Chapter 1

Spec

A Spec is a static way to describe a user interface. Used as a presenter, it is a major element of UI and specific model reuse.

In this chapter, I will talk about the Spec Interpreter and how to specify model and UI, and more useful than that, how to reuse existing model and UI.

1.1 Code

The code is available on squeaksource:

```
Gofer new
   squeaksource: 'DirtyExperiments';
   package: 'Spec';
   load
```

1.2 Spec structure and Spec Interpretor

Spec Structure

To have a static structure easily inspectable, we have decided to use an array. It avoids to have to implement our own parser, and can be analyzed regardless of the implementation of Smalltalk.

A spec is composed of a receiver which is the first element of the Spec. If the first element is a class name symbol, it will be turn into an instance of this class by the interpreter. Then follow selectors and arguments. For this part, the spec is considered as a stack: each time a selector is read, as

many arguments as needed by the selector are pulled and analyzed (see the following example).

Method 1.1: *Example of a spec*

```
internSpec

  ↑{#PluggableListMorph.
      #model:.                #model.
      #getListSelector:.    #getList.
      #getIndexSelector:. #getIndex.
      #setIndexSelector:. #setIndex:.
      #wrapSelector:.      #wrapItem:.
      #hResizing:.           #spaceFill.
      #vResizing:.           #spaceFill     }
```

As the specs are defined on class side, we have introduced a specific keyword #model to create the binding between an instance and its spec.

Spec Interpreter

The goal of the Spec Interpreter is to build an instance starting from a spec.

So it iterates on a spec and recursively build each part of the spec. Right now, for instance creation, it basically takes the name of a class to create an instance (as seen in the previous example), but a better abstraction could be done by using a Dictionary with keywords as key and morphic classes as values. It allows to separate the morphic part from the presenter, and to simply change the binding to be able to create other interfaces with the same presenter (a dictionary for seaside objects by example).

Method 1.2: *Example of a recursive spec*

```
internSpec

  ↑  {  #PanelMorph.
      #changeTableLayout.
      #listDirection:. #bottomToTop.
      #addMorph:.   { #model. #listModel. #internSpec.    }.
      #addMorph:.   { #model. #textModel. #internSpec.    }.
      #vResizing:.    #spaceFill.
      #hResizing:.    #spaceFill. }.
```

1.3 Models and presenters construction

ComposableModel

ComposableModel is an abstract class which is the superclass of all applicative models used with specs.

There is a small API used to handle spec and the connection between the instances and their specs.

- morph: an instance variable used to store the UI once the interpreter have created it.
- internSpec: this is the default spec. The mechanism assumes each applicative model have such a method on class side.
- title: this method return the title of the window used to render the presenter.

Then , each model have to define his own entry and exit points used to bind them together.

Basic Models

There is an applicative model for each basic UI widget Benjamin ►*now only list, text and button*◄ with their entry point for each customizable behavior of the widget Benjamin ►*not yet full coverage*◄

ListComposableModel It is the model used to model a list. Now my entry points are:

- listHolder: a value holder which contains a list of elements to display.
- wrapHolder: a value holder which contains a block used to wrap item from the list.

My exit point is:

- selectionHolder: a selection value holder. It contains the index of the current selection and the current selected object.

TextComposableModel It is the model used to model a text area. Now my entry points are:

- textHolder: a value holder which contains the text displayed (it's also an exit point).
- actionToPerformHolder: a value holder which contains a block performed when the modified text is saved.

There are also entry points for colorization:

- aboutToStyleHolder: this value holder is evaluated to determine is the text should be colorize or not. It should return a boolean.
- behaviorHolder: this holder contains the class or metaclass related to the current text, for a good colorization.

ButtonComposableModel It is the model used to model a button. Now my entry points are:

- actionHolder: a value holder which contains a block performed when button is clicked.
- labelHolder: a value holder which contains a the label of the button.
- stateHolder: a value holder containing a boolean which determine if the button is active or not.
- enabledHolder: a value holder containing a boolean which determine if the button is enabled or not.

Composition

This is the most important point of the spec. The ease to reuse models and/or presenters. In this section, I will show you how to build a little browser of methods (like senders/implementors) and how to reuse this browser to build a classes and methods browser (like a code browser).

Methods Browser

A methods browser is composed of two areas: a list, and a text zone. And the entry point is the list of methods provided.

So let's define a MethodBrowser class

Class 1.3: *MethodBrowser class*

```
ComposableModel subclass: #MethodBrowser
   instanceVariableNames: 'listHolder listModel textModel'
   classVariableNames: ''
```

```
    poolDictionaries: ''
    category: 'Spec-Examples'
```

I let you write accessors. The will be needed by the presenter. I will just provide you a setter example, because there is a little trick:

Method 1.4: *MethodBrowser»#listHolder:*

```
MethodBrowser>>#listHolder: anHolder

    listHolder := anHolder.
    listModel listHolder: anHolder
```

The trick is to think to also set the list holder of the list model when the list holder is changed. This way, we ensure that a each every moment, the holder of the browser and the holder of the list model is the same.

So now we have the entry point and the models. So lets make the connections between them.

Method 1.5: *MethodBrowser»#listHolder:*

```
MethodBrowser>>#initialize
    "Initialization code for MethodBrowser"
    | textHolder behaviorHolder |

    super initialize.

    listModel := ListComposableModel new.
    self listHolder: {} asValueHolder.

    textHolder := ComposableValueHolder on: (listModel selectionHolder) do: [:index
        :selection | selection contents
        ifNil: [ '' ]
        ifNotNil: [:m | m sourceCode ]].

    behaviorHolder := ComposableValueHolder on: (listModel selectionHolder) do: [:index
        :selection | selection contents
        ifNil: [ nil ]
        ifNotNil: [:m | m methodClass ]].

    textModel := TextComposableModel textHolder: textHolder.

    textModel behaviorHolder: behaviorHolder.
    textModel aboutToStyleHolder contents: [ true ].
```

Lets explain what ComposableValueHolder is.

ComposableValueHolder is a special value holder used to transform the value of another value holder. As an example, lets have a look at:

```
textHolder := ComposableValueHolder on: (listModel selectionHolder) do: [:index
        :selection | selection contents
        ifNil: [ '' ]
        ifNotNil: [:m | m sourceCode ]].
```

Here we need a link between the methods list and the text area (when a method is selected, the text zone have to display its source code).

So we have a value holder which changes each time another value holder changes.

Now that links are created, we have to specify the spec on class side:

Method 1.6: *MethodBrowser class»#internSpec:*

```
MethodBrowser>>#internSpec

  ↑{ #PanelMorph.
        #changeTableLayout.
        #listDirection:. #bottomToTop.
        #addMorph:. {#model. #listModel. #internSpec.}.
        #addMorph:. {#model. #textModel. #internSpec.}.
        #vResizing:. #spaceFill.
        #hResizing:. #spaceFill.}.
```

We specify the layout to be a column, with submorphs added form top to bottom. And then for each model, we add its spec recursively.

So if you evaluate

```
| browser |
browser := MethodBrowser new.
browser openWithSpec.
```

you should have a window like the Figure 1.1.

And then, if you do

```
browser  listHolder contents: (ComposableModel methodDict values ,
      ListComposableModel methodDict values).
```

The list should automatically be updated, and looks like Figure 1.2.

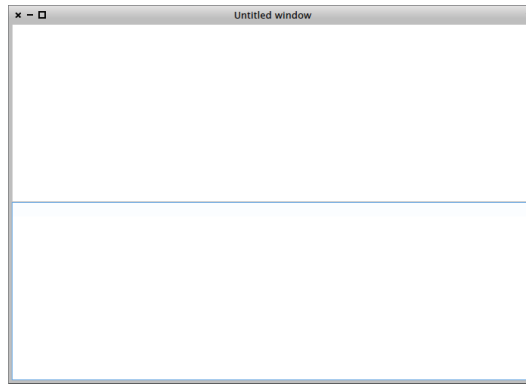

Figure 1.1: Our browser with an empty list

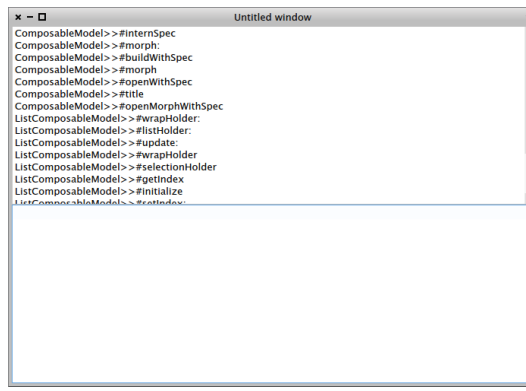

Figure 1.2: Our browser with list of methods

And if you select a method, its source code should appear, like Figure 1.3

We are almost done, only the window's title is still wrong. So we will define

Method 1.7: *MethodBrowser class»#title*

```
MethodBrowser class>>#title

    ↑ 'Method Browser'
```

So now, if you re-evaluate the code, you should have a window like Figure 1.4

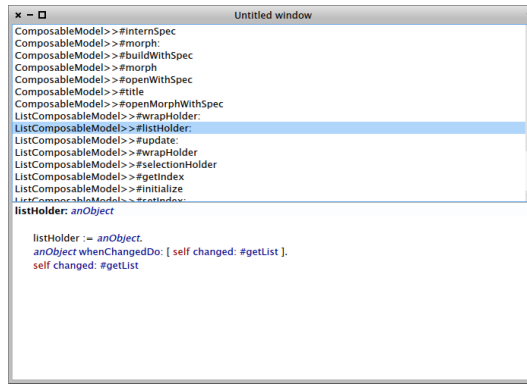

Figure 1.3: Our browser with a method selected

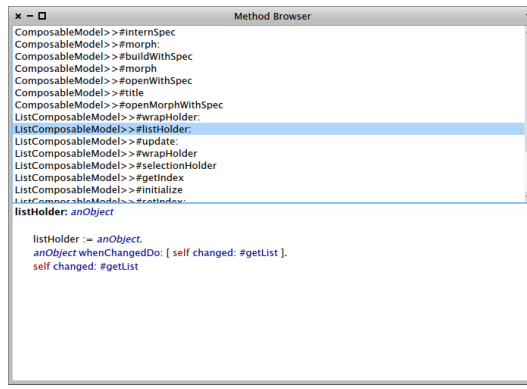

Figure 1.4: Now with a nice title

Classes and methods browser

Now, we want to build a classes and methods browser. To do that, we need a list for classes, and a methods browser. And previously, the entry point is a list, but this time, a list of methods.

Class 1.8: *ClassMethodBrowser class*

```
ComposableModel subclass: #ClassMethodBrowser
    instanceVariableNames: 'listHolder listModel methodModel'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spec−Examples'
```

As previously, I let you wrote the accessors. And as previously, we have to ensure the holder from the browser and the holder from the list is the same.

Method 1.9: *ClassMethodBrowser»#listHolder:*

```
ClassMethodBrowser>>#listHolder: anHolder

	listHolder := anHolder.
	listModel listHolder: anHolder
```

Now we have to create the link. It's simpler here than previously because there is only one link to create between the list of classes and the method browser:

Method 1.10: *ClassMethodBrowser»#initialize*

```
ClassMethodBrowser>>#initialize

	"Initialization code for ClassMethodBrowser"
	| listHolder2 |

	super initialize.

	listHolder := {} asValueHolder.
	listModel := ListComposableModel new.
	listModel listHolder: listHolder.

	methodModel := MethodBrowser new.
	listHolder2 := ComposableValueHolder on: (listModel selectionHolder) do: [:index
		:selection | selection contents ifNotNil: [:c | c methodDict values sort: [:a :b | a
		selector < b selector]]].
	methodModel listHolder: listHolder2.
	methodModel listModel wrapHolder contents: [:method | method selector ].
```

So now, lets do the spec. Firstly, let's create the top row, with the two lists

Method 1.11: *ClassMethodBrowser class»#topSpec*

```
ClassMethodBrowser class>>#topSpec

	↑{ #PanelMorph.
		#changeTableLayout.
		#listDirection:. #rightToLeft.
		#addMorph:. {#model. #listModel. #internSpec.}.
		#addMorph:. {#model. #methodModel. #listModel. #internSpec.}.
		#vResizing:. #spaceFill.
		#hResizing:. #spaceFill.}.
```

And now we can do the interSpec:

Method 1.12: *ClassMethodBrowser class»#interSpec*

```
ClassMethodBrowser class>>#interSpec

	↑ { #PanelMorph.
		#changeTableLayout.
		#listDirection:. #bottomToTop.
		#addMorph:. self topSpec.
		#addMorph:. {#model. #methodModel. #textModel. #internSpec.}.
		#vResizing:. #spaceFill.
		#hResizing:. #spaceFill.}.
```

In each spec, we call the spec of our internal models. Lastly, we can specify a title for the window:

Method 1.13: *ClassMethodBrowser class»#title*

```
ClassMethodBrowser class>>#title

	↑ 'Class Method Browser'
```

Then, if you evaluate

```
| browser |
browser := ClassMethodBrowser new.
browser openWithSpec.
browser listHolder contents: (Smalltalk allClasses).
```

you will get a nice browser like Figure 1.5.

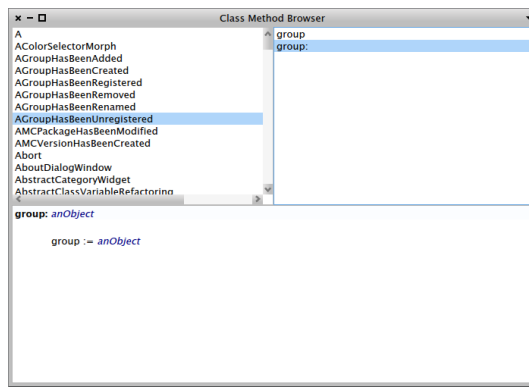

Figure 1.5: Our classes and methods browser with a method selected

1.4 Conclusion

As conclusion, you have now tools to create quickly Uis and to reuse them. So you have to keep in mind that you tool can be reuse a think of providing an API to allow that.