Chapter 1

Building a Simple Calculator with Spec

Written by Ignacio Sniechowski

Spec is a new way to build user interfaces with Pharo.With Spec you define an application model that glue together widgets and your model. In this chapter we will develop a simple calculator to get started with Spec.Our focus will be on building the GUI to come up with a simple but good-looking calculator. After designing the GUI we will move on to render this nice, but hollow interface operational (let's say we will focus first on the VC part of MVC, and later on the M). This chapter uses the version 2 of Spec that is available in Pharo 3.0. Once it's finished, our calculator will look like Figure 1.1.

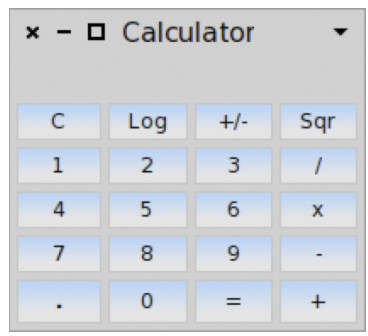

Figure 1.1: The Spec Calculator.

1.1 Introduction

To construct our GUI we need to know what Spec expects to find in the class we will be creating. There are certain selectors and methods that have to be present at instance and class side and that will be used by Spec when we do Specalc new openWithSpec.

Basically, at the instance side of the class we will declare the widgets that we will be using, some aspects of these widgets like labels, actions, states, and the logic of the application and the glue between these widgets. On the other hand, at class side, we will set up the basic design of our GUI -how the widgets are laid out- and its title.

Summarizing:

- instance side: widget instantiation along with actions performed by these and application logic.
- class side: widget layout i.e., how all our widgets will be laid out in a window.

So let us start by analyzing the widgets we will be using, how to glue them together and adjust them to get a usable GUI.

1.2 The widgets

In this simple calculator, we will need buttons to represent the 0 to 9 digits, some basic arithmetic operations, the dot to use floating numbers, a clear screen button, the = button to get the result, and some others like changing the sign, square rooting, etc. Also, we will need a display to see the numbers entered and get the result.

But first things first. We have to create the class from which our calculator will be instantiated; we are going to use a composed layout model for our class, therefore our class will be a subclass of ComposableModel.

So open the browser and create a new package called Specalc, enter the following class definition and accept it. Don't worry, for now, about the instance variables we will get to that part soon.

Defining the class Specalc

```
ComposableModel subclass: #Specalc
   instanceVariableNames: ''
   classVariableNames: ''
```

```
  poolDictionaries: ''
  category: 'Specalc'
```

Remember that we mentioned that when an instance of our calculator is openWithSpec, Spec will send a message to some selectors that expect to find in our class both at class and instance side.

Our starting point is to implement the initializeWidgets method. We want to have:

- 10 buttons for the digits (0 to 9)
- A plus button, minus button, multiplication button, division button, and change sign button
- A clear display button, a dot button for entering float-numbers, an equals button, a square root button, and a log button.

That is a total of 20 buttons. In addition, we will need the display. This method can grow considerably depending on the number of widgets you might need. Therefore, it is sometimes convenient to break it down into simpler methods. That's what we are going to do:

If the GUI you are building has only a limited number of widgets, you might even want to send some messages to the newly instantiated objects to set some of their attributes. In our case, as we said before, we are going to leave that to another method to avoid excessive cluttering.

Our initializeWidgets method

```
Specalc>>initializeWidgets
  "instantiate the digit buttons"
  self initializeDigitButtons.
  "instantiate the remaining buttons"
  self initializeOtherButtons.
  "instantiate the display"
  display := self newLabel
```

Upon accepting, you will be prompted whether initializeDigitButtons and, initializeOtherButtons are typos or are methods to be coded later. Of course, it is not a typo, these methods have not been defined yet. The same will happen to the instance variable display, so select 'declare as new instance variable' when prompted.

Then we define the other methods:

The initializeDigitButtons method

```
Specalc>>initializeDigitButtons
  "instantiate the digit buttons"
  button0 := self newButton.
  button1 := self newButton.
  button2 := self newButton.
  button3 := self newButton.
  button4 := self newButton.
  button5 := self newButton.
  button6 := self newButton.
  button7 := self newButton.
  button8 := self newButton.
  button9 := self newButton.

  self defineDigitButtons
```

The same happens here, accept all buttonX as instance variables and defineDigitButtons as a new selector.

The initializeDigitButtons method

```
Specalc>>initializeOtherButtons
   "instantiate the basic arithmetic buttons"
   buttonPlus := self newButton.
   buttonMinus := self newButton.
   buttonMult := self newButton.
   buttonDiv := self newButton.
   buttonEq := self newButton.
   "instantiate the remaining buttons"
   buttonDot := self newButton.
   buttonSqr := self newButton.
   buttonLog := self newButton.
   buttonChange := self newButton.
   buttonClear := self newButton.

   self defineOtherButtons
```

Finally, categorise these three methods including them in the initialization protocol.

Defining the widgets

Next, we define the methods defineDigitButtons and defineOtherButtons that are needed. These methods send messages to the instance of buttons and label we have just instantiated defining what to show, what to perform, etc.

The defineDigitButtons method

```
Specalc>>defineDigitButtons
    button0
        label: '0';
        action: [ display label: ((display label,'0') asNumber) asString];
        state: true.

    button1
        label: '1';
        action:  [ display label: ((display label,'1') asNumber) asString ];
        state: true.

    button2
        label: '2';
        action: [ display label: ((display label,'2') asNumber) asString ];
        state: true.

    button3
        label: '3';
        action: [ display label: ((display label,'3') asNumber) asString ];
        state: true.

    button4
        label: '4';
        action: [ display label: ((display label,'4') asNumber) asString ];
        state: true.

    button5
        label: '5';
        action: [ display label: ((display label,'5') asNumber) asString ];
        state: true.

    button6
        label: '6';
        action: [ display label: ((display label,'6') asNumber) asString ];
        state: true.

    button7
        label: '7';
        action:[ display label: ((display label,'7') asNumber) asString ];
        state: true.

    button8
        label: '8';
        action: [ display label: ((display label,'8') asNumber) asString ];
        state: true.

    button9
```

```
label: '9';
action: [ display label: ((display label,'9') asNumber) asString ];
state: true
```

This is a long method, but it's very simple to understand because once we have grasped the first button, the rest is similar. So here is what each of these lines do. First we had the object instantiated in the previous methods: that's button0. If you follow the sequence of messages it's a cascade sent to the button0 object.

We tell button0 to display some label on it, using the label: message with a string as argument -in this case '0'. The expression button1 label:'1' will obviously make the button show the digit 1 as label. Next is another selector: action: followed by a block.

Let's analyze what this block does. Recall that we instantiated a display object from Spec's LabelModel class which is actually the screen of our calculator. Following the precedence of messages, the first evaluated expression is: (display label,'9') We are sending a message to our display to retrieve the contents of it and concatenate the string '9'. Why? Basically because our display object might content some string of numbers previously entered and we want to put the newly pressed number behind of what's already on the screen. Then you noticed that we send the message asNumber, the purpose of this, is to avoid entering zeros that are not significant. When we convert our string to a number the non-relevant zeros are discarded otherwise we will have a calculator that will show awkward numbers like 000234, when only 234 should be displayed. Finally we convert that back to a string and pass this as an argument to the label: selector of our display. The result is that the display now contains what was previously there plus the content of our recently pushed button.

The last message we send is state: which accepts as an argument a boolean. We set it to true to avoid having to double click on each button to have a value entered on the display. You will notice when buttons are active because they show a pale blue shadow. Try setting the state to false and you will see what annoying is to have to press two times each button.

SD: we should check with ben because this look strange to me. The name is strange.

IS: To what name are your referring?.

A comment on style is appropriate here. The exposed method could have been written in a much more elegant and simple way, making the digit button object pass its label instead of hard-coding the string '1' or '2', etc. However, we choose not to do it that way to keep it easier to understand and follow.

Now is turn to format the other buttons. This is a little more tricky but no rocket science at all. Let's start by taking a look at the method, and then define one method per action and analyse it.

Defining the other buttons

```
Specalc>>defineOtherButtons
   "basic arithmetic operations"
   self addition.
   self subtraction.
   self multiplication.
   self division.
   "other arithmetic opertions"
   self squareRoot.
   self logarithm.
   self changeSign.
   "remaining buttons"
   self clearDisplay.
   self enterDot.
   self equals.
```

This method all it does is send a message to self for each of the remaning buttons that are not digit buttons. Categorize these two methods under the button-defintion protocol.

We will begin with the basic arithmetic operations: addition, subtraction, multiplication, and division; which will be performed by our buttonPlus, buttonMinus, buttonMult, buttonDiv respectively.

Setting up our calculator

```
Specalc>>addition
   buttonPlus
      label: '+';
      action: [ self operation: $+ ];
      state: true
```

The method sends a cascade of three keyword message to our previously instantiated buttonPlus. The first one label:'+' sets the label our button will show, the second one action:[...] will be analysed shortly. Finally, the last message state: true which accepts as an argument a boolean and, as mentioned before, we set it to true to avoid having to double click on each button.

So now let's concentrate on the action: [...] selector, which in this case is [self operation: $+]. Clearly, the operation: aCharacter method -which hasn't been defined yet- accepts as argument a character literal $_ that will allow

our calculator's model know what type of arithmetic operation we intend to perform.

So before continuing defining the other three basic arithmetic operations, we will take a close look at the operation: method.

The operation: method

```
Specalc>>operation: aCharacter
   "Defines which operation will be performed: it can be a + / − x"

   "stores the first value entered"
   previousNumber := (display label) asNumber.

   "Informs our model about the operation to be performed"
   currentOperation := aCharacter.

   "clears the display"
   display label: ''
```

The first line stores the current content of our display object into an instance variable we named previousNumber, as a number so it's ready to be used by the basic binary arithmetic selectors. Then we store the character literal passed as argument to the operation: selector and store it into another instance variable called currentOperation. And finally we reset the display making it ready to receive another string: remember that the display of our calculator shows strings. Up to here, we have an instance variable that holds our previously entered number and an instance variable that holds a character literal that will inform our model the operation we will perform once the second number is entered.

So let's go back to the rest of the arithmetic operations which methods are similar to addition.

```
Specalc>>subtraction
   buttonMinus
      label: '−';
      action: [ self operation: $− ];
      state: true
```

```
Specalc>>multiplication
   buttonMult
      label: 'x';
      action: [ self operation: $x ];
      state: true.
```

```
Specalc>>division
```

```
    buttonDiv
        label: '/';
        action:  [ self operation: $/ ];
        state: true.
```

So enter the four basic arithmetic methods, the operation: method, and categorise them under the operations protocol

Now let's focus on those operations that do not require more than a number and a button press such as obtaining the square root, changing the sign, and getting the logarithm of an entered number.

```
Specalc>>changeSign
    buttonChange
        label: '+/−';
        action: [ display label: (display label asNumber negated asString) ];
        state: true
```

This method simply changes the sign of the digit displayed on the screen. The message precedence is crucial here so it's important to follow it.

Parenthesis are evaluated first, therefore (display label asNumber negated asString) gets the precedence. As all the message following our display object are unary, the contents are evaluated as usual from left to right. The sequence is: First we get the contents of the display by sending the label selector to our display object, second we convert this string into a number, and third negated it, finally convert it again to a string. After the parenthesis is evaluated the resulting object is sent back to our display object using the label: selector.

```
Specalc>>squareRoot
    buttonSqr
        label: 'Sqr';
        action: [ self sqrCheck ];
        state: true
```

Obviously to fully grasp the intention of this method we have to take a look a sqrCheck. What this method does is simple check that we are not attempting to square root a negative number.

```
Specalc>>sqrCheck
    "checks whether we are attempting to calculate the square root of a negative
        number"
    (display label asNumber < 0)
        ifTrue:  [ display label:'Error' ]
        ifFalse: [ display label: (display label asNumber sqrt asString) ]
```

This method shows Smalltalk's beauty and clarity, almost like pidgin English being, perhaps, (display label asNumber sqrt asString) the only part that requires an explanation. As in the previous method, between parenthesis

we have an object followed by unary messages that are evaluated form left to right and then sent back to out display calculator.

Let's go ahead and enter this methods, categorising the first two into the operations protocol and the last one into the private protocol.

You should do:

Finally, the buttonEq is the core of our calculator and will be discussed once we have finished building the GUI. So go ahead and define the formatOtherButtons method accepting those selectors that are not yet defined.

1.3 Class-side methods

The ~~Dark~~ Class side of Spec

Now that we have defined, instantiated and formatted all the necessary widgets to build our calculator it's time to instruct Spec how to layout all this elements. To do that, we'll have to create some class side methods. We are going to begin with defaultSpec which is what Spec will use to layout our model.

So here is the method, remember that has to be entered at class-side:

The class-side defaultSpec method

```
Specalc class>>defaultSpec
  <spec>

  ^ SpecLayout composed
    newColumn: [  :column | column  add: #display;

          newRow: [ :row1 | row1  add: #buttonClear;
                            add: #buttonLog;
                            add: #buttonChange;
                            add: #buttonSqr ]
          height: 26;

          newRow: [ :row2 | row2  add: #button1;
                            add: #button2;
                            add: #button3;
                            add: #buttonDiv ]
          height:26;

          newRow: [ :row3 | row3  add: #button4;
                            add: #button5;
```

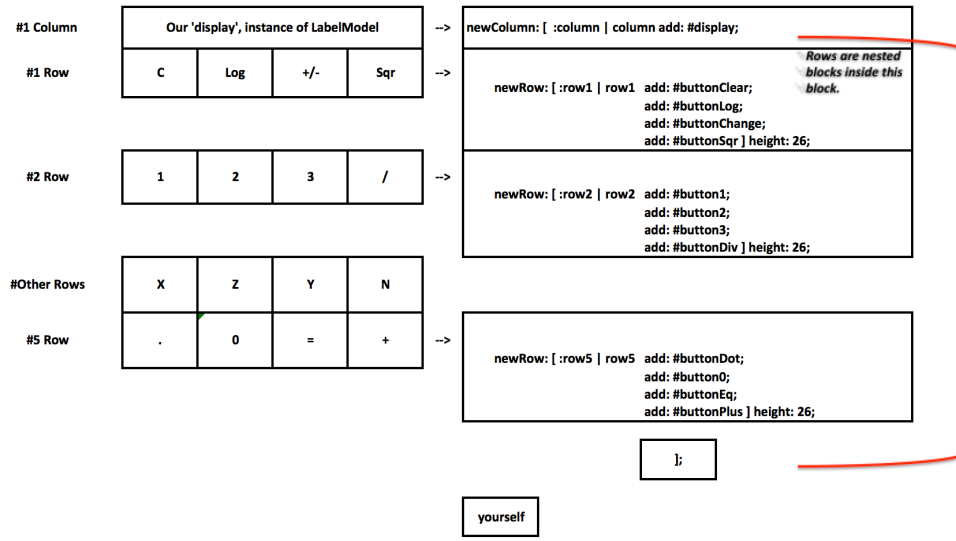

Figure 1.2: Composed Layout.

```
                    add: #button6;
                    add: #buttonMult ]
        height: 26;

        newRow: [  :row4 | row4 add: #button7;
                    add: #button8;
                    add: #button9;
                    add: #buttonMinus ]
        height: 26;

        newRow: [  :row5 | row5 add: #buttonDot;
                    add: #button0;
                    add: #buttonEq;
                    add: #buttonPlus ]
        height: 26.
                ];
    yourself.
```

The following Figure 1.2, will help us clarify how a SpecLayout works.

Basically we are setting up our widgets using a kind of matrix approach. Note, however that there's only one column and the rows are nested inside of it. So what this method actually does is to return SpecLayout composed [...], now if you inspect SpecLayout class you will notice that at class-side there's a method called composed which actually instantiates a SpecLayout object with

#ContainerModel as its type. And what's a ContainerModel? If you browse that class you will find the answer, but here is its comment: "I am a model for a container. My purpose is to hold multiple subwidgets to glue them together. I should not be used directly"

Therefore our [...] block in the previous paragraph, are exactly subwidgets glued together to form this container!. We are not going to go into details of how Spec works as there's a chapter devoted entirely to it.

Nevertheless, newColumn: and newRow: height: are two of many of the messages SpecLayout can answer and you should really browse that class to know the subtle differences among some of them.

One of the rows of our ContainerModel

```
newRow: [ :row1 | row1  add: #buttonClear;
                        add: #buttonLog;
                        add: #buttonChange;
                        add: buttonSqr ]
height: 26
```

We are sending a keyword message here, newRow:height:, first with a block that, when evaluated adds each of the widgets we have created to a new row; and second with the size of our buttons.

On a separate note, you can name this class-side method whatever you want as long as the <spec> pragma is there, but better use the convention to name it defaultSpec.

The final class-side method we have to define is the one that will let Spec know how to title our window. It's very easy an self-explanatory.

Putting a title in our window

```
Specalc class>>title

  ^ 'Calculator'
```

Pretty simple, isn't it?.

1.4 Almost ready to calculate

So let's go ahead and execute our calculator. Open a Workspace, type and do: Specalc new openWithSpec

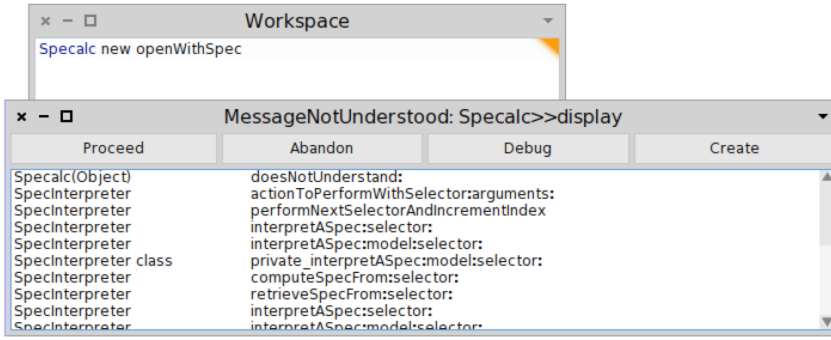

Figure 1.3: The debugger.

The debugger will complain with a: MessageNotUnderstood :Specalc>>display

The following Figure 1.3 shows the Workspace with Specalc new openWith-Spec and the outcome of the debugger after execution.

What is happening is that we didn't add yet the accessing methods to our widgets. Therefore, SpecInterpreter has no access to them. The easiest way to solve this is to select the 'Create' option which will automatically create that method for us. Go ahead and select 'Create' and when prompted choose to define display in Specalc class. For the moment do not waste time categorizing the method and just select 'OK'.

The debugger will now present you the automatically created method as shown in figure 1.4.

Select the code, accept it and then check 'Proceed' the same will happen again but with another of our widgets; therefore you must proceed exactly as previously explained until all our widgets have a proper accessor method. Once you're finished with these sequences, the GUI of our calculator should appear!

Finally, the GUI of our calculator 1.5.

But something looks awkward and not quiet well. Especially if we compare what we are seeing with the figure at the beginning of this chapter. We are almost there thou, categorize the methods so they appear under the 'accessing' protocol and let's fix this.

Overriding the initialExtent method

What we have to do, is to override DisplayObject>>initialExtent to resize our window to suit our needs.

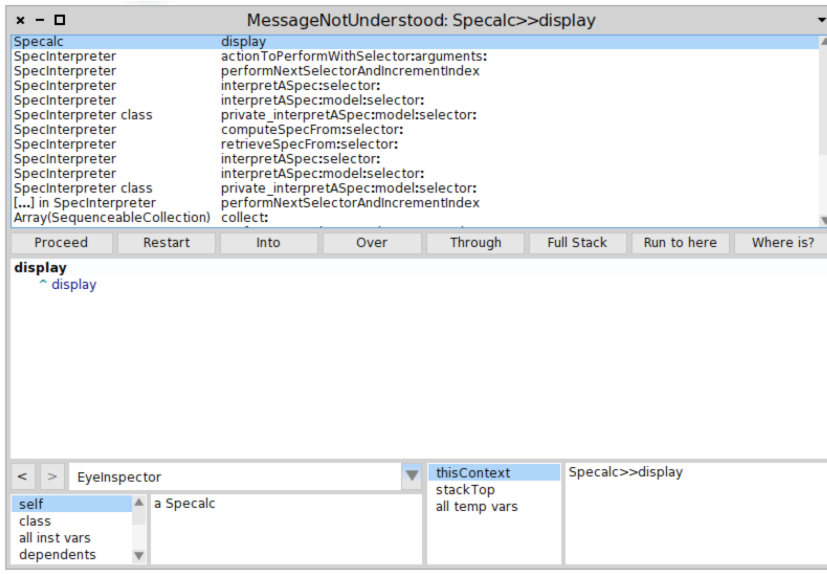

Figure 1.4: Creating a method in the debugger.

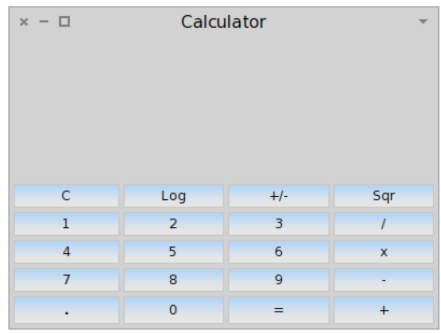

Figure 1.5: Specalc's first run.

Overriding #initialExtent

```
Specalc class>>initialExtent
    "(4@4) is the size of the border"
    ^ (207@187) + (4@4)
```

Now, execute again `Specalc new openWithSpec` and you should see the calculator as shown in Figure 1.1.

1.5 Building the core model of our calculator.

Our's calculator core model is really simple and works in three steps:

1. Enter the first number and store it in an instance variable called previousNumber.
2. Enter an operation. This could trigger an immediate result -like in sqrt, log or sign change- or store the operation in another instance variable called currentOperation.
3. Enter another number and press the equals (=) button to obtain the result.

1.6 Conclusion

This chapter illustrates how to use Spec to build applications. A deeper explanation of Spec will be available in a companion chapter.