phone number.

```
ContactBookPresenter >> newPhone

    | phone |
    phone := self
        request: 'New phone for the contact'
        initialAnswer: self selectedContact phone
        title: 'Set new phone for contact'.
    (phone matchesRegex: '\d\d\d\s\d\d\d')
        ifFalse: [
            SpInvalidUserInput signal: 'The phone number is not well
    formatted.
Should match "\d\d\d\s\d\d\d"' ].
    ^ phone
```

The last missing piece is the declaration of the extension. This is done using the pragma `<extensionCommands>` on the class side of the presenter class as follows:

```
ContactBookPresenter class >>
    changePhoneCommandWith: presenter
    forRootGroup: aRootCommandsGroup

    <extensionCommands>

    (aRootCommandsGroup / 'Context Menu')
        register: (ChangePhoneCommand forSpec context: presenter)
```

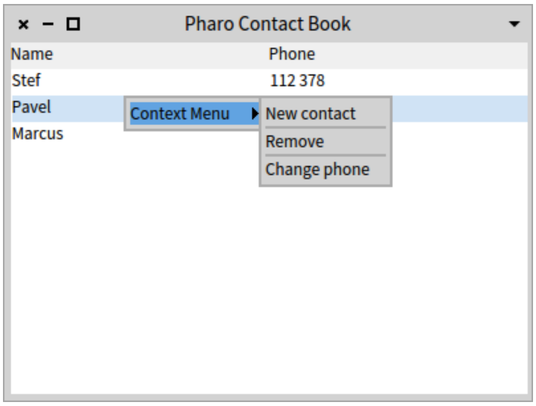


Figure 18-4 With menu extension.

18.8 Managing icons and shortcuts

By default a command does not know about Spec-specific behavior, because a command does not have to be linked to UI. Obviously you want to have icons and shortcut bindings when you are designing an interactive application.

Commander supports the addition of icons and shortcut keys to commands.Let us see how it works from a user perspective. The framework offers two methods to set an icon and a shortcut key: `iconName:` and `shortcutKey:`. We should specialize the method `asSpecCommand` as follows:

```
RemoveContactCommand >> asSpecCommand

    ^ super asSpecCommand
        iconName: #removeIcon;
        shortcutKey: $x meta;
        yourself
```

```
AddContactCommand >> asSpecCommand

    ^ super asSpecCommand
        shortcutKey: $n meta;
        iconName: #changeAdd;
        yourself
```

Note that the commands are created using the message `forSpec`. This message takes care of the calling `asSpecCommand`.

18.9 Enabling shortcuts

At the time of writing this chapter, Commander management of shortcuts has not been pushed to Spec to avoid dependency on Commander. It is then the responsibility of your presenter to manage shortcuts as shown in the following method. We ask the command group to install the shortcut handler in the window.

```
ContactBookPresenter >> initializeWindow: aWindowPresenter

    super initializeWindow: aWindowPresenter.
    self rootCommandsGroup installShortcutsIn: aWindowPresenter
```

18.10 In-place customisation

Commander supports the reuse and in-place customisation of commands. It means that a command can be modified on the spot: for example, its name or description can be adapted to the exact usage context. Here is an example that shows that we adapt the same command twice.

Let us define a really simple and generic command that will simply inspect the object.

```
ContactBookCommand << #InspectCommand
    package: 'ContactBook-Extensions'
```

```
InspectCommand >> initialize

    super initialize.
    self
```

```
        name: 'Inspect';
        description: 'Inspect the context of this command.'
```

```
InspectCommand >> execute

    self context inspect
```

By using a block, the context is computed at the moment the command is executed and the name and description can be adapted for its specific usage as shown in Figure 18-5.

```
ContactBookPresenter class >>
    extraCommandsWith: presenter
    forRootGroup: aRootCommandsGroup

    <extensionCommands>

    aRootCommandsGroup / 'Context Menu'
        register:
            ((CmCommandGroup named: 'Extra') asSpecGroup
                description: 'Extra commands to help during
    development.';
        register:
            ((InspectCommand forSpec context: [ presenter
    selectedContact ])
                name: 'Inspect contact';
                description: 'Open an inspector on the selected
    contact.';
                iconName: #smallFind;
                yourself);
        register:
            ((InspectCommand forSpec context: [ presenter contactBook
    ])
                name: 'Inspect contact book';
                description: 'Open an inspector on the contact book.';
                yourself);
        yourself)
```

18.11 Managing a menu bar

Commander also supports menu bar creation. The logic is the same as for contextual menus: we define a group and register it under a given root, and we tell the presenter to use this group as a menubar.

Imagine that we have a new command to print the contact.

```
ContactBookCommand << #PrintContactCommand
    package: 'ContactBook'
```

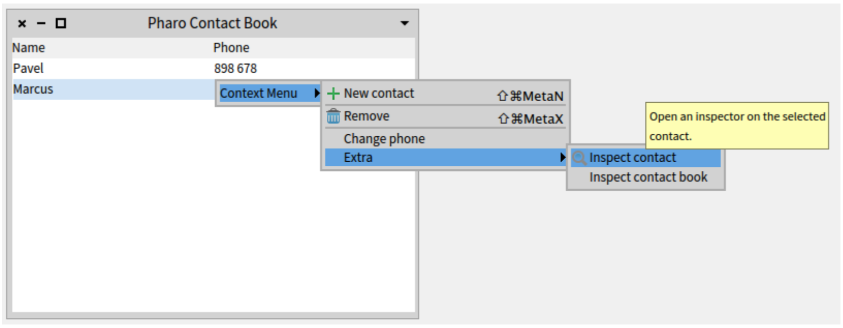


Figure 18-5 With menu extension.

```
PrintContactCommand >> initialize

    super initialize.
    self
        name: 'Print';
        description: 'Print the contact book in Transcript.'
```

```
PrintContactCommand >> execute

    Transcript open.
    self contactBook contacts do: [ :contact | self traceCr: contact
     name , ' - ' , contact name ]
```

We create a simple group that we call 'MenuBar' (but it could be called anything).

```
ContactBookPresenter class >> buildMenuBarGroupWith: presenter

    ^ (CmCommandGroup named: 'MenuBar') asSpecGroup
        register: (PrintContactCommand forSpec context: presenter);
        yourself
```

We modify the root to add the menu bar group in addition to the previou one.

```
ContactBookPresenter class >>
    buildCommandsGroupWith: presenter
    forRoot: rootCommandGroup

    rootCommandGroup
        register: (self buildMenuBarGroupWith: presenter);
        register: (self buildContextualMenuGroupWith: presenter)
```

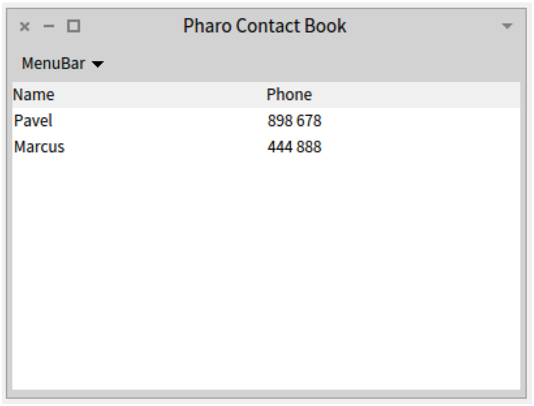
We hook it into the widget as the last line of the `initializePresenters` method.

Notice the use of the message `asMenuBarPresenter` and the addition of a new instance variable called `menuBar`.

```
ContactBookPresenter >> initializePresenters

    table := self newTable.
    table
        addColumn: (SpStringTableColumn title: 'Name' evaluated:
    #name);
        addColumn: (SpStringTableColumn title: 'Phone' evaluated:
    #phone).
    table contextMenu: [ (self rootCommandsGroup / 'Context Menu')
    beRoot asMenuPresenter ].
    table items: contactBook contents.
    menuBar := (self rootCommandsGroup / 'MenuBar') asMenuBarPresenter.
```

Finally, to get the menu bar we declare it in the layout. We use `SpAbstractPresenter class>>#toolbarHeight` to specify the height of the menu bar.

```
ContactBookPresenter >> defaultLayout

    ^ SpBoxLayout newVertical
            add: #menuBar
            withConstraints: [ :constraints | constraints height: self
    toolbarHeight ];
            add: #table;
            yourself
```



Name	Phone
Pavel	898 678
Marcus	444 888

Figure 18-6 With a menubar.

18.12 Conclusion

In this chapter, we saw how you can define a simple command and execute it in a given context. We show how you can turn a command into a menu item in Spec by sending the message `forSpec`. You learned how we can reuse and customize commands. We presented groups of commands as a way to structure menus and menu bars.