Adding a Widget

So far in this course, we have considered how to build applications with user interface components from various sources: a widget provided by the Palette; reusing Canvasses using the SubCanvas widget; and embedding a custom view using the ViewHolder widget. In this module, we provide a basic introduction to one other source: a locally–provided widget added to the Palette. Since the mechanism by which a new widget is created is extremely complex, this module provides an overview of creating a new widget by way of a worked example.

1. What is a Widget?

Each widget is represented by two classes. One class describes the properties of the widget, the other provides the widget's appearance on a Canvas.

For example, if you add a Slider widget to a Canvas and specify its properties via the Properties Tool, you are interacting with an instance of SliderSpec — it contains attributes to describe the properties of the widget as it is displayed on the screen. These properties include: layout, name, model, colors, orientation, and the start, stop and step values for the slider.

The display of the widget in the Canvas is provided by an instance of a platform–specific subclass of SliderView (e.g. MacSliderView). It is this instance that is responsible for displaying an appropriate representation of the slider's model in a Canvas. The specific subclass of SliderView that is instantiated is determined by the current look–and–feel policy. (See "Look and Feel" on page 9 of the "Coding for Multiple Platforms" module.)

As all widgets share some of the properties and behavior of SliderSpec, there is (as you might imagine) a class hierarchy to represent the commonality of behavior and state. Class SliderSpec's superclass hierarchy is as follows:

```
Object
    UISpecification
        ComponentSpec
            NamedSpec
                WidgetSpec
                    SliderSpec
```

Class UISpecification provides an abstract superclass for all forms of specifications, including window specifications, composite specifications (i.e. groups) and component specifications (i.e. widgets). The specifications are used by an instance of UIBuilder to determine the appearance and features of the widget it is building.

Class ComponentSpec provides an abstract superclass for component specifications (i.e. widgets). In particular, it introduces the layout property for a widget (an instance of Point, Rectangle, LayoutOrigin or LayoutFrame) to describe the placement of the widget.

Class NamedSpec provides an abstract superclass for those component specifications that have the following properties:

- name — appears as *ID* in the Properties Tool.
- decoration — describes the border decoration and presence of scroll bars.
- opacity — indicates whether or not the widget should have a background color.
- color — described using the **Color** page of the Properties Tool.
- drop source and drop target — see the “Drag and Drop” module for more details.

Class WidgetSpec provides an abstract class for those components that require a model. Additionally, it includes properties for Notification and Validation (called “callbacks”). Finally, it also introduces a property of “tabability”, which describes whether or not the component can receive the keyboard focus as a result of the user pressing the <tab> key.

The process of adding a widget to the Palette requires the following steps:

1. Create a class to specify the properties of the widget.
2. Create a class to provide the appearance of the widget on a Canvas. (If there are significant differences in appearance for each GUI platform, it may be desirable to produce an abstract class with platform–specific subclasses — one for each GUI platform.)
3. Add a method to class UILookPolicy (or its subclasses) that use an instance of the class produce in step 1 to create an instance of the class produced in step 2.

The remainder of this module provides a worked example of adding a widget to the Palette. The widget will represent a simple “Gauge” — a circular representation of a numeric value. It provides a similar interface to a Slider widget, since (like a slider) its *value* is a number between 0 and 1, describing the sweep angle of a wedge. However, unlike a Slider widget, a Gauge is read–only. Let us go through the steps:

2. Create a “Spec” Class

2.1. Class Definition

The Gauge widget requires a model, so we could create a class called GaugeSpec as a subclass of WidgetSpec, as it provides access to a model. However, the Gauge will have similar functionality to a Slider, so we will create it as a subclass of SliderSpec, thus:

```
SliderSpec subclass: #GaugeSpec
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Adding a Widget'
```

2.2. Installing the Spec

We can now add the class to the Palette, using the following message expression:

```
(UIPalette activeSpecsList includes: #GaugeSpec)
    ifFalse: [UIPalette activeSpecsList add: #GaugeSpec]
```

(It's a good idea to put this expression in a class method such as initialize, so that when producing the file–in the Spec will be automatically added to the Palette.)

If you wish to remove the widget, use the following expression:

```
UIPalette activeSpecsList remove: #GaugeSpec ifAbsent: []
```

2.3. Palette Icons

If you now open a new Canvas, we will see that the Palette contains an extra icon, but unfortunately it is the same as that for the Slider widget. The icon is provided by one of two class methods called paletteIcon and paletteMonoIcon; the latter is used when displaying on a monochrome screen.

Open the Image Editor a install two Image resources in class GaugeSpec, corresponding to the methods above. An example Image is shown in Fig.1.

2.4. Component Name

Finally, before the Palette used successfully we have to give the class a *component name* — the String that is displayed in the Palette when a widget is selected. A widget's component name is determined by sending the message componentName to its specification class. So, add the following class method:

```
componentName
    ^'Gauge'
```

2.5. Generating a Widget

If you now select the Gauge widget from the palette and place it on the Canvas, you will see that it appears as a Slider. This is because we have to provide a *specGenerationBlock* for the class. This block is evaluated by the Canvas controller to produce a new instance of the class that provides the widget's appearance. Unless we provide one for class GaugeSpec, it will use the one inherited from SliderSpec. To remedy this, copy the specGenerationBlock method from ComponentSpec class and add it to GaugeSpec class.

2.6. Placement

You might have noticed that the method above consists of two major message expressions:

1. a message to create a new instance of the receiver (layout:extent:)
2. a message to specify the placement of the new widget (placementExtentFor:inBuilder:)

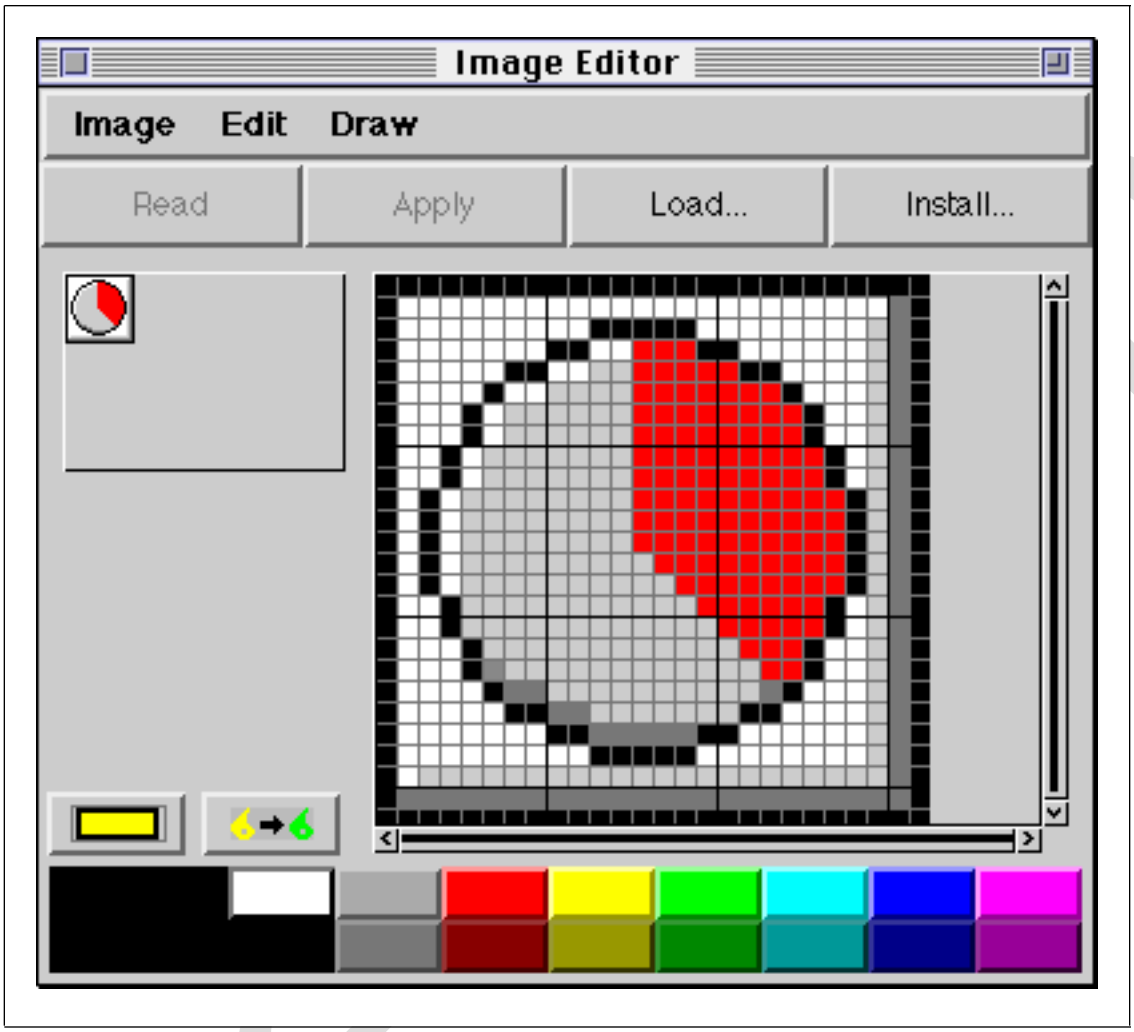

Figure 1: An Icon for the Gauge Widget

The second of these expressions in turn causes the message placementExtentBlock to be sent to the receiver. It is expected to return a one–argument block that, when evaluated, will return the default extent of the widget. At present, GaugeSpec inherits the placementExtentBlock method from its superclass. Rather than use that implementation, add the method to GaugeSpec class, thus:

```
placementExtentBlock

    ^[:bldr |  100 @ 100]
```

2.7. Properties

Having placed a Gauge widget on the Canvas, we can now give it some properties using the Properties Tool. The first thing to notice when opening the Properties Tool is that we don't need all those pages: we only need **Basic**, **Details**, **Color** and **Position**. The pages are controlled by the class method slices, which returns a literal Array of pages. Copy the slices method from SliderSpec class and modify it accordingly.

If you look closely at the **Basic** page of the Properties Tool, you will see that the page's title is incorrect. To remedy this, copy the method named basicsEditSpec from SliderSpec class, modify the page's title and install it in GaugeSpec class.

The Details page of the Properties Tool contains two unnecessary properties: the gauge has no orientation, and it is always disabled (since it's read–only). Copy the detailsEditSpec method from SliderSpec class, remove the unnecessary widgets and install it in GaugeSpec class.

3. Create a "View" Class

Now we need to create a class to represent the appearance of the widget on a Canvas. Let's call it GaugeView.

Class SimpleView is an abstract class that acts as the superclass for almost all classes that represent the appearance of widgets. Class SliderView is a direct subclass of SimpleView, and it's to this class that we can turn to discover how to create our new GaugeView class. We'll start by defining GaugeView as a subclass of SimpleView, thus:

```
SimpleView subclass: #GaugeView
    instanceVariableNames: 'rangeMap '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Adding a Widget'
```

The instance variable rangeMap will be used to reference an instance of RangeAdaptor — an instance of which represents a start, stop and step in a range. Firstly, you need to add an accessing method, thus:

```
rangeMap: aRangeMap

    rangeMap := aRangeMap
```

The sole responsibility of this class is to provide a display of its model. The method that implements this behavior is called displayOn:. We'll divide out the responsibility for displaying the gauge into two methods: one that displays the background and outer edge of the Gauge; and another that displays the wedge corresponding to the value of its model. The method is as follows:

```
displayOn: aGraphicsContext
    self displayFaceOn: aGraphicsContext.
    self displayRegionOn: aGraphicsContext
```

The first of these methods is fairly straightforward: it should display its background color (if it has one), before drawing a complete arc within the bounds of its widget; thus:

```
displayFaceOn: aGraphicsContext
    self displayBackgroundIfNeededOn: aGraphicsContext in: self bounds.
    aGraphicsContext paint: self foregroundColor.
    aGraphicsContext lineWidth: 2.
    aGraphicsContext
        displayArcBoundedBy: self bounds
        startAngle: 0
        sweepAngle: 360
```

The second method is slightly more complex: it needs to determine its *value* (a number between 0 and 1) by asking its model. For example, if you examine the computeMarker method in class SliderView, you will see that it gets its *value* (the temporary variable val) either by interrogating its model or its rangeMap instance variable. We need a similar sequence of message expressions in our method, so copy that piece of code into the displayRegionOn: method. This value is then used to specify the sweep angle of the wedge, thus:

```
displayRegionOn: aGraphicsContext
    | degrees value |
    value := rangeMap == nil
                ifFalse: [rangeMap map: model value]
                ifTrue: [model value].
    degrees := (value * 360) truncated.
    aGraphicsContext paint: self selectionForegroundColor.
    aGraphicsContext
        displayWedgeBoundedBy: self bounds
        startAngle: 0
        sweepAngle: degrees
```

4. Building the Right View

So far we have created a class to represent the properties of the Gauge widget, and a class to provide its appearance. As the last step in the process, we have to connect them together.

An instance of UIBuilder is responsible for building a Canvas and its widgets. It builds each widget by sending its specification the message dispatchTo:with:. Currently, GaugeSpec inherits that method from class SliderSpec — the method sends the message selector slider:into: to the UIBuilder's policy object (the object that represents the current look–and–feel). For GaugeSpec to work properly, it must implement its own dispatchTo:with: method. Add a dispatchTo:with: method to class GaugeSpec as follows:

```
dispatchTo: policy with: builder

    policy gauge: self into: builder
```

The message selector gauge:into: is sent to an instance of some subclass of UILookPolicy. We're going to implement the method corresponding to that message selector in class UILookPolicy so that it is inherited by all its subclasses. We could also implement it (in different forms) in each of the subclasses of UILookPolicy if we wished the appearance of the Gauge widget to vary according to platform.

As an example of a method implemented in class UILookPolicy, consider the slider:into: method. It's very close to what we need — modify it by changing the method selector to gauge:into: and by editing the method body so that appears as below.

```
gauge: spec into: builder
    | component model rangeMap |
    model := spec modelInBuilder: builder.
    rangeMap := spec rangeMap.
    component := GaugeView model: model.
    component rangeMap: rangeMap.
    component widgetState isEnabled: spec initiallyEnabled.
    builder isEditing ifFalse: [component widgetState isVisible: spec initiallyVisible].
    builder component: component.
    self
        setDispatcherOf: component
        fromSpec: spec
        builder: builder.
    builder wrapWith: self borderedWrapperClass new.
    builder wrapper border: BeveledBorder inset.
    builder wrapper inset: 0.
    builder applyLayout: spec layout.
    builder wrapWith: (self simpleWidgetWrapperOn: builder spec: spec)
```

5. Using the Widget

Create a new Canvas and add a Gauge widget from the Palette. If you resize the widget, you may find that an error is produced — this is because class GaugeSpec is inheriting the method dragHandlesFor:subject:isPrimary: from class SliderSpec. This method returns an Array of DragHandle objects for use by the controller of the Canvas specification window. As you may have noticed, a Slider widget changes its orientation as its extent is manipulated by the user — an effect provided by the DragHandle objects. We don't want this behavior for GaugeSpec, so copy the corresponding method from class ComponentSpec to class GaugeSpec.

6. Testing the Widget

Create a new Canvas and add a Gauge widget and a Slider widget. Give both widgets the same properties, so that they share the same model. Install the Canvas and Define the model. Now, when you open the application you should find that as the slider is moved the Gauge re–displays itself.

Ex 1. Modify class GaugeView so that the start angle of the wedge is at “12 o’clock”.

Ex 2. Apply some color properties to the Gauge widget using the Properties Tool.

Ex 3. Modify the gauge:into: method so that, when painted, the Gauge widget appears more like its Palette icon.

Ex 4. Modify class GaugeView so that it flashes less when re–displaying.