Chapter 1

Spec

Creating user interfaces programmatically is often a tedious and repetitive task. Some attempts to offer UI-Builders solved the problem on positioning UI elements and their inter connection. However, there is still the challenge of reusing widgets and the linked logic. There is a need for a declarative way to specify widgets allowing for composition and reuse of the UI declaration and logic.

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

In this chapter, we will begin by an example of how to design a UI and most important, how to reuse an existing model and UI. Then the SpecInterpreter will be explained.

1.1 Code

The code is available on squeaksource:

```
Gofer new
   squeaksource: 'DirtyExperiments';
   package: 'Spec';
   version: 'Spec−BenjaminVanRyseghem.24';
   load
```

Example

The most important point of the spec is the reuse and the possibility of composition at two levels, UI and models. In this section we will show you how to build an abstract method browser (similar to the senders/implementor

) and how to reuse this browser to build a class and an extended method browser (similar to the code browser).

Here comes a simple example to quickly introduce to the use Specs. For now do not try to understand all the details but focus on the results. We will provide detailed insights right after the example.

Methods Browser

Firstly, let's think about how the visual structure of a method browser GUI. It is basically composed of two areas: a list and a text zone.Moreover there is an entry point which is the list of methods to be browsed.

So let's define a MethodBrowser class

Class 1.1: *MethodBrowser class*

```
ComposableModel subclass: #MethodBrowser
    instanceVariableNames: 'listModel textModel'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Spec−Examples'
```

I let you write getters. The will be needed by the presenter.

It's still miss the entry point. By the way the entry point of the method browser could also be the list model's entry point.

Method 1.2: *MethodBrowser»#methods:*

```
methods: aList

    "Here I reroute my entry point to the list model's entry point"
    self listModel items: aList
```

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

Method 1.3: *MethodBrowser»#initialize*

```
initialize
    "Initialization code for MethodBrowser"

    super initialize.

    listModel := self instantiate: ListComposableModel.
    textModel := self instantiate: TextModel.

    listModel whenSelectedItemChanges: [:selection |
        selection
            ifNil: [
```

```
            textModel text: ''.
            textModel behavior: nil ]
        ifNotNil: [:m |
            textModel text: m sourceCode.
            textModel behavior: m methodClass ]].

    textModel aboutToStyle: true.
```

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

Method 1.4: *MethodBrowser class»#internSpec:*

```
internSpec

    ↑{ #Panel.
        #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 from 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.

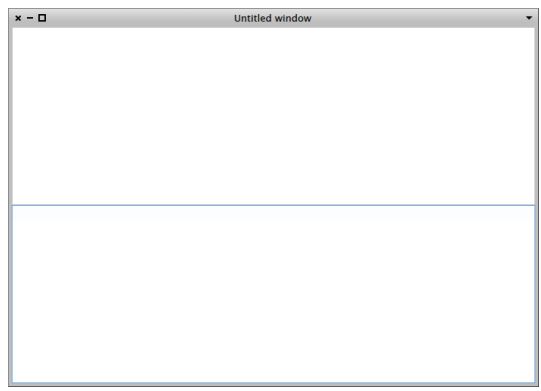

Figure 1.1: Our browser with an empty list

And then, if you do:

```
browser  methods: (ComposableModel methodDict values , ListComposableModel
      methodDict values).
```

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

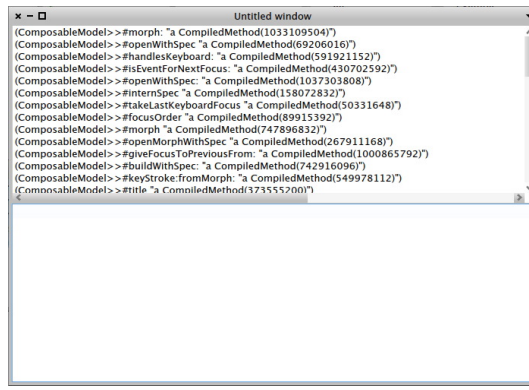

Figure 1.2: Our browser with list of methods

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

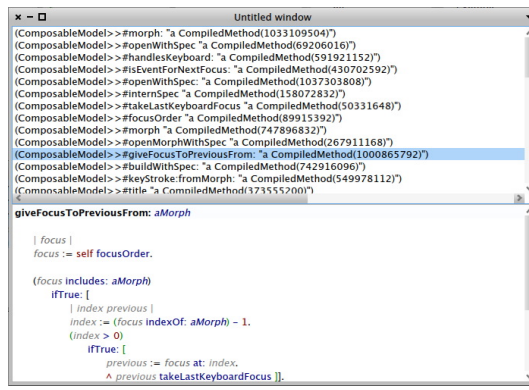

Figure 1.3: Our browser with a method selected

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

Method 1.5: *MethodBrowser class»#title*

```
title
```

```
↑ 'Method Browser'
```

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

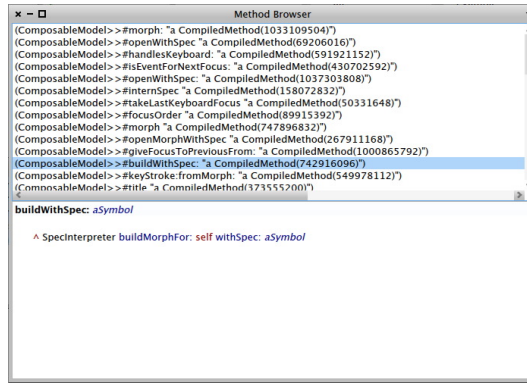

Figure 1.4: Now with a nice title

We can still improve the way the items are displayed. We just have to precise to the list model how to wrap items. So let's add the following line into `MethodBrowser>>#initialize`:

```
listModel wrappingBlock: [:item | item methodClass name,'>>#', item selector ].
```

But you could also decide to add the items wrapping as a new entry point, so let's define

Method 1.6: *MethodBrowser»#wrapWith:*

```
wrapWith: aBlock

    listModel wrappingBlock: aBlock
```

We can now use this method into `MethodBrowser>>#initialize` to finally obtain

Method 1.7: *MethodBrowser»#initialize*

```
initialize
    "Initialization code for MethodBrowser"

    super initialize.

    listModel := self instantiate: ListComposableModel.
```

```
textModel := self instantiate: TextModel.

listModel whenSelectedItemChanges: [:selection |
    selection
        ifNil: [
            textModel text: ''.
            textModel behavior: nil ]
        ifNotNil: [:m |
            textModel text: m sourceCode.
            textModel behavior: m methodClass ]].

textModel aboutToStyle: true.
self wrapWith: [:item | item methodClass name,'>>#', item selector ].
```

And you finally get a browser which looks like Figure 1.5

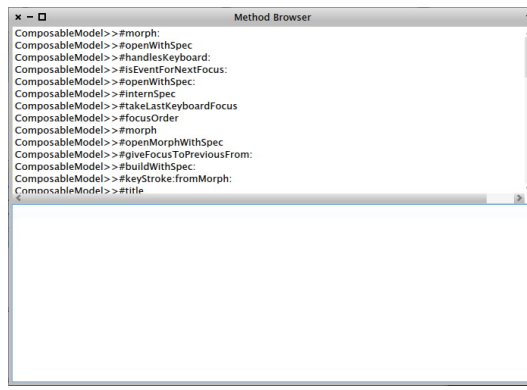

Figure 1.5: Voila

Classes and method browser

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

Class 1.8: *ClassMethodBrowser class*

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

As previously, I let you implement the getters.

For the entry point, it's deja vu:

Method 1.9: *ClassMethodBrowser»#classes*

```
classes: aList

    self listHolder contents: aList
```

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

Method 1.10: *ClassMethodBrowser»#initialize*

```
initialize
    "Initialization code for ClassMethodBrowser"

    super initialize.

    listModel := self instantiate: ListComposableModel.
    methodModel := self instantiate: MethodBrowser.

    listModel whenSelectedItemChanges: [:selection |
        selection
            ifNotNil: [:class |
                methodModel methods: (class methodDict values sort: [:a :b | a selector < b
        selector]).
                methodModel listModel resetSelection ]].

    methodModel wrapWith: [:method | method selector ].
```

So now, let's do the spec.

Method 1.11: *ClassMethodBrowser class»#internSpec*

```
internSpec2

    ↑ { #Panel.
            #changeProportionalLayout.
            #addMorphWrapper:.
                {   #MorphWrapper.
                            #morph:.    {#model. #methodModel. #listModel. #internSpec.}.
                            #frame:.    ( 0@0 corner: 0.5@0.5 )}.
            #addMorphWrapper:.
                {   #MorphWrapper.
                            #morph:.    {#model. #listModel. #internSpec.}.
                            #frame:.    ( 0.5@0 corner: 1@0.5 )}.
            #addMorphWrapper:.
                {   #MorphWrapper.
```

```
                    #morph:.    {#model. #methodModel. #textModel. #internSpec.}.
                    #frame:.    ( 0@0.5 corner: 1@1 )}.
    #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 1.12: *ClassMethodBrowser class»#title*

```
ClassMethodBrowser class>>#title

  ↑ 'Class Method Browser'
```

Then if you evaluate:

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

you will get a nice browser like in Figure 1.6.

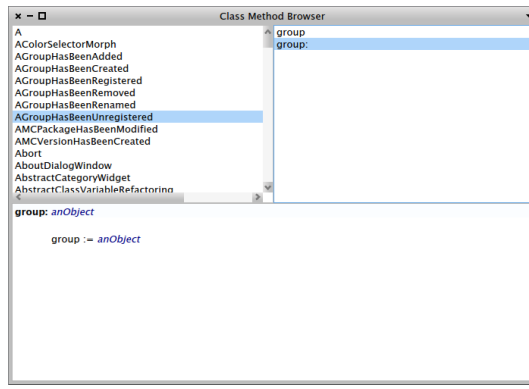

Figure 1.6: Our classes and method browser with a method selected

Here ends this example. Basically you now know how to implement basic widget and how to compose Spec widgets together to get new powerful widgets.

Those examples (with few improvements) and few others are provided in the package `Spec-Example`, feel free to have a look.

1.2 Spec structure and Spec Interpretor

Now that you've seen an example I will explain how it works and what is the principle of the mechanism behind.

Spec Structure

The presenter side of a Spec is a structure used to simply hold a figurative representation of what you want your UI to look like. The more static it is

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

A spec is composed of a receiver followed by selector and arguments. The receiver is the first element of the Spec. If the first element is a class name symbol, it will be turned into an instance of this class by the interpreter. The following selectors and arguments are used 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 1.13: *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.

The spec interpreter iterates over a spec and recursively builds each part of the spec. Right now, for instance creation, you can either specify a keyword (as `#Panel`, `#List`, ...) bound to a rendering class using a specific dictionary or a class name. It allows the developer to separate the rendering part from the formal representation which is the presenter, and to simply

change the binding to be able to create other interfaces with the same presenter (a dictionary for seaside objects for example).

Method 1.14: *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 widget models used with specs.

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

- focusOrder: an ordered list of your sub-models sorted in the order the developer want the focus to change.
- internSpec: this is the default spec. The mechanism assumes each applicative model have such a method on class side.
- morph: an instance variable used to store the UI once the interpreter have created it.
- title: this method return the title of the window used to render the presenter.

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

Basic Models

There is an widget model for each basic UI widget **Benjamin** ►*now only list, text, button, radiobutton, checkbox and, droplist*◄ with their entry points for each customizable behavior of the widget and exit point for information you want to retrieve. **Benjamin** ►*almost full covered*◄

ListComposableModel is the model lists.

The entry points are:

- items: set the list of items to be displayed.
- menu: set the block used to indirect messages used to create menus.
- wrappingBlock: set the block used to wrap items from the list.

My exit point is:

- selectedIndex: the index of the currently selected item (`0` if none).
- selectionItem: the currently selected item (`nil` if none).

You can communicate with this widget using:

- resetSelection: to reset the selection of the widget.
- setSelectedIndex: : to manually set the index of the item you want to be selected.
- setSelectedItem: : to manually set the item you want to be selected.
- updateList: to manually ask the list to refresh its own display.

TextModel It is the model for text areas.

The entry points are:

- actionToPerform: a value holder which contains a block performed when the modified text is saved.
- text: set the text to be displayed.

There are also entry points for text shouting:

- aboutToStyle: set wether or not the text should be shouted.
- aboutToStyleBlock: set the block used to know if wether or not the text should be shouted.
- behavior: set the behavior corresponding with the text currently displayed if relevant. It's used to properly colorize the syntax elements.

You can communicate with this widget using:

- setSelection: : used to manually specify the range you want o be selected.

ButtonModel It is the model for buttons.

The entry point is:

- action: set the block executed when the button is clicked.

There is no exit points.

You can communicate with this widget using:

- label: : set the button's label.
- state: : set wether the button is on or off.
- enabled: : set wether the button is activated or not.

RadioButtonModel/CheckboxModel They are the models for RadioButton and Checkbox. Since they are almost the same widget, they share the same API.

The entry points are:

- activationAction: : set the action block to be valued when the item is activated.
- desactivationAction: : set the action block to be valued when the item is desactivated.

There is no exit points.

You can communicate with this widget using:

- label: : set the button's label.
- state: : set wether the button is on or off.
- enabled: : set wether the button is activated or not.

DropListModel It is the model for drop lists.

The entry points are:

- items: set the list of items to be displayed. You have to provide a collection of DropListItem instances.

My exit point is:

- selectedIndex: the index of the currently selected item (`0` if none).
- selectionItem: the currently selected item (`nil` if none).

You can communicate with this widget using:

- resetSelection: to reset the selection of the widget.
- setSelectedIndex: : to manually set the index of the item you want to be selected.
- setSelectedItem: : to manually set the item you want to be selected.
- updateList: to manually ask the list to refresh its own display.

1.4 Conclusion

As conclusion, you have now tools to quickly create UIs and to reuse them. Since the reuse is really a strong value for specs, keep in mind that your tools can be reused. So do not forget to provide a proper API for it.