CHAPTER 1

# First Contact With Examples

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

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

## 1.1 A customer satisfaction UI

In this first example of a Spec UI we will construct a simple customer satisfaction UI, which will allow a user to give feedback about a service by clicking on one of three buttons. (This feedback should be recorded and processed, but this is outside of the scope of this text). We show a screenshot of the UI in Figure 1.1.

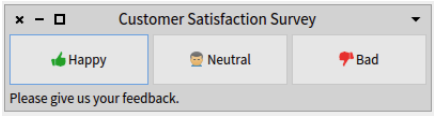


**Figure 1.1** A screen shot of the customer satisfaction UI.

### Create the class of the UI and variable accessors

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

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

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

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

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

### Instantiate and configure subwidgets

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

First we show widget instantiation:

```
CustomerSatisfaction >> initializeWidgets

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

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

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

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

Second, we configure the buttons of our UI. The message `label:` defines their label and the message `icon:` specifies the icon that will be displayed near the label.

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

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

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

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

```
CustomerSatisfaction >> initializePresenter

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

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

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

The widgets have now been defined and configured, but their placement in the UI has not yet been specified. This is the role of the class side method `defaultSpec`.

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

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

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

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

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

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

CustomerSatisfaction >> extent
  ^ 400@100
```

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

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

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

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

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

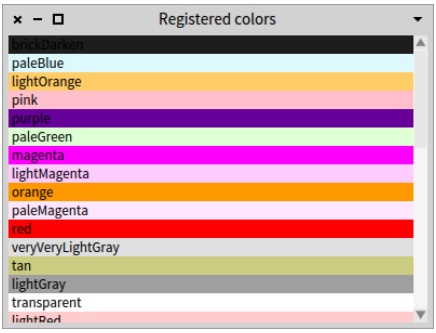


**Figure 1.2** Screen shot of the list with modified background colors

## 1.2 Fun with Lists

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

These examples show two important features of Spec:

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

### Registered colors as item background

We start with an example of a `ListModel` where the elements have different background colors, shown in Figure 1.2. The items held in the list are the names of different colors, and the list shows them using a background of that color.

The following code shows how this is done:

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

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

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

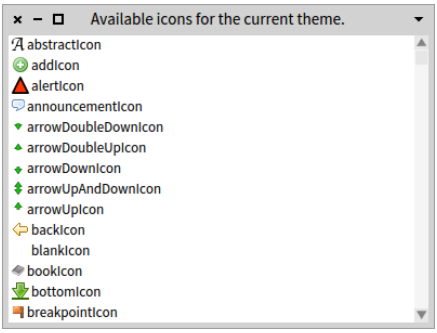


**Figure 1.3** Screen shot of the list of Icons

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

## List of icons

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

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

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

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

## 1.3 Conclusion

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

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

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