CHAPTER 1

Layout Construction

Placing and describing the behavior of widgets on resize of their container is an important yet complex issue. In this chapter we present the different ways to express layouts with Spec. We describe how layouts of reused widgets can be composed to create the layout of the composite widget.

1.1 About layouts

Layout of widgets in a window or in their reusing UI is a nontrivial problem. This is because there are many factors at play in determining where a widget should be placed in its container (the container can be a window or another widget).

A straightforward solution is to place widgets at absolute coordinates of the enclosing container, but this breaks when the container is resized: widgets do not grow or shrink together with the container. However, if the window is never resized this is not really an issue either, and absolute positioning allows for pixel-perfect placing of every widget.

Since there are multiple usage scenarios, multiple ways for widgets to be laid out need to be provided. Spec provides various options to perform layouts and these are visible as methods in the `SpecLayout` class, in the `commands` protocols. Up until now, as layouts go we have only seen the use of rows and columns, and the adding of a single widget to the container. Furthermore, in all of these results, a row, column or a single widget always took up all available space in its container, which is the default behavior.

The `add:` method on `SpecLayout` only allows one widget to be added, so if you want more than one widget in your user interface you will need to specify one of the layouts we present in this chapter. We discuss in depth two

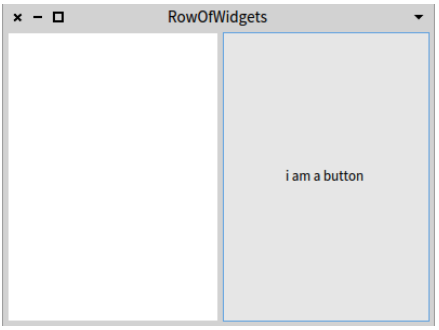

Figure 1.1 Screen shot of a row of widgets.

broad strategies for laying out widgets: first the various options for specifying rows and columns, and second the diverse ways in which widgets can be laid out freely. Each of them has their advantages and disadvantages depending on the kind of UI that is being constructed so it is best to know all of them well enough to be able to make the tradeoff.

A working example

To illustrate these layouts, we use an example class that has two buttons, a list and a text field, the non-layout code of which is below:

```
ComposableModel subclass: #LayoutExample
  instanceVariableNames: 'list button button2 text'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
LayoutExample >> initializeWidgets
  button := self newButton.
  button2 := self newButton.
  list := self newList.
  text := self newText.
  button label: 'i am a button'.
  button label: 'me too!'.
```

1.2 Row and column layouts

A straightforward and often used kind of layout to have more than one widget is to have the widgets aligned in rows or columns. Spec provides for an easy way to specify such layouts through the use of the `newRow:` and `newColumn:` messages on the `SpecLayout` class. They create, respectively, a row or a column in which the space given to the widgets is evenly distributed.

For example, the code below lays out the list and the button in a row:

Figure 1.2 Screen shot of a column of widgets.

```
LayoutExample >> oneRow
  ^ SpecLayout composed
    newRow: [ :row | row add: #list; add: #button ];
    yourself
```

This code is a layout method that builds a row of widgets using the `newRow:` message. The argument of the message is a one-argument block, and the block argument will contain an instance of a `SpecRowLayout`. The widgets are then added to this layout object, which aligns them all in a row.

The code below opens the example with this layout specification, resulting in the UI shown in Figure 1.1. The `title:` message is self-explanatory.

```
| le |
 le := LayoutExample new.
 le title: 'RowOfWidgets'.
 le openWithSpec: #oneRow
```

Having the widgets rendered as a column is similar, the only real difference being that instead of the `newRow:` message, a `newColumn` message is used. This message also takes a one-argument block, and the block argument will contain an instance of a `SpecColumnLayout`.

```
LayoutExample >> oneColumn
  ^ SpecLayout composed
    newColumn: [ :col | col add: #list; add: #button ];
    yourself
```

The code below opens this example, resulting in Figure 1.2. Note that for brevity, in the remainder of this chapter we will not include any more UI opening code as it is straightforward.

```
| le |
 le := LayoutExample new.
 le title: 'ColumnOfWidgets'.
 le openWithSpec: #oneColumn.
```

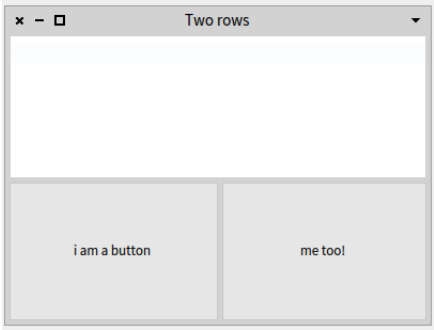

Figure 1.3 Screen shot of two rows.

Spec also allows for using the `SpecRowLayout` and `SpecColumnLayout` directly instead of using a `SpecLayout`. This makes the above code for the `oneRow` and `oneColumn` layout methods more concise:

```
LayoutExample >> oneRowConcise
  ^ SpecRowLayout composed
      add: #list; add: #button;
      yourself.
```

```
LayoutExample >> oneColumnConcise
  ^ SpecColumnLayout composed
      add: #list; add: #button;
      yourself
```

1.3 Combining rows and columns

Rows and columns can be combined to build more complex layouts, by sending both the `newRow:` and `newColumn:` messages in different combinations. We show some examples here to illustrate the possibilities.

The first example shows how we can have two rows of widgets: we create a column and twice call `newRow:`. The result is shown in Figure 1.3.

```
LayoutExample >> twoRows
  ^ SpecColumnLayout composed
      newRow: [ :row | row add: #text ];
      newRow: [ :row | row add: #button; add: #button2 ];
      yourself
```

> **Note** To have multiple rows, these need to be added to a `SpecColumnLayout`, and to have multiple columns, these need to be added to a `SpecRowLayout`. Sending `addRow:` or `addColumn:` to a `SpecComposedLayout` multiple times will only produce the last row, resp. column.

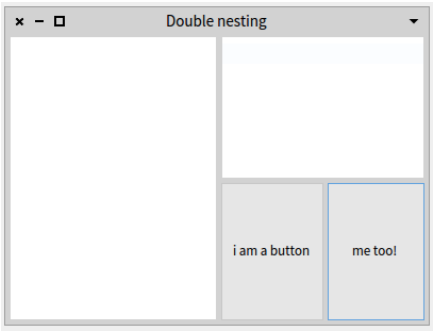

Figure 1.4 Screen shot of multiply-nested rows and columns.

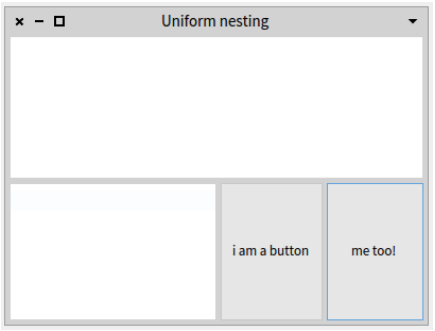

Figure 1.5 Screen shot of multiply-nested rows.

Rows and columns can of course also be multiply-nested, for example here we add the two buttons in a row that is nested in a column that is nested in a `SpecRowLayout`. The resulting UI is shown in Figure 1.4.

```
LayoutExample >> nesting1
  ^ SpecRowLayout composed
    newColumn: [ :col | col add: #list];
    newColumn: [ :col |
      col
        add: #text;
        newRow: [ :row |
          row
            add: #button;
            add: #button2]
    ];
    yourself
```

In addition to nesting columns in rows (and vice-versa), rows can also be nested in rows (and columns in columns), which allows the used space per

widget to be uniformly halved, as shown in Figure 1.5

```
LayoutExample >> nesting2
  ^ SpecColumnLayout composed
    newRow: [ :row | row add: #list];
    newRow: [ :row |
      row
        add: #text;
        newRow: [ :inRow |
          inRow
            add: #button;
            add: #button2]
    ];
    yourself
```

1.4 Setting row and column size

By default, rows and columns take up all available space, and the space in a row (and in a column) is evenly distributed across all elements of that row (or column). In this section we show three different ways in which the size of rows and columns can be changed. The first is by letting the user resize them and the two last are two different ways in which their size can be specified.

Adding UI splitters so the user can resize

A simple resizing option is to allow the user to resize widgets horizontally (for rows) and vertically (for columns). This is done by adding an `addSplitter` message between the `add:` messages for the widgets of that row or column. For example, the code below makes the horizontal line between the list and the button draggable up or down. Visually the result is the same as in Figure 1.2.

```
LayoutExample >> oneColumnWithSplitter
  ^ SpecColumnLayout composed
      add: #list;
      addSplitter;
      add: #button;
      yourself
```

Size in pixels

Programmatically it is possible to explicitly specify the absolute size of a row or column in pixels by using the `newRow:height:` and `newColumn:width:` methods. This is useful, for example, if a row of buttons is to be placed above or below a text field where it can avoid an ugly layout with huge buttons as seen previously in Figure 1.3.

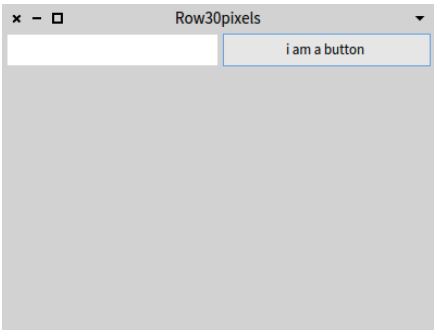

Figure 1.6 Screen shot of a row of 30 pixels high.

Figure 1.7 Screen shot of a column of 50 pixels wide.

We show two simple examples of the use of absolute size below, the first leading to the UI of Figure 1.6 and the second to the UI of Figure 1.7.

```
LayoutExample >> rowOf30
  ^ SpecLayout composed
    newRow: [ :row | row add: #list; add: #button] height: 30;
    yourself
```

```
LayoutExample >> columnOf50
  ^ SpecLayout composed
    newColumn: [ :col | col add: #list; add: #button] width: 50;
    yourself
```

Alternatively, inside of a column layout and a row layout, `add:height:`, respectively `add:width:`, can also be used to state the height, respectively the width, of that specific widget.

Note It is considered a bad practice to hardcode the size of widgets in pixels, as changes (e.g., font size) can invalidate this number. To alleviate

> this, `ComposableModel class` provides accessors for practical sizes in its `defaults` protocol.

An example of the use of accessors that compute height is `toolbarHeight`, which we used in the protocol method browser, shown in Figure **??**. This accessor depends on the font size and is also useful for sizing button rows.

Proportional layout

A last option is to specify the percentage of the container, e.g., the window, that a row or column should occupy. This is performed using the messages `newRow:top:bottom:` and `newColumn:left:right:`. In contrast with specifying size in pixels, the use of these messages will cause the row or column size to change accordingly when the container is resized.

Both the above messages take two numbers as extra arguments. These should be between 0 and 1: they are a percentage that states how far **towards the other edge** the element should begin, resp. end. For example, a column that starts at the left end of the window and takes 30 percent of the width is a `newColumn: [:c| ...] left: 0 right: 0.7` because 30 percent of the width is 70 percent away from the right edge.

> **Note** Both these numbers indicate a percentage that is towards 'the other end' of the container, **not** a percentage from top to bottom or from left to right.

A more complex example is the code below, an arguably artificial variant on Figure 1.5 that makes the buttons proportionally smaller. (The example is artificial because, e.g., for the row height of the buttons it would make more sense to use sizes in pixels.) It states that the top row takes the first 80 percent of the space (since `top: 0 bottom: 0.2`) and the second row the last 20 percent (since `top: 0.8 bottom: 0`). Furthermore, in the bottom row, the text field takes up the first 55 percent of the space (`left: 0 right: 0.45`) and the two buttons the last 45 percent (`left: 0.55 right:0`). The result of this code can be seen in Figure 1.8.

```
nestingTB
  ^ SpecColumnLayout composed
    newRow: [ :row | row add: #list] top: 0 bottom: 0.2 ;
    newRow: [ :row |
      row
        newColumn: [:c | c add: #text] left: 0 right: 0.45;
        newColumn: [ :c |
          c newRow: [ :inRow |
            inRow
              add: #button;
              add: #button2]] left: 0.55 right: 0
    ] top: 0.8 bottom: 0 ;
```

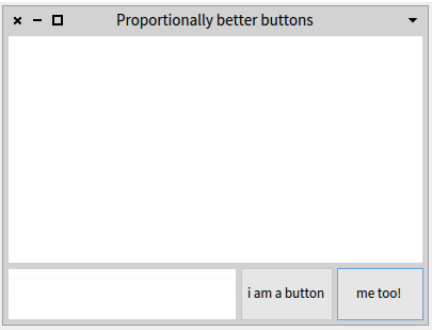

Figure 1.8 Screen shot of a use of proportional rows and columns.

```
    yourself
```

1.5 Layouts without rows or columns

Rows and columns are of course not the only way in which a user interface can be laid out. Spec also allows widgets to be placed more freely. This can be done either using absolute positions of the enclosing container or using relative positioning, which takes into account window resizing. We show here how these different placement options can be used.

Absolute widget positions

A first manner in which widgets can be laid out, is by giving them absolute positions in their enclosing container. To do this, the `SpecLayout` method `add:top:bottom:left:right:` is used. It takes four extra arguments, each representing a distance as a number of pixels. The number of pixels should be positive, as it indicates a distance from the given edge towards the opposite edge.

For example, below we perform a layout of a button at 10 pixels from the top, 200 pixels from the bottom, 10 from the left and 10 from the right, and the result is shown in Figure 1.9.

```
LayoutExample >> oneButtonAbsolute
  ^ SpecLayout composed
    add: #button top: 10 bottom: 200 left: 10 right: 10;
    yourself
```

> **Note** The underlying logic of the arguments is the same as in `newRow:top:bottom:` and `newColumn:left:right` of Section 1.4: distances are towards the 'other end' of the container.

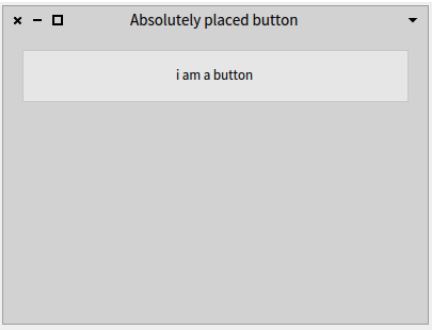

Figure 1.9 Screen shot of an absolutely placed button

> **Note** Obviously, the use of absolute widget positions breaks completely if a window is resized. It is therefore best to use this only in combination with windows that cannot be resized.

Relative widget positions

Relative widget positions cause the widget to be resized according to how its container is resized. The method `add: origin: corner:` of `SpecLayout` specifies such relative layout of a widget, percentage-wise from the origin point to the corner point. These two points represent respectively the top left corner and the bottom right corner of the widget. Since the arguments express a percentage of the container, they must be between 0@0 and 1@1.

For example, below we place one button that is half the container size in the center of the container, which can be seen in Figure 1.10.

```
LayoutExample >> oneButtonSmaller
  ^ SpecLayout composed
    add: #button origin: (0.25 @ 0.25) corner: (0.75 @ 0.75);
    yourself
```

Alternatively, the method `add:top:bottom:left:right:` can also be used with percentage arguments, i.e., between 0 and 1, to produce a relative layout with percentages computed from the opposite edge. For example, the code below produces exactly the same result as the code above. We however discourage this use of the `add:top:bottom:left:right:` method, as there may be confusion with positions of 0 or 1 pixels (is it a relative or an absolute position?) and it does not support the use of offsets, which we discuss next.

```
LayoutExample >> oneButtonSmallerAlternative
  ^ SpecLayout composed
    add: #button top: 0.25 bottom: 0.25 left: 0.25 right: 0.25;
    yourself
```

Figure 1.10 Screen shot of an always centered button

Relatives with offsets

Widget relative positions have the advantage that they resize widgets as their container grows and shrinks, but have the disadvantage that widgets are always placed right next to each other or next to the container border. Placing a widget at, e.g., five percent of a container border, can make the border of the correct thickness for a specific window size but then too big or too small when the window is resized. Actually, what is needed is for a way in which we can place widgets relatively, but specify absolute offsets in addition, so that the gaps between widgets will always be the same. This is possible with the `add: origin: corner: offsetOrigin: offsetCorner:` method of `SpecLayout`. The `origin: corner:` arguments are the same as in the `add: origin: corner:`method and both offset arguments take a point as argument. These points express the number of pixels from the corresponding corner, in the classical computer graphics coordinate system where the origin is in the top left corner. Consequently, the `x` or `y` component can be negative representing an offset towards the left, respectively the top.

For example, the code below lays out two buttons on top of each other. Each of them takes half the space of the window, minus the window border of 10 pixels and a the space between them of 10 pixels.

```
LayoutExample >> twoButtonsRelativeOffset
  ^ SpecLayout composed
    add: #button origin: (0 @ 0) corner: (1 @ 0.5)
         offsetOrigin: (10 @ 10) offsetCorner: (-10 @ -5);
    add: #button2 origin: (0 @ 0.5 ) corner: (1 @ 1)
         offsetOrigin: (10 @ 5) offsetCorner: (-10 @ -10);
    yourself
```

Relative widget positions for rows and columns

Lastly, for easy composition of rows and columns with other widgets, both rows and columns can be placed relatively as well as relatively with an offset. The methods `newRow:` and `newColumn:` have their variants with suffix `origin: corner:` and `origin: corner: offsetOrigin: offsetCorner:`. These add the relative layout options we have seen for widgets to rows and columns.

One example of the use of this placement was in the very first chapter, in the customer satisfaction example. The layout code for this example is below, and the result can be seen in Figure **??**.

```
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
```

1.6 Conclusion

In this chapter we have discussed the various layout strategies that Spec provides. If your user interface has more than one widget, you will need to use one of these strategies. We talked about two broad strategies of layouts: firstly row and column based layouts and secondly absolute and relative layouts (and their combination). It is prudent to know the advantages and disadvantages of each layout when constructing your UI so the choice of layout that is used corresponds to a good tradeoff considering the visual appearance of the user interface.