BahmanM.com
Carving Deep Into IT, Information Systems and ERP

Spec - Part I: The Basics

GUI Development in Pharo Smalltalk using Spec. It covers the basics concepts and walks you through the building up of a Spec version of the legendary "Hello, world".

Contents

Introduction

Spec is a UI library for Pharo Smalltalk. Well, to be more precise, Spec is a library for describing UI elements, their properties and their behaviour. Under the hood the output of Spec is fed into a UI framework like Polymorph to draw the widgets on the screen.

In this series of tutorial episodes, you will understand its concepts and learn how to use it for your everyday UI programming requirements.

Prerequisites

For this series we will use Pharo 2.0 (the latest stable version at the time of writing this) and since Spec is already included in Pharo 2.0, you don't need to do anything. Just create a package to put in all the classes that we create. Package name `My-Spec-Tutorial` will be used in this tutorial.

The First Window

Windows are the primary containers for other widgets in Spec. Creating and showing a window is simple.

Create a new class named `MyFirstWindow` with the following definition:

```
ComposableModel subclass: #MyFirstWindow
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'My-Spec-Tutorial'
```

If you are coming from a .NET, Swing or SWT background, you may notice that instead of just creating a new instance of the framwork's Window object (e.g. `JFrame`), `ComposableModel` has been sub-classed. As Spec is, obviously, about UI specification, naturally the next step is to provide the specifications!

Override the *class-side* method `defaultSpec` which basically tells Spec about UI specifications:

```
1 defaultSpec
2         ^ SpecLayout composed.
```

For now look at line 2 as a dummy layout place holder.

The last step is to override the instance-side method `initializeWidgets` which later we will use to initialize the widgets in our window. But for now the method doesn't do anything:

```
initializeWidgets
        "Empty method"
```

That's done it. Now in a workspace run `MyFirstWindow new openWithSpec` and voila!

The Legend

Almost all computer programming books and tutorial start with the legendary *"Hello, world"* example; who I am to break this holy ritual!? Let's make our dull window greet the universe. All that is needed is adding a label widget to our container.

First, the label. Spec interacts with widgets using instance variables and their accessors. So change the class defition and add a variable named `labelGreeting` to it.

```
1 ComposableModel subclass: #MyFirstWindow
2         instanceVariableNames: 'labelGreeting'
3         classVariableNames: ''
4         poolDictionaries: ''
5         category: 'My-Spec-Tutorial'
```

Note line 2. Now generate the accessor for `labelGreeting`. Remember, you don't need to write the accessors yourself; just right-click on the class and from the menu select *Refactoring → Class Refactoring → Generate Accessors*.

Time to initialise the label with the message of legends. As you may have already guessed, `initializeWidgets` needs to be modified:

```
1 initializeWidgets
2         self instantiateModels: #(
3                 labelGreeting    LabelModel
4         ).
5         labelGreeting text: 'Hello, world'.
```

On lines 2, 3 and 4, Spec is told to instantiate the label and attach a `LabelModel` to it. On line 5, label value is set.

And finally, `labelGreeting` should be added to our window. Like many other UI frameworks (e.g. Swing), widgets are not directly added to the container. Rather they are added to a layout which is in turn attached to the container. Same philosophy here. Editing the layout is done in the class-side method `defaultSpec`, recall?

```
defaultSpec
        ^ SpecColumnLayout new
                        add: #labelGreeting;
                        yourself.
```

MyFirstWindow layout has now changed to SpecColumnLayout which, as the name suggests, puts widgets in several rows of one column.

Now run `MyFirstWindow new openWithSpec` and behold the dawn of programming!

I Salute You!

As magnificent as your window is, it will probably get boring after a few runs to see the same message. Let's refactor our program so that it accepts a name from the user and greets that name.

Widgets

The greeter needs 3 widgets: a label (which is already there), a text input field and a button. As you can recall, the first step is to define instance variables for each widget:

```
ComposableModel subclass: #MyFirstWindow
        instanceVariableNames: 'labelGreeting textName buttonGreet
        classVariableNames: ''
        poolDictionaries: ''
        category: 'My-Spec-Tutorial'
```

Don't forget to **generate accessors** for the variables.

The next step is to modify the spec to match our simplistic design:

```
defaultSpec
        ^ SpecColumnLayout new
                        add: #labelGreeting;
                        add: #textName;
                        add: #buttonGreet;
```

```
                              yourself.
```

And finally widgets initialization:

```
1 initializeWidgets
2         self instantiateModels: #(
3                 labelGreeting   LabelModel
4                 textName        TextInputFieldModel
5                 buttonGreet     ButtonModel
6         ).
7         labelGreeting text: ''.
8         textName autoAccept: true.
9         buttonGreet label: 'Greet Me!'; disable.
```

On lines 4 and 5, the text field and the button with their respective models are added. On line 7 label's default text is set to empty. On line 8, Spec is told that upon every keystroke the text in the text field should be acceptd (in contrast to pressing `CTRL+S`). And on line 9, the label for button is set and the button is disabled as it will become enabled only when there is at least one character in the text field.

If you run the program, you can see how Spec has added widgets to the display and how their initial values and properties are set.

Let's See Some Action!

Right now, the greeter doesn't do anything; it just shows up. We need to tell spec when to do what: when to enable the button and when and with what to update the label. Wiring up actions and action/event handlers is done with overriding `initializePresenter` method.

```
1 initializePresenter
2         textName whenTextChanged: [
3                 buttonGreet enable ].
4         buttonGreet action: [
5                 labelGreeting text: 'Hello, ', textName text, '!'.
6                 buttonGreet disable ].
```

You may spot two action handlers here which are block closures. Very much like Swing and its anonymous classes but more readable and concise. On line 2 and 3, Spec is told to enable `buttonGreet` when the text in `textNam` changes. On line 4, 5 and 6, it is told to change the value of `labelGreeting` upon activating `buttonGreet` and then disable `buttonGreet` so that nobody gets a double greeting!

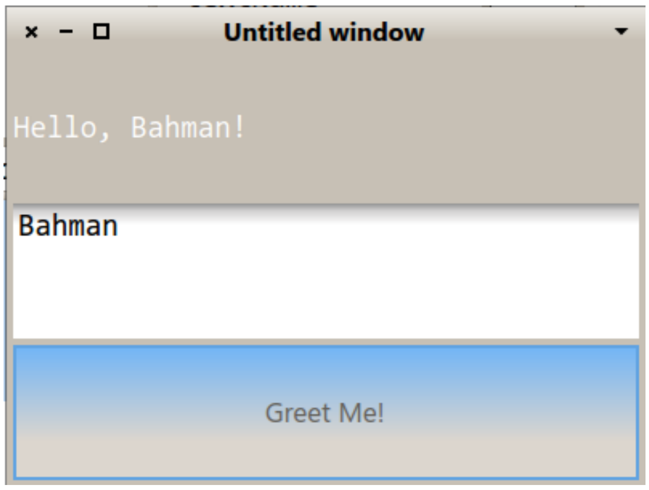


Figure 1 – Greeter version 1

Where's The Title?

One of the things that discriminates our greeter from a professional one is that it lacks the user's title in the greeting message. To achieve this, we add 3 radio buttons for the common titles Mr., Mrs., and Ms. to the greeter.

As usual, first we add the instance variables:

```
ComposableModel subclass: #MyFirstWindow
	instanceVariableNames: 'labelGreeting textName buttonGreet
	classVariableNames: ''
	poolDictionaries: ''
	category: 'My-Spec-Tutorial'
```

Don't forget to **generate the accessors**.

As you know, the next step is to instantiate the radio buttons:

```
initializeWidgets
	self instantiateModels: #(
		labelGreeting	LabelModel
		textName	TextInputFieldModel
		buttonGreet	ButtonModel
		radioMr		RadioButtonModel
		radioMs		RadioButtonModel
		radioMrs	RadioButtonModel
	).
	labelGreeting text: ''.
	textName autoAccept: true.
	buttonGreet label: 'Greet Me!'; disable.
```

```
13        self setupTitleRadioButtons.
```

On lines 6, 7 and 8, as expected, each radio button is associated with a model. And to avoid polluting `initializeWidgets`, on line 13, further radio buttons setup is done in `setupTitleRadioButtons`:

```
 1 setupTitleRadioButtons
 2         radioMr label: 'Mr.'.
 3         radioMs label: 'Ms.'.
 4         radioMrs label: 'Mrs.'.
 5 
 6         RadioButtonGroup new
 7                 addRadioButton: radioMr;
 8                 addRadioButton: radioMs;
 9                 addRadioButton: radioMrs;
10                 default: radioMr.
```

On lines 6 to 10, we create a new `RadioButtonGroup`, add all the radio buttons to it and set the default one. `RadioButtonGroup` is not a widget, it's a utility class that keeps track of its radio buttons in a `Collection` and enables/disables them when one is activated.

Next step, as you already know, is telling Spec what to show:

```
defaultSpec
        ^ SpecColumnLayout new
                        add: #labelGreeting;
                        add: #textName;
                        add: #buttonGreet;
                        add: #radioMr;
                        add: #radioMrs;
                        add: #radioMs;
                        yourself.
```

And finally, the action handlers. Now that we have the title, the action handler of `buttonGreet` should be modified to take that into account.

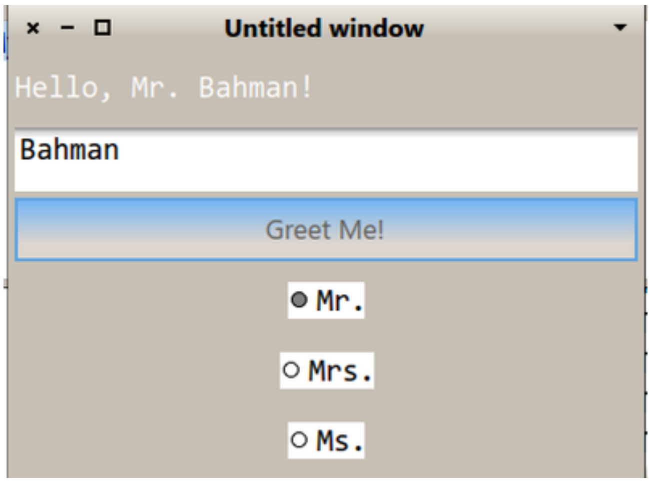
```
initializePresenter
        textName whenTextChanged: [
                buttonGreet enable ].
        buttonGreet action: [
5                 labelGreeting text: 'Hello, ', self userTitle, ' '
6                 buttonGreet disable ].
```

To avoid polluting the action handler, extracting user's title is done in `userTitle` on line 5.

```
userTitle
        "Find out user's title by checking the radio buttons."

        radioMr state
                ifTrue: [ ^ radioMr label ]
                ifFalse: [
                        radioMrs state
                                ifTrue: [ ^ radioMrs label ]
                                ifFalse: [ ^ radioMs label ] ].
```

And voila!



Hello, Mr. Bahman!

Bahman

Figure 2 – Greeter version 2

What Next?

So far we have developed a fully functional greeter. However, as you certainly have noticed, it's very ugly! But don't worry! In the next episode (Spec – Part II – The Layout), we will look at layouts and widget positioning in Spec.

Please subscribe to this RSS feed to be notified when a new episode is available.

Filed under: tutorial, spec, smalltalk, pharo, ui

Site Map
Accessibility
Contact