Application Note:
TLT-0664-AN-MICROEJ-MWTTutorial

MWT Tutorial

In relation to: MICROEJ products

Features

This Application Note explains all aspects of MWT library.

Description

This Application Note assumes the reader wishes to get a quick understanding of the MWT library concepts.

Table of Contents

List of Figures

1 Introduction

1.1 What is MWT?

Widgets are the visual elements of graphical user interfaces. Widgets exist at the point of interaction between the user and the machine. Common types of widget include radio buttons, lists and text fields.

A widget library comprises classes that can be instantiated directly to create widgets in a user interface or extended to define new kinds of widgets.

The MicroUI Widget Toolkit (MWT), the subject of this tutorial, makes it simple to create and use widget libraries. It includes an abstract superclass, Widget, that can be extended to define widget types, plus other ready-to-use classes that act as widget containers. It also contains a framework that controls the operation of the widgets and their interaction with users. However, MWT is not a widget library – it contains no widget classes that can be used directly.

MWT targets pixelated display devices in resource-constrained environments, such as embedded controllers.

MWT is defined by a specification, the Micro Widget Toolkit Profile Specification. This tutorial assumes the use of version 1.0 of MWT and IS2T's implementation of the specification.

1.2 About this tutorial

This tutorial is designed to provide a self-taught introduction to MWT. If you are new to MWT we recommend that you work through the examples, taking time to study, run, and perhaps modify, the supplied source code. Each example introduces new concepts and builds on previous ones. More experienced MWT users may wish to be more selective, and focus on the topic they wish to learn about.

The tutorial examples gradually build up a small widget library that is used consistently throughout. Readers may find this library useful as the basis for their own work.

1.3 Roles

In this tutorial we will consider MWT from three different points of view. We will show how to create a widget library, looking separately at the functional and look-and-feel aspects. We will show how to use a widget library built using MWT. These points of view correspond to the three roles defined in MWT:

- The Widget Designer – the person who defines the types of widgets that make up the widget library, and how they work.
- The Look And Feel Designer – the person who defines how widgets and containers appear on the display.
- The Application Designer – the person who uses an MWT-based widget library to create a graphical user interface.

Obviously, these three roles are not always played by different people; the last two in particular will often be undertaken by the same people.

1.4 Before you start

Before you start working through this tutorial you should install:

- MicroEJ product
- A MicroEJ Java Platform that provides MWT.
- The tutorial examples project available on MWTTutorial-apps.zip archive file.

The tutorial assumes you already know:

- How to use Eclipse and MicroEJ.
- How to write simple programs using Java.
- How to create and run a MicroEJ application

The MicroEJ Workbench User's Manual includes step-by-step guidance on creating a simple application.

2 A Simple Widget

We will start by adopting the “Widget Designer” role, and we will create a very simple widget. We want to be able to test our widget, so we need to understand how to construct a simple MWT-based user interface.

At its most basic, an MWT-based user interface is created by arranging some widgets in a Panel, and showing that panel on a Desktop, which is itself shown on a display. Panel and Desktop are classes within MWT. We don't arrange the widgets directly on the desktop because an application may want to be able to create a number of different panels and switch between them quickly. In our first examples the panel will contain a single widget, but by using composites, which we will discuss later, the panel can be the root of a complex hierarchy of widgets.

2.1 Example 1

It's time to create our first MWT application. Open the class `Ex1` in the package `com.is2t.mwt.tutorial.ex01`:

```
package com.is2t.mwt.tutorial.ex01;

import ej.microui.MicroUI;
import ej.mwt.Desktop;
import ej.mwt.MWT;
import ej.mwt.Panel;

public class Ex1 {

 public static void main(String[] args) {
  MicroUI.errorLog(true);
  MWT.RenderingContext.add(new TutorialTheme());
  Desktop desktop = new Desktop();
  Panel panel = new Panel();
  Label label = new Label("Hello World");
  panel.setWidget(label);
  panel.show(desktop);
  desktop.show();
 }
}
```

This example displays a label. Create a launch configuration for the example and run it using the simulator. You should see something like this:

Figure 2.1. Example 1 executed on simulator

The example depends on three other classes that are in the same package. We will look at each of these in turn, but first we will examine Ex1 line by line:

- `MicroUI.errorLog(true)` turns on error logging so that you can see any errors in the Eclipse console. This line isn't essential but we will include it in all our examples to help with debugging.
- `MWT.RenderingContext.add(new TutorialTheme())` establishes the theme to be used. A theme defines a look-and-feel. When you create an MWT widget library you can define several themes that can be applied to the widgets in the library.
- `Desktop desktop = new Desktop()` creates a Desktop bound to the default display device.
- `Panel panel = new Panel()` creates a Panel. A panel is analogous to a window – there can be several on them on the desktop, but only one is active.
- `Label label = new Label("Hello World")` creates a Label widget and sets its text. Label is one of the three other classes that make up the example.
- `panel.setWidget(label)` connects the label widget to the panel. A panel can hold only one widget (but that widget can be a composite, as we will see later).
- `panel.show(desktop)` instructs MWT to show the panel on the desktop. If there were already other panels on the desktop it would be added to the top of the stack of panels being shown, becoming the active panel. This step also causes the panel to lay out its contents – simple in this case.
- `desktop.show()` instructs the desktop to show itself on the display.

2.2 Label

The `Label` class defines a kind of widget, using the MWT Widget class as its base.

```
package com.is2t.mwt.tutorial.ex01;

import ej.mwt.Widget;

public class Label extends Widget {

 protected String text;

 public Label(String text) {
  super();
  this.text = text;
 }

 public String getText() {
  return text;
 }
}
```

This very simple widget holds a string and provides a getter method for it. You will immediately notice that this class does not define how the label is to be rendered on the display. That is the job of a renderer.

2.3 LabelRenderer

A renderer is an object that can render widgets of a specific type or types. It holds no widget-related state: the widget is always passed to it as a parameter. As a result a single renderer object can be used to render many widgets.

```
package com.is2t.mwt.tutorial.ex01;

import ej.microui.Colors;
import ej.microui.io.GraphicsContext;
import ej.mwt.Renderable;
import ej.mwt.Widget;
import ej.mwt.rendering.WidgetRenderer;

public class LabelRenderer extends WidgetRenderer {

 public Class getManagedType() {
  return Label.class;
 }

 public int getPreferredContentWidth(Widget widget) {
  return 100;
 }

 public int getPreferredContentHeight(Widget widget) {
  return 16;
 }

 public void render(GraphicsContext g, Renderable renderable) {
  Label label = (Label)renderable;
  g.setColor(Colors.WHITE);
  g.fillRect(0, 0, label.getWidth(), label.getHeight());
  g.setColor(Colors.BLACK);
  g.drawString(label.getText(), 0, 0, GraphicsContext.TOP | GraphicsContext.LEFT);
 }
}
```

A widget renderer class must extend the `WidgetRenderer` base-class. The `LabelRenderer` in this example implements four methods which must be implemented by every widget renderer class:

- `getManagedType()` returns the widget class that this renderer can render. Implicitly it can also render widgets that are instances of sub-classes of the managed type. The MWT framework selects the most appropriate renderer for each widget.

- `getPreferredContentWidth()` and `getPreferredContentHeight()` allow the renderer to indicate how much screen space it would like to occupy (in pixels). Here we have hard coded some values but more commonly the renderer will compute the required space based on the widget content. In the case of a label the computation might be based on the label's string. You can see that sort of computation in the next example.

- `render()` is called by the MWT framework to ask the renderer to render the widget on the display. The first parameter is the `GraphicsContext` that the renderer should use to do its painting. The origin of the graphics context is set to the top-left corner of the space allocated to the widget, and the graphics context will clip any attempt to paint outside the bounds of the widget.

In this simple example the colors used by the renderer are hard-coded. In realistic examples the colors would instead be defined by the Look associated with the selected `Theme`.

2.4 TutorialTheme

The MWT framework needs a way of knowing what renderers are available. It does this by having a `RenderingContext` that holds one or more Themes. Each `Theme` holds a coherent set of renderers and a `Look`, which defines visual properties such as colors.

You will recall that the main Ex1 class contained the line: `MWT.RenderingContext.add(new TutorialTheme());`

This statement creates an instance of TutorialTheme and makes it available to MWT's rendering context. Here is the trivial theme we are using for this example:

```
package com.is2t.mwt.tutorial.ex01;

import ej.mwt.rendering.Look;
import ej.mwt.rendering.Theme;

public class TutorialTheme extends Theme {

 public Look getDefaultLook() {
  return null;
 }

 public String getName() {
  return "Tutorial theme";
 }

 public boolean isStandard() {
  return true;
 }

 protected void populate() {
  add(new LabelRenderer());
 }

}
```

This tutorial theme does not define a look (`getDefaultLook()` returns null), but a theme would normally do so. We will return to `Look` in a later example, when we will also discuss the significance of the `isStandard` method.

The `populate` method is called by the rendering context when the theme is added to it, and it must add to the theme an instance of each kind of renderer that makes up the theme. These instances are then available to MWT to use for rendering. Our theme has only one renderer at present: the `LabelRenderer`. Whenever the rendering context needs to find a renderer to use to render a widget it looks through the themes it has available and selects the best possible renderer. The exact algorithm used for that will be discussed in a later example.

3 Adding Behavior

In Example 2, we will create a simple button widget that can be pressed using the pointer (the mouse, when using the simulator).

3.1 Example 2

The example can be found in `com.is2t.mwt.tutorial.ex02.Ex2`:

```
public class Ex2 {

 public static void main(String[] args) {
  MicroUI.errorLog(true);
  MWT.RenderingContext.add(new TutorialTheme());
  Desktop desktop = new Desktop();
  Panel panel = new Panel();
  Button button = new Button("Press Me");
  button.setListener(new Listener() {
   public void performAction() {
    System.out.println("Button pressed");
   }
   public void performAction(int value) {}
   public void performAction(int value, Object object) {}
  });
  panel.setWidget(button);
  panel.show(desktop);
  desktop.show();
 }
}
```

This is very similar to the previous example except that we are creating a `Button` widget rather than a `Label`. The interesting part is the call to setListener, which sets up a callback that is invoked when the button is pressed. `Listener` is an interface commonly used in MWT and MicroUI to define these kinds of notifications. When the button is pressed it will invoke the `performAction` method and the message will appear on the Eclipse console. Try running Ex2 and clicking on the button.

3.2 Button

The `Button` class extends the `Label` class we created in Example 1:

```
public class Button extends Label {

	...

	public boolean handleEvent(int event) {
		int type = Event.getType(event);
		if (type == Event.POINTER) {
			int action = Pointer.getAction(event);
			if (action == Pointer.RELEASED) {
				if (listener != null)
					listener.performAction();
				return true;
			}
		}
		if(type == Event.COMMAND) {
			int command = Event.getData(event);
			if (command == Command.SELECT) {
				if (listener != null)
					listener.performAction();
				return true;
			}
		}
		return super.handleEvent(event);
	}
}
```

Figure 3.1. Button source code

As we have already seen, the `setListener` method is used by the application to hook up the callback. A reference to the listener is held in a field of the button.

The `handleEvent` method is called by the MWT framework when an event occurs that may be of interest to the widget. Our button will receive several different events but here we are looking specifically for the pointer button being released. When that happens we notify the listener, if there is one, and return `true`, indicating that we have consumed the event. For all other events we let the superclass handle it. The `Pointer` class contains constants for the possible events, and were are looking for `Pointer.RELEASED` which will be sent when the button is released.

Generally speaking, input events are sent to the widget that has the focus. The widget that is under the pointer when the mouse button is pressed is given focus. That widget then retains focus until another widget is similarly selected. That is why the `Pointer.RELEASED` event is sent to the widget that was under the pointer when the button was pressed, even if the pointer is moved outside the widget. Input focus can also be changed in other ways, such as by using a joystick or cursor keys, we will see it in later examples.

The general pattern for event handling is that the event is sent to the most specific widget that might be interested in it. That widget will either consume it or not. If it is not consumed the event is sent to the widget's parent (which in this case is the panel) and so on until either the event is consumed or the top of the desktop hierarchy is reached.

3.3 ButtonRenderer

The ButtonRenderer is very similar to the `LabelRenderer` from Example 1:

```
public class ButtonRenderer extends Renderer implements SizeComputer {

   ...

   public void render(GraphicsContext g, Renderable renderable) {
         Button button = (Button) renderable;
         g.setColor(Colors.WHITE);
         g.fillRect(0, 0, button.getWidth(), button.getHeight());
         g.setColor(Colors.BLACK);
         g.drawString(button.getText(), button.getWidth() / 2, button.getHeight() / 2,
                     GraphicsContext.HCENTER | GraphicsContext.VCENTER);
         if (button.hasFocus()) {
               g.setStrokeStyle(GraphicsContext.DOTTED);
               g.drawRect(1, 1, button.getWidth()-3, button.getHeight()-3);
         }
   }

   public int getPadding() {
         return 4;
   }
```

Figure 3.2. ButtonRenderer source code

Notice the more sophisticated (and accurate!) calculation of the preferred width and height, based on the size of the font to be used and the text to be displayed. Notice also that the text is drawn in the center of the widget, rather than the top-left.

4 Commands

In the previous example we used the pointer to press the button. But often we want to use other input devices, such as physical push-buttons or a keypad to “press” the button. This is achieved using command events.

The MicroUI library, on which MWT is based, converts low-level inputs into abstract commands. So, for example, the center joystick button on the STM3220G-EVAL is normally configured so that it generates the `Command.SELECT` command. In Example 3 we will enhance our button widget so that it responds to this command.

Commands such as `Command.SELECT` are sent as events by MWT to whichever widget has the focus, as discussed in the previous exercise.

4.1 Button Enhancements

To handle command events we simply need to enhance the `handleEvent` method in the `Button` class:

```
public class Button extends Label {

 ...

 public boolean handleEvent(int event) {
  int type = Event.getType(event);
  if (type == Event.POINTER) {
   int action = Pointer.getAction(event);
   if (action == Pointer.RELEASED) {
    if (listener != null)
     listener.performAction();
    return true;
   }
  }
  if(type == Event.COMMAND) {
   int command = Event.getData(event);
   if (command == Command.SELECT) {
    if (listener != null)
     listener.performAction();
    return true;
   }
  }
  return super.handleEvent(event);
 }
}
```

4.2 ButtonRenderer Enhancements

It is normal to indicate to the user which widget has focus by means of a visual clue. We can enhance the `ButtonRenderer` to do this:

```
public class ButtonRenderer extends Renderer implements SizeComputer {

 ...

 public void render(GraphicsContext g, Renderable renderable) {
  Button button = (Button) renderable;
  g.setColor(Colors.WHITE);
  g.fillRect(0, 0, button.getWidth(), button.getHeight());
  g.setColor(Colors.BLACK);
  g.drawString(button.getText(), button.getWidth() / 2, button.getHeight() / 2,
    GraphicsContext.HCENTER | GraphicsContext.VCENTER);
  if (button.hasFocus()) {
   g.setStrokeStyle(GraphicsContext.DOTTED);
   g.drawRect(1, 1, button.getWidth()-3, button.getHeight()-3);
  }
 }

 public int getPadding() {
  return 4;
 }
```

The render method asks the button if it has focus using the `hasFocus` method. If it does, the renderer indicates this by drawing a dashed rectangle just inside the bounds of the widget. To leave room for this we need to allow some space around the text. Rather than just changing the size calculated by `getPreferredContentWidth` and `getPreferredContentHeight` we override the inherited `getPadding` method and return the number of pixels we want between the outer edge of the widget and its content (the text on the button).

4.3 Running Example 3

There are a couple of points to note when running Ex3.

You must execute the application on a platform which one of its input devices generates the SELECT command. To know which events are generated by a platform, an example named "Check Input Events" can be executed. This application logs received events during an input device usage.

When you first start the application no widget has focus, so pressing whatever button you have configured to generate SELECT will not “press” the button widget. You can see that this is the case because the button is not outlined on the display. First use the mouse pointer to press the button, which will give the button widget focus. Then you can use the SELECT command.

5 Widgets, Renderers and the MVC pattern

5.1 Model-View-Controller

The design of MWT is based on a long-established pattern for GUIs: the Model-View-Controller (MVC) pattern. MVC was first described by Trygve Reenskaug in 1979 and has been the basis for most GUI design approaches since. As the name implies, MVC revolves around three concepts, each with its own responsibilities, and their interactions.

The Model holds some data, or state, that needs to be represented in some way in the GUI. It knows nothing about how the state will be displayed or how the user will interact with the GUI to amend it. It does not even know whether there are any current representations of it in the GUI.

The View is responsible for rendering the state of the model.

The Controller is responsible for accepting user inputs, interpreting them, and using them to update the model. If has no knowledge of how the model is rendered in the View.

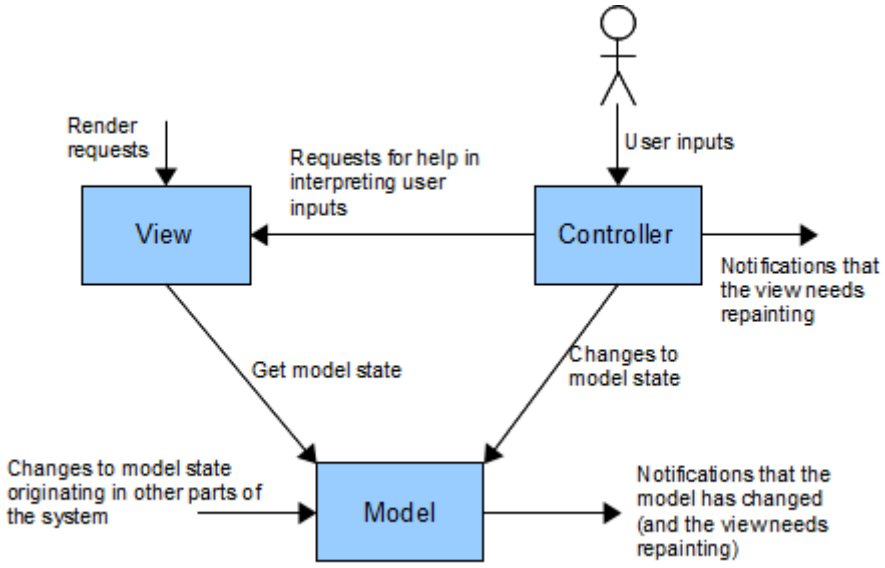

Figure 5.1. MVC pattern

In the diagram the arrows represent method calls. There are three main flows of control:

- The user makes an input that implies a change in the state of the Model. The Controller accepts the user input, deduces the required model change (possibly with help from the view), and calls a method of the Model to effect the change. The Model reacts to this by notifying the environment that its state has changed. The environment asks the View to re-render the Model.

- The user makes an input that implies a change in the View but not the Model, such as navigating from one sub-element of the View to another. In this case the Controller notifies the environment, and the environment asks the View to re-render itself.

- The Model is updated by some other part of the system (for example, a regularly recurring timer task).The Model notifies the environment that its state has changed. The environment asks the View to re-render the Model.

The notifications from the Model to the View are typically implemented using an observer (or listener) pattern, similar to the mechanism we used earlier to connect the button to the main application. An important feature of MVC is that the same Model can simultaneously be represented in the GUI by more

than one View, each with its own Controller. The Model neither knows nor cares how many Views are open on it. When its state changes the notification is passed to all the Views that depend on it.

There are two key benefits of the MVC approach:

- The domain logic (in the Model) is cleanly separated from the presentation logic (in the View and Controller).
- There is no direct coupling between the Model and the View/Controller, making it easy to have multiple simultaneous and different views of a model.

5.2 Widget-Renderer-Presenter

Now let us try to interpret MVC in the context of MWT. In the widgets we have been using so far, the widget has played the role of both the Controller and the Model. If has held the state (the text of the label or button, and implicitly a Boolean indicating whether or not the button is pressed) and handled user inputs. The renderer has played the role of the View. There has also been another concept in our examples: the Presenter (the Exn class). It is the Presenter that has been notified of changes in model state (e.g. when the button is pressed).

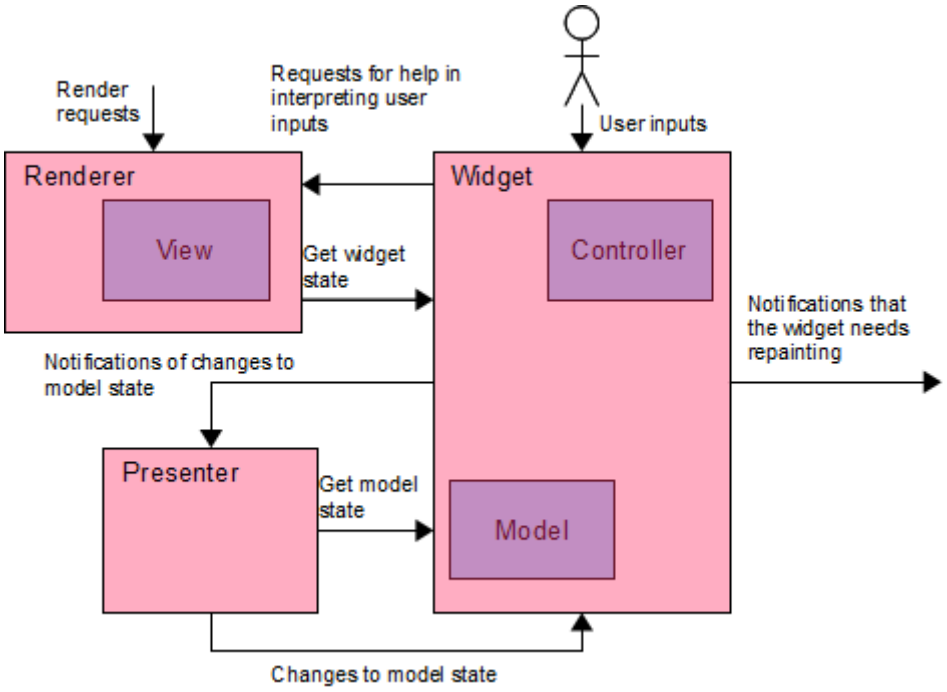

Figure 5.2. WRP pattern

This interpretation of MVC is found very frequently in widget-based architectures. The Presenter takes responsibility for assembling the GUI widgets needed for the application and uses notifications from the widgets to affect the state of the application.

MWT says nothing about how the Presenter should be designed but care is needed to ensure that the Presenter deals only with the GUI interactions and does not contain domain logic.

A disadvantage of this Widget-Renderer-Presenter approach is that the model is encapsulated in the widget. If several widgets need to display different aspects of the same model, with all the widgets updating together as the model updates, then this will need to be explicitly coordinated by the Presenter.

5.3 Widget-Renderer-Model-Presenter

A more flexible, but more complex, approach is to keep the model as a separate concept rather than encapsulating it in the widget.

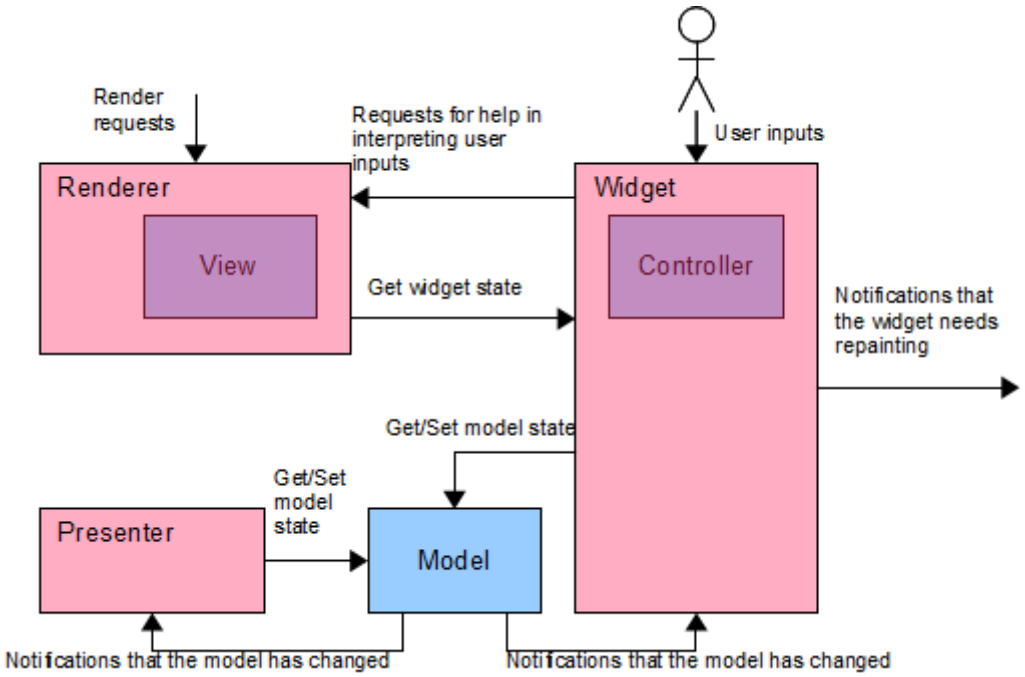

Figure 5.3. WRMP pattern

In this approach the same model can be attached to multiple widgets. More importantly, the model can contain domain logic. This is one way of dealing with the risk that domain logic will leak into the Presenter.

We will see this approach used in section "Using explicit model objects". It is common for widget libraries to contain some widget classes that use the Widget-Renderer-Presenter approach and others that use this Widget-Renderer-Model-Presenter approach.

6 Redisplaying a widget

So far in our examples we have not made any updates to the display (other than showing changes in focus, which happened automatically). In the next example we will create a `ToggleButton` whose model is a Boolean and where each press of the button toggles the button's state. We want the button to be rendered in a way that shows whether the button is currently on or off, so we will need to ensure the button is repainted each time it is pressed. In addition, we will run an asynchronous timer task that periodically toggles the button.

The example illustrates the two control flows that result in repainting:

- There is a user input (button press in this case) that causes the widget to decide that a repaint is required,
- Some other part of the system (a timer task in this case) updates the state of the widget.

The normal practice when designing MWT widgets is for the widgets to take responsibility for getting themselves redisplayed, rather than requiring the Presenter or some other object to do that. The widget is in the best position to understand the nature of the change and take the most appropriate action.

6.1 Repainting

Here is the code of the example:

```
public class Ex4 {

	public static void main(String[] args) {
		MicroUI.errorLog(true);
		MWT.RenderingContext.add(new TutorialTheme());
		Desktop desktop = new Desktop();
		Panel panel = new Panel();
		final ToggleButton button = new ToggleButton("Press Me");
		button.setListener(new Listener() {
			public void performAction(int value) {
				System.out.println("Button is " + (value==0?"off":"on"));
			}
			public void performAction() {}
			public void performAction(int value, Object object) {}
		});
		panel.setWidget(button);
		panel.show(desktop);
		desktop.show();
		TimerTask task = new TimerTask() {
			public void run() {
				button.setOn(!button.isOn());
			}
		};
		new Timer().schedule(task , 3000, 3000);
	}
}
```

Figure 6.1. Example 4 source code

We are now using the one-argument version of `performAction` to communicate from the button to the Presenter (the Ex4 class) so that we can pass the current state of the button, using zero for off and non-zero for on. The highlighted statement in the timer task is toggling the state of the button every three seconds.

The `ToggleButton` class looks like this:

```
public class ToggleButton extends Label {

	private Listener listener;
	private boolean isOn;

	public ToggleButton(String text) {
		super(text);
		isOn = false;
	}

	public void setListener(Listener listener) {
		this.listener = listener;
	}

	public boolean isOn() {
		return isOn;
	}

	public void setOn(boolean isOn) {
		this.isOn = isOn;
		repaint();
	}
```

Figure 6.2. ToggleButton source code

The key line here is the one highlighted. A call to `repaint()` schedules an asynchronous repainting of the widget receiving the call, and only that widget. The repaint request is added to an event queue and when executed will result in the widget's renderer receiving a `render` request.

A call to `repaint` does not cause the panel will not be laid out again (what MWT calls revalidation), so the widget's size remains unchanged. Therefore it is only appropriate to use `repaint` when the change to the widget is a change that will not affect its size. In the next example we will see how to handle updates that do change the size of the widget.

The renderer for the `ToggleButton` is very similar to the renderer we used for our previous buttons:

```
public class ToggleButtonRenderer extends WidgetRenderer {

      public Class getManagedType() {
            return ToggleButton.class;
      }

      public int getPreferredContentWidth(Widget widget) {
            ToggleButton button = (ToggleButton) widget;
            String text = button.getText();
            return DisplayFont.getDefaultFont().stringWidth(text);
      }

      public int getPreferredContentHeight(Widget widget) {
            return DisplayFont.getDefaultFont().getHeight();
      }

      public void render(GraphicsContext g, Renderable renderable) {
            ToggleButton button = (ToggleButton) renderable;
            g.setColor(button.isOn()?Colors.RED:Colors.WHITE);
            g.fillRect(0, 0, button.getWidth(), button.getHeight());
            g.setColor(Colors.BLACK);
            g.drawString(button.getText(), button.getWidth() / 2, button.getHeight() / 2,
                        GraphicsContext.HCENTER | GraphicsContext.VCENTER);
            if (button.hasFocus()) {
                  g.setStrokeStyle(GraphicsContext.DOTTED);
                  g.drawRect(1, 1, button.getWidth()-3, button.getHeight()-3);
            }
      }

      ...
```

Figure 6.3. ToggleButtonRenderer source code

(We have omitted the `getPadding` method to save space.) The highlighted line sets the background color of the button to reflect its state: red = on, white = off.

6.2 Revalidating

Sometimes the state of a widget will change in a way that requires the panel to be laid out again, usually because the size of the widget has changed. This layout action is called validation, and is initiated by calling `revalidate()` on the panel or any widget in it. This call will also cause the panel to be repainted after validation.

Revalidation can be an expensive operation: the sizes and positions of all the widgets on the panel must be calculated, and the entire panel must be repainted. Therefore we do not want to revalidate unless we need to. That is why revalidation is distinct from repainting.

Here is an example that shows revalidation:

```
public class Ex5 {

      public static void main(String[] args) {
            MicroUI.errorLog(true);
            MWT.RenderingContext.add(new TutorialTheme());
            Desktop desktop = new Desktop();
            Panel panel = new Panel();
            final ToggleButton button = new ToggleButton("Press me");
            button.setListener(new Listener() {
                  public void performAction(int value) {
                        boolean isOn = (value==0?false:true);
                        if (isOn) {
                              button.setText("Press me again to turn off");
                        } else {
                              button.setText("Press me");
                        }
                        System.out.println("Button is " + (isOn?"on":"off"));
                  }
                  public void performAction() {}
                  public void performAction(int value, Object object) {}
            });
            panel.setWidget(button);
            panel.show(desktop);
            desktop.show();
      }
}
```

Figure 6.4. Example 5 source code

We have added to our label widget a method that allows us to change the text of the label, and because `ToggleButton` inherits from Label we can use it for those, too. In this example we change the text of the button when the button is pressed. You will recall that the preferred size of the button is calculated from the text it must display.

Setting the text of the button is clearly a change that affects the size of the widget, so this requires revalidation rather than repainting. Here is the method we have added to `Label`:

```
      public void setText(String text) {
            this.text = text;
            revalidate();
      }
```

Figure 6.5. Label enhancement

The key line here is the one highlighted. A call to `revalidate()` schedules an asynchronous revalidation of the whole panel of which the widget is part. The revalidation request is added to an event queue and when executed will result in the validation process described in the next section.

Eagle-eyed readers may have noticed that we have added a `DesktopRenderer` to the theme in this example. Any `Renderable` object (which includes desktop, panel, and all widgets including composites) can have an associated renderer. In this example, when the button changes its size we need to ensure that the background is repainted to avoid leaving a mess. We can do this by including a simple renderer for the desktop that just fills the whole desktop area. Desktop and panel renderers extend the MWT class `Renderer`, rather than `WidgetRenderer`.

6.3 Packing

A panel has two important attributes: its size, and a flag that indicates whether it should be packed. In the examples so far we have not explicitly set either of these attributes. When a panel is constructed using its no-argument constructor, as we have done, its size is 0,0 and its packed flag is true.

The packed flag is used when the panel is validated (laid out). If the flag is true then after the panel has laid out its contents it is resized so that just big enough to hold its content. So although we haven't been able to see it, in all the examples so far the panel has been resized to fit its contents.

There is another panel constructor (Panel(int x, int y, int width, int height)) that allows the panel's size and position on the desktop to be specified. This constructor has the side-effect of setting the packed flag to false. So if you use this constructor the panel will not be resized when it is validated.

There is a short-cut if you want the panel to occupy the whole of the desktop:

```
...
panel.show(desktop, true);
...
```

With the second parameter set to true this will cause the size of the panel to be set to the size of the desktop and the packed flag set to false, as well as causing the panel to be shown on the desktop.

6.4 The validation process

As a widget designer it is important to understand how the process of validation works because more complex widgets will need to participate in it.

A call to revalidate() schedules an asynchronous revalidation of the whole panel. When the asynchronous layout is performed the validate() method of the panel is called. This:

- calls the `validate(int, int)` method of its child widget, passing in the available space,
- asks the child for its preferred size (which the child will have computed during its execution of `validate(int, int)`),
- sets the bounds of the child widget to the child's preferred size, taking into account the child's margin and its own padding,
- if it is packed, sets its own bounds to suit those of its child widget,
- forces a repaint of the whole panel.

The arguments passed to the validate call at step 1, called the `widthHint` and the `heightHint` respectively, depend on whether the panel has a defined size. If it has no defined size it will send 0 for both arguments. It has a defined size if:

- The panel's size was set on construction or subsequently and has not subsequently been set to be packed (using `setPacked(true)`), or
- The panel is implicitly set to the desktop size by using `show(desktop, true)`.

The `validate(int, int)` method of a widget must compute and store its preferred size. The default implementation in the Widget class determines the preferred size by calling its renderer's `getPreferredContentWidth` and `getPreferredContentHeight` methods, which we have seen in earlier examples, and adding on twice the defined padding. If the hint is 0 the computed size is stored, otherwise the stored value is the maximum of the computed size and the hint: the hints constrain the preferred size.

6.5 Summary, and impact on renderers

To summarize:

- If an attribute of a widget changes in a way that will not affect its size call `repaint()` on the widget. Only the area occupied by the widget will be repainted.
- If an attribute of a widget changes in a way that will affect its size call `revalidate()` on the widget or any object in the widget's panel hierarchy.

There are two important implications of this:

- When a renderer is computing the preferred size of a widget in its `getPreferredContentWidth` and `getPreferredContentHeight` methods it can only use attributes of the widget that if changed would cause revalidation.

- We said earlier that it is normally the widget class that takes responsibility for deciding when redisplay is necessary and, if it is, deciding whether to call `repaint()` or `revalidate()`. But widgets do not know how they are rendered – that is the responsibility of renderers. - so how can they tell which attributes are going to be used to determine the preferred size of the widget?

The usual resolution of this apparent conundrum is for the widget designer simply to take an informed view about which of the widget's attributes are allowed to affect the preferred size, and hence will cause revalidation if changed, and which are not, and hence will simply repaint if changed. These decisions are documented in the API of the widget class and look-and-feel designers must respect them when designing renderers.

If for some reason the widget designer felt it was essential to have more flexibility then a protocol can be established between the widget and its renderers where the widget will ask the renderer to make the “repaint or revalidate?” decision each time an attribute changes.

6.6 Synchronizing display updates

This section includes some advanced material you may wish to skip on first reading.

As we have mentioned before, calls to `repaint()` and `revalidate()` are not processed synchronously. Instead they cause events to be added to the MicroUI event queue. This is the same queue that is used for input events, such as pointer button presses. There is a single thread that processes the event queue, so events are processed strictly sequentially.

MicroUI performs some event merging to optimize system execution. For example, consecutive repaint events for the same object are merged. It follows, therefore, that if several widgets are being modified at the same time, causing multiple repaints and/or revalidates, there can be a performance advantage in ensuring that none of the resulting events are processed until all the events have been generated.

If the modifications are being made as a result of a call to a `handleEvent` method then the modifications are being made by the event thread, so none of the resulting events can be processed until the `handleEvent` method returns, which is exactly what we want. But we need to be more careful if the modifications are being made by some other thread, because then the event thread might start processing the resulting events before all the updates have been made.

This is not just a performance issue. Sometimes correct application behavior requires that we ensure that a set of updates to widgets are completed as a unit before any re-rendering takes place; animations are a good example.

MicroUI provides a facility to run any code you want as an event on the event thread. By using this feature you can ensure that the code will complete before any of the resulting events are processed. This feature is accessed via the `Display.callSerially(Runnable)` method. The code you need will look something like this:

```
desktop.getDisplay().callSerially(new Runnable() {
 public void run() {
  // your code here
 }
});
```

Bear in mind that you only need to use `callSerially` when:

- You are making several updates to widgets that need to be done as a unit, and

- The updates are not being done by the event thread.

7 Composites

A panel can hold only one widget but typically we want to include several widgets on the panel, which means using a composite. A composite is a widget that holds several other child widgets, and has built-in rules for laying them out. MWT provides the abstract class `Composite` that can hold multiple children but has no layout rules. A composite class has two responsibilities:

- It must implement the `validate(int widthHint, int heightHint)` method to encode its layout rules. As we saw in the previous section this method is called by the panel when it needs to be validated (laid out).
- It must ensure that focus navigation between its children works correctly. This may mean overriding the `getNext(int from, int direction)` method.

The validate method of a composite must always do the following:

- Call `validate` on each child, with the appropriate arguments,
- Call `setBounds` on each child, to set its size and position it within the composite,
- Call `setPreferredSize` to store the preferred size of the composite.

The preferred size of a composite should always be large enough to allow all its children to be seen.

Composites are themselves widgets, so it is possible to build more complex user interfaces by putting composites inside other composites.

When a composite is painted all its children are also painted.

7.1 A simple composite

In this section we will create a simple composite class, called `HorizontalStack`, that lays out its children in a row, giving each component as much space as it needs. Ex6 shows an example with five buttons:

```
public class Ex6 {

 public static void main(String[] args) {
  MicroUI.errorLog(true);
  MWT.RenderingContext.add(new TutorialTheme());
  Desktop desktop = new Desktop();
  Panel panel = new Panel();
  Composite composite = new HorizontalStack();
  for (int i = 0; i < 5; i++) {
   final int buttonNumber = i;
   Button button = new Button("Button " + buttonNumber);
   button.setListener(new Listener() {
    public void performAction() {
     System.out.println("Button " + buttonNumber + " pressed");
    }
    public void performAction(int value) {}
    public void performAction(int value, Object object) {}
   });
   composite.add(button);
  }
  panel.setWidget(composite);
  panel.show(desktop);
  desktop.show();
 }
}
```

The `HorizontalStack` is constructed and in the loop each button is added to it. The panel's widget is now the composite. The HorizontalStack class contains just one method: its implementation of validate:

```
public class HorizontalStack extends Composite {

	public void validate(int widthHint, int heightHint) {
		Widget[] widgets = getWidgets();
		int totalWidth = 0;
		int maxHeight = 0;
		for (int i = 0; i < widgets.length; i++) {
			Widget widget = widgets[i];
			widget.validate(0, heightHint);
			widget.setBounds(totalWidth, 0, widget.getPreferredWidth(), widget.getPreferredHeight());
			totalWidth += widget.getPreferredWidth();
			maxHeight = Math.max(maxHeight, widget.getPreferredHeight());
		}
		setPreferredSize(totalWidth, maxHeight);
	}
}
```

Figure 7.1. HorizontalComposite validate method

The three highlighted lines fulfill the three requirements of a `validate` method, as given earlier. The first asks the child to validate, with no constraint on the width and the supplied constraint on the height. The second line sets the bounds of the child so that its top left corner is in the correct place and its size is its preferred size. The third line sets the preferred size of the composite to the computed width and height.

If you run Ex6 you should see something like this:

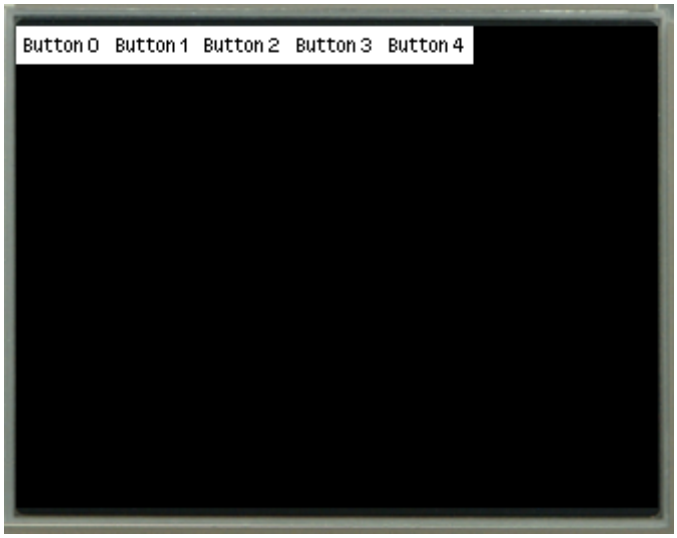

Figure 7.2. Example 6 executed on simulator

It doesn't look very nice because all the buttons are touching each other – we will see how to improve this in the next example.

If you have configured a joystick or cursor keys so that they generate `Command.LEFT` and `Command.RIGHT` you will be able to use those to move the input focus from button to button. The default implementation of `getNext`, which simply selects the next or previous child in the composite's list, works for our simple composite; for more complex composites it may be necessary to override and implement it.

7.2 Bounds, Margins and Padding

You will notice that the buttons displayed in example Ex6 are tightly packed against each other. We would like to leave some space between them. To understand how to do that we need to explore the concept of margin.

The margin is the amount of space, in pixels, that should be left clear around the widget. The button widget's parent (the `HorizontalStack` in this case) should ask the widget what margin it requires when allocating space to it.

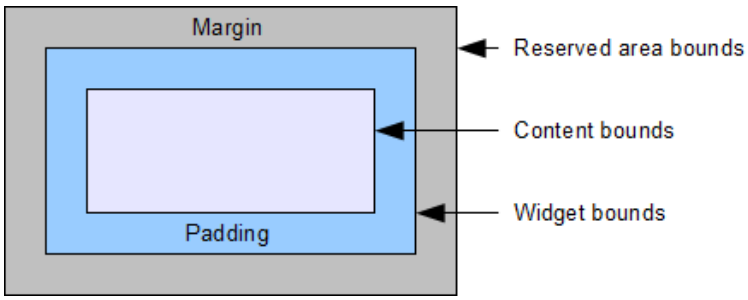

Figure 7.3. WidgetRenderer bounds

Composites should always always attempt to reserve an area for the widget that is at least as large as the widget's preferred size (which is the preferred content size computed by the widget's renderer plus twice the padding size as returned by `getPadding()`) plus twice the margin size. Note that the required margin and padding are defined by the widget's renderer and not by the widget itself.

The bounds of a widget define its position (relative to its parent) and its size. These bounds are set as part of the process of laying out a set of widgets on a panel, as we have already seen. When the `render` method is called on a renderer the origin of the supplied `GraphicsContext` is set to the top left of the widget's bounds. The renderer can only paint the area defined by the bounds – painting outside that area will be clipped.

From a composite's point of view, its “content” is made up of its children, where each child is surrounded by a margin and any padding defined for the composite itself. It follows, then, that we could obtain space between the buttons in the previous example either by defining a margin for the button or by creating a renderer for the composite and having it specify some padding, or by doing both.

7.3 Implementing margins and padding

In example Ex7 we will modify the `HorizontalStack` so that it respects any settings of margins and padding. We will specify a margin for buttons, and we will create a renderer for the composite that specifies padding. First we modify the `ButtonRenderer` to include a new method:

```
public class ButtonRenderer extends WidgetRenderer {

 ...

 public int getMargin() {
  return 3;
 }

 ...
}
```

This specifies a margin of 3 pixels around each button.

Next we create a `HorizontalStackRenderer` that includes the method:

```
public class HorizontalStackRenderer extends Renderer {

 ...

 public int getPadding() {
  return 6;
 }

 ...
}
```

This specifies a padding of 6 pixels between children in the composite.

Finally, we must enhance the validate method of `HorizontalStackRenderer` to take these settings into account. Unfortunately, although this does not affect the structure of the code it does make it look much more complicated:

```
public class HorizontalStack extends Composite {

     public void validate(int widthHint, int heightHint) {
          Widget[] widgets = getWidgets();
          int totalWidth = 0;
          int maxHeight = 0;
          Renderer myRenderer = getRenderer();
          int myPadding = myRenderer==null ? 0 : myRenderer.getPadding();
          heightHint -= myPadding * 2;
          for (int i = 0; i < widgets.length; i++) {
               Widget widget = widgets[i];
               Renderer widgetRenderer = widget.getRenderer();
               int widgetMargin = widgetRenderer==null ? 0 : widgetRenderer.getMargin();
               widget.validate(0, heightHint < (widgetMargin * 2) ? 0 : heightHint - (widgetMargin * 2));
               widget.setBounds(totalWidth + myPadding + widgetMargin, myPadding + widgetMargin,
                    widget.getPreferredWidth(), widget.getPreferredHeight());
               totalWidth += widget.getPreferredWidth() + (widgetMargin * 2) + myPadding;
               maxHeight = Math.max(maxHeight, widget.getPreferredHeight() + (widgetMargin * 2) + (myPadding * 2));
          }
          setPreferredSize(totalWidth + myPadding, maxHeight);
     }
}
```

Figure 7.4. Margin and padding management on composite

The first highlighted line obtains the padding specified for the composite. The second line obtains the margin specified for the child widget. Notice how the height hint passed to the child is reduced by both the composite's padding and the widgets margin to reflect the actual height available to the child.

We have added code to the `HorizontalStackRenderer` to make the bounds of each child obvious. The output of Ex7 looks like this:

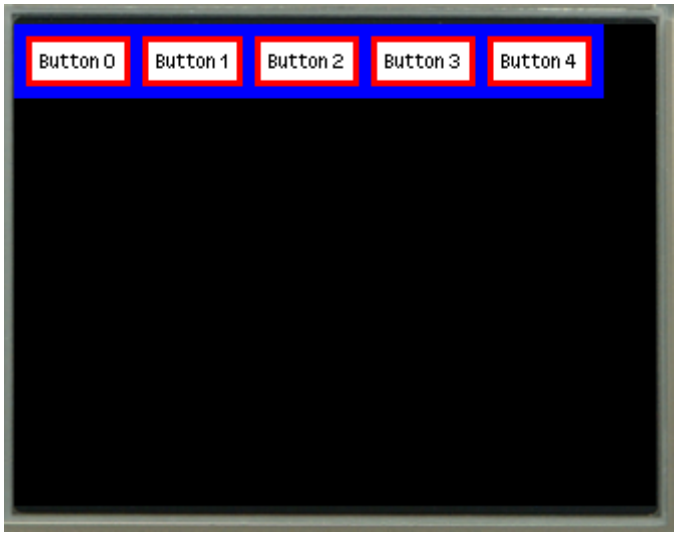

Figure 7.5. Example 7 executed on simulator

The background of the panel is blue and the background of each button's bounding box is red. The red rectangle is the margin area around each child and the blue space is the padding added by the composite.

In practice, having to create a renderer for a composite just to define padding for it can be tedious, so it is usually better if padding for the composite can be set on the composite itself, with this value being used in the absence of a renderer. The composite could define its own `getPadding` method, and the highlighted line in the `HorizontalStack` code above would become:

```
...
int myPadding = myRenderer==null ? getPadding() : myRenderer.getPadding();
...
```

The padding could be held in a field in the composite and set from the application.

7.4 Nested composites

As we have mentioned already, the child of a composite can be another composite, allowing the application designer to assemble widgets into complex panel structures. As a simple example we have created a variant of `HorizontalStack` called `VerticalStack` that arranges its children one above the other rather than side by side. We can use a combination of a `HorizontalStack` and a `VerticalStack` to create a grid (albeit one with imperfect alignment), as in Ex8:

```
public class Ex8 {

 public static void main(String[] args) {
  ...
  Composite rows = new VerticalStack();
  for (int rowNumber = 0; rowNumber < 3; rowNumber++) {
   Composite row = new HorizontalStack();
   for (int colNumber = 0; colNumber < 5; colNumber++) {
    final int buttonNumber = rowNumber*5 + colNumber;
    Button button = new Button("Button " + buttonNumber);
    button.setListener(new Listener() {
     public void performAction() {
      System.out.println("Button " + buttonNumber + " pressed");
     }
     public void performAction(int value) {}
     public void performAction(int value, Object object) {}
    });
    row.add(button);
   }
   rows.add(row);
  }
  panel.setWidget(rows);
  ...
 }
}
```

This example arranges 15 buttons in three rows of five. The output looks like this:

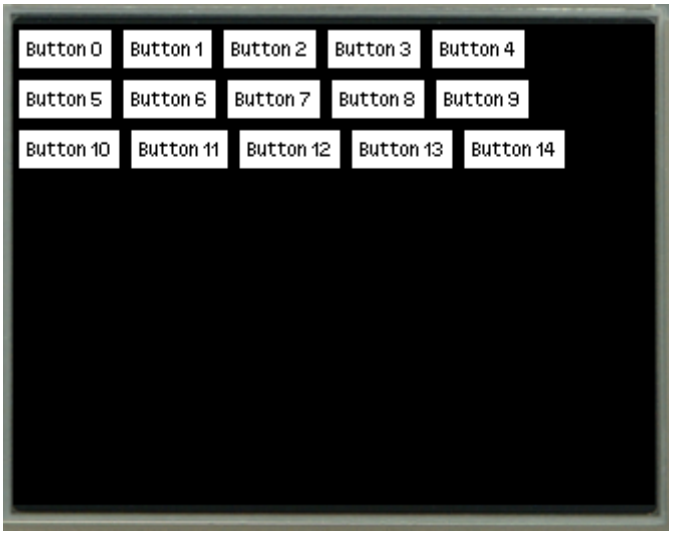

Figure 7.6. Example 8 executed on simulator

7.5 What if the children won't fit?

Our implementations of `HorizontalStack` called `VerticalStack` do not deal with the possibility that the stack of children will not fit in a single row or column across the panel. Try modifying Ex7 to make the button labels longer. The right-hand buttons are no longer visible. Possible strategies for coping with this are:

- Do nothing, as we have done in our examples. This strategy takes the view that the child widgets will have set the smallest preferred sizes they can and so if they don't fit the only solution is for the application designer to change their panel design.
- Enhance the composite so that it reduces the space made available to each child. A typical approach is to iterate over the children calling `validate(0, 0)` on each and computing the preferred total size, then calculating a factor by which each child must be reduced in size, and then iterating over the children again calling `validate` and `setBounds` with the computed size available to the child. Note that just calling `setBounds` in this second pass is not acceptable because the child (which might be a composite) must be given the opportunity to determine a new layout in the reduced area.
- Develop a scrolling composite that lets the user scroll or pan across the composite to see the children that would otherwise be hidden.

8 Widget control

8.1 Visibility

We can control whether or not a widget is visible. Widgets default to visible when constructed but can set invisible by calling `setVisible`, as in this example:

```
public class Ex9 {

   public static void main(String[] args) {
         ...
         Composite composite = new HorizontalStack();
         final Label label = new Label("Here are the details");
         label.setVisible(false);
         composite.add(label);
         ToggleButton button = new ToggleButton("Show/Hide Details");
         button.setListener(new Listener() {
               public void performAction(int value) {
                     System.out.println("Button is " + (value==0?"off":"on"));
                     label.setVisible(value==0?false:true);
               }
               public void performAction() {}
               public void performAction(int value, Object object) {}
         });
         composite.add(button);
         panel.setWidget(composite);
         ...
   }
}
```

Figure 8.1. Example 9 source code

The label is initially invisible but becomes visible when the button is pressed. A call to `setVisible` automatically schedules revalidation of the panel, as do several other Widget methods such as `setSize` and `setLocation`.

8.2 Disabled widgets

Widgets can be disabled. A disabled widget is still visible but it cannot gain focus and cannot receive events. A widget is enabled on construction and can be disabled (or re-enabled) by calling its `setEnabled` method.

In the example below there are two buttons on the panel. The right button is used to enable and disable the left button. Here is the main class of the example:

```
public class Ex9a {

   public static void main(String[] args) {
         ...
         Composite composite = new HorizontalStack();
         final ToggleButton button = new ToggleButton("Press Me");
         button.setListener(new Listener() {
               public void performAction(int value) {
                     System.out.println("Button is " + (value==0?"off":"on"));
               }
               public void performAction() {}
               public void performAction(int value, Object object) {}
         });
         ToggleButton enablingButton = new ToggleButton("Enable/Disable");
         enablingButton.setOn(true);
         enablingButton.setListener(new Listener() {
               public void performAction(int value) {
                     button.setEnabled(value==0?false:true);
               }
               public void performAction() {}
               public void performAction(int value, Object object) {}
         });
         composite.add(button);
         composite.add(enablingButton);
         panel.setWidget(composite);
         ...
   }
}
```

Figure 8.2. Example 9a source code

The highlighted line enables or disables the left button. The call to `setEnabled` automatically causes the widget to be repainted.

It is normal to render a disabled widget in a different way, so that the user knows it is disabled. We have changed the `render` method of the toggle button renderer to use different background colors when the button is disabled:

```
     public void render(GraphicsContext g, Renderable renderable) {
           ToggleButton button = (ToggleButton) renderable;
           g.setColor(button.isOn()?
                     button.isEnabled()?Colors.RED:Colors.MAROON
                 :
                     button.isEnabled()?Colors.WHITE:Colors.GRAY);
           g.fillRect(0, 0, button.getWidth(), button.getHeight());
           g.setColor(Colors.BLACK);
           g.drawString(button.getText(), button.getWidth() / 2, button.getHeight() / 2,
                     GraphicsContext.HCENTER | GraphicsContext.VCENTER);
           if (button.hasFocus()) {
                 g.setStrokeStyle(GraphicsContext.DOTTED);
                 g.drawRect(1, 1, button.getWidth()-3, button.getHeight()-3);
           }
     }
```

Figure 8.3. ToggleButtonRenderer extended

Some widgets should always be disabled. For example, the `Label` widget we created earlier should always be disabled because it cannot accept user input. In the next example we have extracted an abstract superclass from `Label`, which both `Label` and `ToggleButton` extend, so that the `Label` subclass can ensure labels are disabled on construction.

9 Look and Feel

The look and feel of an MWT user interface is defined by a theme – realized as an instance of a subclass of the MWT class Theme. Each theme holds a coherent set of renderers and a `Look`, which defines visual properties such as colors. By “coherent” we mean that the renderers share some common style elements – maybe they all use rounded rectangles, for example. We have already created a theme, the `TutorialTheme` class we have used in the examples so far. But that class has not so far been associated with a Look. In the next example we create a look (the class `TutorialLook` that implements the `Look` interface), and refer to it in the renderers to eliminate the hard-coded references to colors and fonts.

9.1 Look

A look comprises:

- An array of fonts, and
- A map that relates constants identifying logical colors and fonts to actual colors and indexes into the font array.

The two important methods of a look are:

- `DisplayFont[] getFonts()`, which returns an array of the fonts configured for the look, and
- `int getProperty(int)`, which returns the value associated with the constant that is the `int` argument.

Renderers obtain a reference to the current look using `getLook()` and retrieve information from the look using those two methods.

A standard set of property constants is defined in the `Look` interface, but provided the theme's renderers and the theme's look use a consistent set of constants the designers can define whatever constants he or she wants. However, there are some implications with using non-standard constants, as discussed below. The property constants in `Look` assume that the information required by a renderer falls into one of these categories:

- A color to be used for the background,
- A color to be used for a border,
- A color to be used for the foreground,
- A style of font.

For each category `Look` defines a separate property constant for each of these possible states or aspects of the widget of widget component:

- Default,
- Content,
- Focused,
- Disabled,
- Selection.

The meanings of these states are not defined by MWT, although their intent can be guessed. There are four categories and five states, making a total of 20 property constants. Even if the `Look` is using only

the standard constants it doesn't have to implement them all, although again there are some implications of not doing so, as we will discuss later.

Here is our implementation of a look for the tutorial (in the `com.is2t.mwt.tutorial.ex10` package):

```
public class TutorialLook implements Look {
 private static final DisplayFont[] fonts;

 static {
  fonts = new DisplayFont[]{
  DisplayFont.getDefaultFont()};
 }

 public DisplayFont[] getFonts() {
  return fonts;
 }

 public int getProperty(int resource) {
  switch(resource) {
   // Borders use this for all borders
   case Look.GET_BORDER_COLOR_DEFAULT:
    return Colors.BLACK;
   // Backgrounds use this for desktop background
   case Look.GET_BACKGROUND_COLOR_DEFAULT:
    return Colors.GRAY;
   // Background of widgets that are enabled and not selected
   case Look.GET_BACKGROUND_COLOR_CONTENT:
    return Colors.WHITE;
   // Background of widgets that are disabled and not selected
   case Look.GET_BACKGROUND_COLOR_DISABLED:
    return 0xAAAAAA; // light gray
   // Background of widgets that are disabled and selected
   case Look.GET_BACKGROUND_COLOR_FOCUSED:
    return 0xAA0000; // dull red
   // Background of widgets that are enabled and selected
   case Look.GET_BACKGROUND_COLOR_SELECTION:
    return Colors.RED;
   // Foregrounds use this for all foreground
   case Look.GET_FOREGROUND_COLOR_DEFAULT:
    return Colors.BLACK;
   // Font indexes
   case Look.GET_FONT_INDEX_DEFAULT:
    return 0;
   default:
    throw new IllegalArgumentException();
  }
 }
}
```

The first thing to notice is that we have not defined all 20 properties, but just the subset we need for our simple set of widgets. Notice also that we have defined using comments what we expect each property to be used for. Following this guide, renderer developers can select the correct property.

Let us take a look at how the look is used on one of our renderers:

```
public class LabelRenderer extends WidgetRenderer {

 public Class getManagedType() {
  return Label.class;
 }

 public int getPreferredContentWidth(Widget widget) {
  TextHolder label = (TextHolder) widget;
  String text = label.getText();
  return getNormalFont().stringWidth(text);
 }

 public int getPreferredContentHeight(Widget widget) {
  return getNormalFont().getHeight();
 }

 public void render(GraphicsContext g, Renderable renderable) {
  TextHolder label = (TextHolder)renderable;
  g.setColor(getLook().getProperty(Look.GET_BACKGROUND_COLOR_CONTENT));
  g.fillRect(0, 0, label.getWidth(), label.getHeight());
  g.setColor(getLook().getProperty(Look.GET_FOREGROUND_COLOR_DEFAULT));
  g.setFont(getNormalFont());
  g.drawString(label.getText(), label.getWidth() / 2, label.getHeight() / 2,
    GraphicsContext.HCENTER | GraphicsContext.VCENTER);
 }

 ...

 private DisplayFont getNormalFont() {
  return getLook().getFonts()[getLook().getProperty(
    Look.GET_FONT_INDEX_DEFAULT)];
 }
}
```

(We have omitted the `getMargin` and `getPadding` methods to save space.) As you can see, all the hard-coded references to colors and fonts have been replaced with calls to the look. Although this makes it much easier quickly to change the look-and-feel it does tend to make the renderer code less readable, so it is best to create helper methods for the more long-winded look-ups, as we have done with `getNormalFont`. In fact, when creating a theme it is common to create a base class for all the renderers in the theme that can hold these helper methods, and provide default implementations of the `getMargin` and `getPadding` methods, which will typically return the same values for every renderer in the theme. You will see a base class of this kind in later examples.

Try running Ex10 to see the standard look-and-feel in action.

9.2 Changing looks and themes at run-time

A theme has a default `Look`, returned by the `getDefaultLook` method, and that look will be used by all renders in the theme unless it is explicitly changed. The look of a theme can be changed by calling the theme's `setLook` method, passing in the the look to be used. Obviously, care must be taken to ensure that the new look provides all the properties required by the renderers in the theme. A typical call might be like this:

```
myTheme.setLook(new OtherLook());
```

A new look for all the themes currently registered with the rendering context can be requested by calling the rendering context's `setLook` method, passing in the look to be used. The new look will only be set on themes that reply true to `isStandard`. A theme should reply true to `isStandard` if the renderers that make it up request only those look properties defined by the constants in the `Look` interface. The idea is to provide a simple way of changing the look for the entire GUI without impacting themes that rely on special look properties. It follows, then, that when making a call like:

```
MWT.RenderingContext.setLook(new PlainLook());
```

The supplied look must support all the properties defined in `Look` because the caller is guaranteeing that the look will work with any theme that guarantees its renderers use only the standard properties.

We have already seen how to add a theme to the rendering context. All the examples include a statement such as this:

```
MWT.RenderingContext.add(new TutorialTheme());
```

The theme as added at the end of the list of registered themes. Additional themes can be added at any time. When a new theme is added its renderers immediately become available for use, but if two or more renderers are equally suitable for rendering a widget the renderer that is nearest the front of the list is used. So if you want to replace a theme by a different but equivalent theme then first add the new theme and then remove the previous theme using:

```
MWT.RenderingContext.remove(myTheme);
```

10 Widgets Library

Before we create any more widgets, let us take stock of what we have done so far, and consider what we mean by a “widget library”. Consider this UML class diagram:

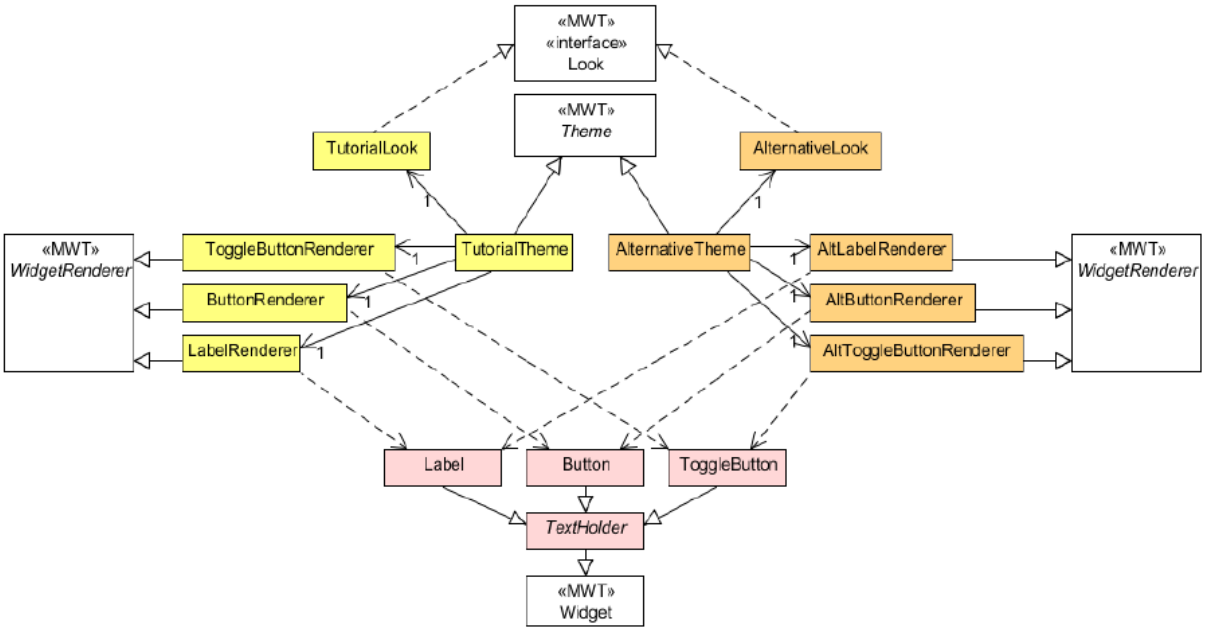

Figure 10.1. Widgets library UML class diagram

The classes shaded pink are the widgets we have built so far in this tutorial. The classes shaded yellow are the look-and-feel we have created. The classes shaded orange are an alternative look-and-feel that we might create but we haven't yet. Together with our two composite classes (not shown on the diagram) these classes constitute the beginnings of a widget library.

So Widgets library comprises:

- A set of widget classes (including composites),
- One or more themes, with their renderers and default looks,
- Optionally, other looks that could be used.

11 Widget and Renderer Collaboration

We will now create a more complex widget it and use it to explain some other features of MWT. Our new widget will have as its model a bounded integer. We will call this widget a scale. Renderers will be able to render this widget to show and adjust its value.

11.1 Renderer contracts

Most of the design of the `Scale` class is straightforward, but there is one tricky issue. With the buttons we created in earlier examples handling pointer events was simple: the widget only receives the event if the pointer button is pressed when the pointer is over the widget, and any such press is considered to have “pressed” the button widget. The there are many different ways in which a scale could be rendered, for example:

- As a slider,
- As a read-only text box with up and down buttons,
- As a rotary control (a knob).

Each of these would require very different handling of pointer events. So for the scale widget we decide to delegate responsibility for handling pointer events to the renderer, by calling a special method of the renderer. Since there can be many different renderers for `Scale` we define an interface, called `ScaleRendererContract`, that specifies this special method, and all renderers than wish to render scales must implement it.

A related problem is that the scale's renderer may need to keep track of what's happening with the user interaction. For example, the renderer may wish to render the scale differently if a pointer drag is in progress. Renderers cannot store widget-related state because they are used to render many widgets. So we define a `Scale` attribute called `renderingState` and provide accessors to it that the renderer can use. The scale has no idea what its renderer is using this for, if indeed it is using it. This is slightly risky in that if the theme is changed at run-time the new renderer may misinterpret the state stored by the previous renderer, but the effect is unlikely to be serious.

```
public boolean handleEvent(int event) {
    int type = Event.getType(event);
    if (type == Event.POINTER) {
        ScaleRendererContract renderer =
            (ScaleRendererContract) getRenderer();
        int action = Pointer.getAction(event);
        Pointer pointer =
            (Pointer)Event.getGenerator(event);
        int x = pointer.getX();
        int y = pointer.getY();
        return renderer.processPointerEvent(this, action,
            x - getAbsoluteX(), y - getAbsoluteY());
    }
    return super.handleEvent(event);
}
```

Figure 11.1. Scale widget source code

The highlighted lines show the use of the `ScaleRendererContract`. You can also see the definition of the `renderingState` attribute.

For this example we will create a renderer that renders the scale using a slider control, called `ScaleRenderer`. Most of the code in that class is concerned with the intricacies of drawing the slider and figuring out where the “thumb” – the knob on the slider that the user drags – should be. But the method that handles the pointer events is worth reviewing:

```
public boolean processPointerEvent(Scale scale, int action, int x,int y) {
	switch (action) {
	case Pointer.PRESSED:
		if (thumbContains(scale, x, y)) {
			scale.setRenderingState(STATE_ADJUSTING);
			scale.repaint();
		}
		return true;
	case Pointer.RELEASED:
		scale.setRenderingState(STATE_IDLE);
		scale.repaint();
		return true;
	case Pointer.DRAGGED:
		if (scale.getRenderingState() == STATE_ADJUSTING) {
			scale.setValue(computeValue(scale, x, y));
		}
		return true;
	}
	return false;
}
```

Figure 11.2. ScaleRenderer source code

You can see that the renderer is using two state constants to implement a trivial state machine, with the current state held in the widget's `renderingState` attribute. When the renderer decides that the widget's value should change it calls back to the widget, as in the highlighted line. The `setValue` method causes a repaint, which is why the renderer does not request one in this case. Note that this method returns true or false to indicate whether the event has been consumed; the widget propagates that back it its caller.

The main class Ex11 demonstrates the use of `Scale` and `ScaleRenderer` by displaying two sliders next to each other. The right-hand slider is disabled. The output looks like this:

Figure 11.3. Example 11 executed on simulator

11.2 Renderer contracts

All the renderers we have created so far in this tutorial always paint the background color over the entire bounding rectangle of the widget. In some cases that is correct – for example, that is probably what we want for buttons. But for the slider we just created it would look better if the renderer painted only the center strip and the thumb, leaving the rest of the area to match the background of the parent. We could try adjusting the `TutorialLook` properties to make the backgrounds match, but that will not solve the general problem – imagine if the desktop background was an image, for example.

In MWT widgets can be transparent, meaning their renderers should not paint the background, just those elements they need to paint. So that the background is shown correctly (for example, when the slider thumb is moved the area previously occupied by it must be repainted) MWT will ask the widget's parent to repaint itself, with the painting clipped to the area of the widget, before calling the renderer's `render` method. If the parent is itself transparent the repaint request is propagated up the panel hierarchy until a non-transparent element is found.

A widget is deemed to be transparent if it replies `true` to `isTransparent()`. Obviously any widget class can override this method to implement any logic it likes, but the Widget class provides a simple default. By default, `isTransparent` will reply `true` if no renderer can be found for the widget and `false` otherwise. This means that by default widgets that are being rendered on the display are non-transparent.

All widget renderers must call isTransparent() on the widget being rendered to determine whether or not to paint the whole background area.

There are various strategies the widget designer can adopt with regard to transparency:

- Override `isTransparent` to return `true` in those widget classes that should be treated as transparent,
- Implement a flag in the widget class that can be set as required in each instance, and use the state of that flag to decide how to respond to `isTransparent`,
- When `isTransparent` is called, delegate the decision to the renderer,
- A combination of options 2 and 3: provide a flag but also take into account the preference of the renderer.

Option 1 is simple but not very flexible. Option 2 is flexible but requires extra memory in every widget. Option 3 is good if it is unlikely that different instances of the widget that are rendered in the same way would need different transparency settings. Option 4 is the most flexible but also the most complex.

In Ex12 we will show examples of options 2 and 3. We decide that all widgets that derive from `TextHolder` (i.e. labels and buttons) will have a transparency flag, while the `Scale` class will defer to the renderer. The changes required to the `TextHolder` class (compared to its previous appearance in Ex10) are:

```
public abstract class TextHolder extends Widget {

   protected String text;
   protected boolean transparent;

   public TextHolder(String text) {
         super();
         this.text = text;
         transparent = false;
   }

   public String getText() {
         return text;
   }

   public void setText(String text) {
         this.text = text;
         revalidate();
   }

   public boolean isTransparent() {
         return getRenderer() == null || transparent;
   }

   public void setTransparent(boolean transparent) {
         this.transparent = transparent;
   }

}
```

Figure 11.4. TextHolder source code

Note that the flag is initialized to `false`, so `TextHolder` objects default to non-transparent. Note also that the widget is always transparent if it has no renderer; this is the MWT standard.

The changes required to the `Label` class (compared to its previous appearance in Ex10) are:

```
public class Label extends TextHolder {

    public Label(String text) {
          super(text);
          setEnabled(false);
          setTransparent(true);
    }
}
```

Figure 11.5. Transparent Label source code

We want labels to be transparent by default, so the `Label` constructor sets the transparency attribute. The `LabelRenderer` must now respect that setting:

```
public void render(GraphicsContext g, Renderable renderable) {
      TextHolder label = (TextHolder)renderable;
      if (!label.isTransparent()) {
            g.setColor(getLook().getProperty(Look.GET_BACKGROUND_COLOR_NORMAL));
            g.fillRect(0, 0, label.getWidth(), label.getHeight());
      }
      g.setColor(getLook().getProperty(Look.GET_FOREGROUND_COLOR_NORMAL));
      g.setFont(getNormalFont());
      g.drawString(label.getText(), label.getWidth() / 2, label.getHeight() / 2,
                  GraphicsContext.HCENTER | GraphicsContext.VCENTER);
}
```

Figure 11.6. Renderer handling tansparency

The Scale implementation of isTransparent will ask the renderer. So we add another method to the ScaleRendererContract and call it in `isTransparent()`:

```
public class Scale extends Widget {
 ...
 public boolean isTransparent() {
  ScaleRendererContract renderer = (ScaleRendererContract) getRenderer();
  return renderer == null || renderer.isTransparentPreferred();
 }
 ...
}
```

The renderer can now state its preference in the isTransparentPreferred method. The `ScaleRenderer` prefers to treat the widget as transparent:

```
public class ScaleRenderer extends WidgetRenderer implements ScaleRendererContract {
 ...
 public boolean isTransparentPreferred() {
  return true;
 }
 ...
}
```

The reader will notice that we have included in the `render` method of `ScaleRenderer` the same conditional logic we put in `LabelRenderer`, even though the renderer expects the widget to be transparent. This is good practice – maybe the `Scale` class will change its logic one day.

The Ex12 main class illustrates the difference between the two renderers. It creates two labels and a scale, with one of the labels transparent and the other not:

```
public class Ex12 {

      public static void main(String[] args) {
            MicroUI.errorLog(true);
            MWT.RenderingContext.add(new TutorialTheme());
            Desktop desktop = new Desktop();
            Panel panel = new Panel();
            Scale scale = new Scale(0, 100);
            Label label1 = new Label("Trans");
            Label label2 = new Label("!Trans");
            label2.setTransparent(false);
            HorizontalStack row = new HorizontalStack();
            row.add(label1);
            row.add(label2);
            VerticalStack rows = new VerticalStack();
            rows.add(row);
            rows.add(scale);
            panel.setWidget(rows);
            panel.show(desktop);
            desktop.show();
      }
}
```

Figure 11.7. Example 12 source code

Labels default to transparent, so the highlighted line changes this setting.

You can see the results in the output:

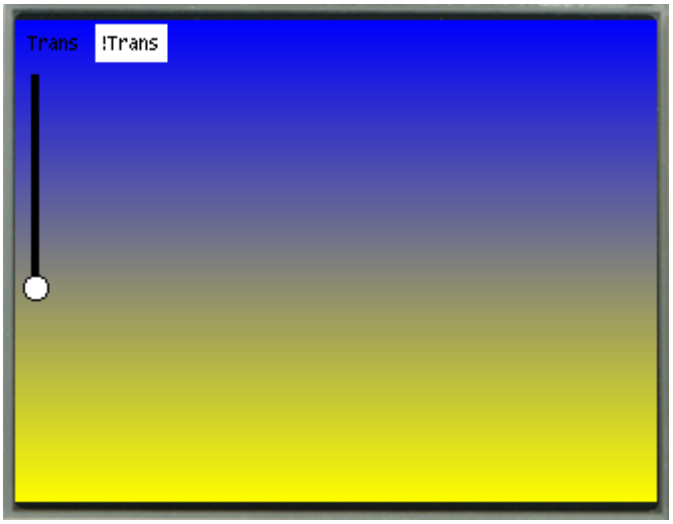

Figure 11.8. Example 12 executed on simulator

To make things more obvious we have changed the desktop renderer to paint a fading color pattern on its background.

11.3 Selecting renderers by style

Throughout this tutorial we have repeatedly made the point that a widget can be rendered in different ways by different renderers, but so far we have only had one renderer capable of rendering each type of widget. Now we will show how you can have multiple renderers.

The primary way in which MWT selects an appropriate renderer for a widget is by comparing the class returned by the renderer's `getManagedType` method with the class of the widget. The best possible match is that the two classes are the same, but if there is no exact match then MWT will select a renderer that indicates it is capable of rendering a superclass of the widget.

If there are two renderers whose `getManagedType` response exactly matches the class of the widget then MWT will select the first such renderer registered, via its theme, with the `RenderingContext`. How, then, can we put two instances of `Scale` on a panel and have them rendered by different renderers?

We already have one renderer for `Scale` widgets, the `ScaleRenderer`. In this example we will create another renderer, called the `HangingBoxRenderer`, which is also capable of rendering `Scale` widgets.

One way in which we could achieve our goal of having two `Scale` widgets on the panel, one rendered by the `ScaleRenderer` and the other by the `HangingBoxRenderer`, is to create a subclass of `Scale` that adds no behavior and make the `getManagedType` method of `HangingBoxRenderer` return that subclass. Then in our application we would create one instance of `Scale` and one instance of the subclass. MWT would find exact matches for both classes and so select the desired renderer. Although that will work it is very inconvenient, and MWT provides an easier and more flexible way.

The `Widget` class implements the method `getStyle()` that returns the integer 0. The `Renderer` class implements the method `getManagedStyle()` that also returns 0. These methods can be overridden by our widget and renderer classes respectively to return non-zero values. MWT will then use these methods to resolve situations where it finds two equally suitable renderers for a widget based on class matching. It does this by calling the widget's `computeScore(Renderer)` method with each possible renderer in turn and selecting the renderer that gives the highest score. The default implementation of `computeScore` in `Widget` simply compares the widget's `getStyle` value with the renderer's `getManagedStyle` value and returns the number of 1-bits in the two integers that match.

Everything above applies equally to desktops and panels as it does to widgets: the same technique of matching by style can be used with them, too.

Now let us look at how this feature has been used in the example (Ex13). The existing `ScaleRenderer` class has been enhanced to define a style constant and override the `getManagedStyle` method:

```
public class ScaleRenderer extends WidgetRenderer implements ScaleRendererContract {
 public static final int STYLE = 1;
 ...
 public int getManagedStyle() {
  return STYLE;
 }
 ...
}
```

The new `HangingBoxRenderer` class extends `ScaleRenderer` and overrides `getManagedStyle` to return a different value. It also overrides render to render to `Scale` widget differently:

```
public class HangingBoxRenderer extends ScaleRenderer {
 public static final int STYLE = 2;
 ...
 public int getManagedStyle() {
  return STYLE;
 }
 public void render(GraphicsContext g, Renderable renderable) {
  ...
 }
}
```

We change the `Scale` class to add an extra argument to its constructor that allows us to specify the required style. This style is then returned by the scale's `getStyle` method:

```
public class Scale extends Widget {
 ...
 private int style;

 public Scale(int lowerBound, int upperBound, int style) {
  super();
  this.lowerBound = lowerBound;
  this.upperBound = upperBound;
  this.style = style;
 }

 public int getStyle() {
  return style;
 }
 ...
}
```

The main class Ex13 looks like this:

```
public class Ex13 {

   public static void main(String[] args) {
         MicroUI.errorLog(true);
         MWT.RenderingContext.add(new TutorialTheme());
         Desktop desktop = new Desktop();
         Panel panel = new Panel();
         Scale scale1 = new Scale(0, 100, SliderRenderer.STYLE);
         scale1.setListener(new Listener() {
               public void performAction(int value) {
                     System.out.println("Value(1) is now " + value);
               }
               public void performAction() {}
               public void performAction(int value, Object object) {}
         });
         Scale scale2 = new Scale(0, 100, HangingBoxRenderer.STYLE);
         scale2.setListener(new Listener() {
               public void performAction(int value) {
                     System.out.println("Value(2) is now " + value);
               }
               public void performAction() {}
               public void performAction(int value, Object object) {}
         });
         HorizontalStack composite = new HorizontalStack();
         composite.add(scale1);
         composite.add(scale2);
         panel.setWidget(composite);
         panel.show(desktop);
         desktop.show();
   }
}
```

Figure 11.9. Example 13 source code

Notice how we are using the style constants defined in the renderer classes to select the style for each of the scale widgets. If there were lots of renderers for scales we might want to create a separate interface to hold these constants so that we can keep track of them in one place.

The output from running Ex13 looks like this:

Figure 11.10. Example 13 executed on simulator

12 Panels and Dialogs

12.1 Multiple panels

So far we have had only one panel on the desktop, but a desktop can hold many panels, of which only one will be active. The active panel receives the input events.

To add a panel to a desktop call `Panel.show(Desktop)` or `Panel.show(Desktop, boolean)` as we have seen in previous examples. When you call one of those methods the panel becomes the active panel. To remove a panel from the desktop call `Panel.hide()`. You can use panels to tile the desktop, but more often you use overlapping panels that effectively increase the desktop area. The desktop holds all its panels in a stack, with the active panel at the top, and when a panel is rendered the drawing is clipped to reflect that stacking – panels under other panels will not be seen.

A panel becomes active either as a result of calling its show method or if the pointer is clicked on it. When a panel becomes active its `becameActive()` method is called. When a panel becomes inactive its `becameInactive()` method is called. It is important to note that a panel is not automatically repainted when it becomes active or inactive, so you will normally want to override those two methods to at least force a repaint.

Example Ex14 includes two overlapping panels:

```
public static void main(String[] args) {
 ...
 Panel panel1 = new Panel(0, 0, desktop.getWidth()/3*2, desktop.getHeight()/3*2) {
  public void becameActive() {
   repaint();
  }
  public void becameInactive() {
   repaint();
  }
 };
 Panel panel2 = new Panel(20, 20, desktop.getWidth()/3*2, desktop.getHeight()/3*2) {
  public void becameActive() {
   repaint();
  }
  public void becameInactive() {
   repaint();
  }
 };
 ...
 panel1.show(desktop);
 panel2.show(desktop);
 desktop.show();
}
```

Each panel is explicitly sized to occupy two-thirds of the desktop, with panel2 offset from the top-left by 20 pixels. Notice the overrides of `becameActive` and `becameInactive` to force repainting. Panel2 is initially the active panel because it is the last to be shown on the desktop. In this example we have defined a panel renderer that clearly shows the active panel: the active panel has a green background and inactive panels have a yellow background.

If you run Ex14 you will see this output:

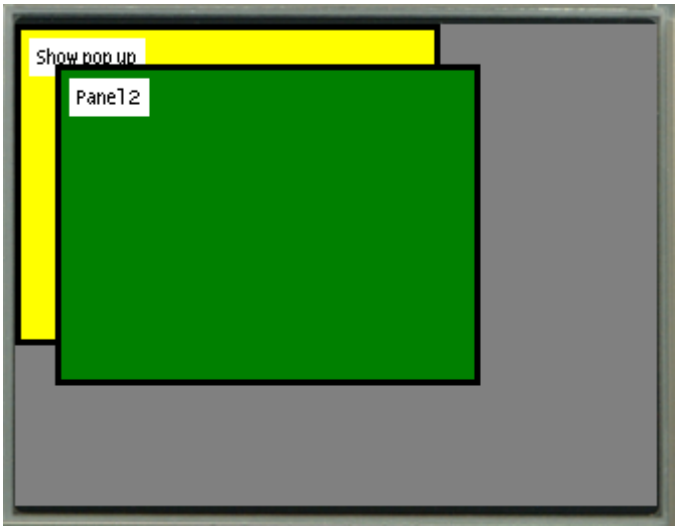

Figure 12.1. Example 14 executed on simulator

Avoiding pressing the buttons on the panels, click on the background of panel1 (the yellow one). It becomes active and is repainted at the front. Clicking on panel2 brings it to the front again.

12.2 Dialogs

A `dialog` is a panel that once active remains active, and so consumes all events, until it is explicitly hidden (often called a modal dialog). A common use for dialogs is to create pop-ups that must be dealt with by the user before he or she can do anything else.

Ex14 includes such a pop-up:

```
   public static void main(String[] args) {
         ...

(1)      final Panel popUp = new Dialog();
         Button panel1button = new Button("Show pop up");
         panel1button.setListener(new Listener() {
               public void performAction() {
(2)                popUp.show(desktop);
               }
               public void performAction(int value, Object object) {}
               public void performAction(int value) {}
         });
         Button panel2button = new Button("Panel 2");
         Button popUpDismiss = new Button("OK");
         popUpDismiss.setListener(new Listener() {
               public void performAction() {
(3)                popUp.hide();
               }
               public void performAction(int value, Object object) {}
               public void performAction(int value) {}
         });
         panel1.setWidget(panel1button);
         panel2.setWidget(panel2button);
         popUp.setWidget(popUpDismiss);
         popUp.validate();
         int width = popUp.getPreferredWidth();
         int height = popUp.getPreferredHeight();
         popUp.setBounds((desktop.getWidth()-width)/2, (desktop.getHeight()-height)/2,
               width, height);
         ...
   }
```

Figure 12.2. Example 14 source code

The highlighted lines show (1) the dialog being created; (2) the dialog being shown when the button on panel1 is pressed; (3) the dialog being hidden when the button on the dialog is pressed. The code at the bottom is sizing and positioning the dialog so that it is just big enough to hold its contents and pops-up in the center of the desktop.

The output below shows the dialog being displayed:

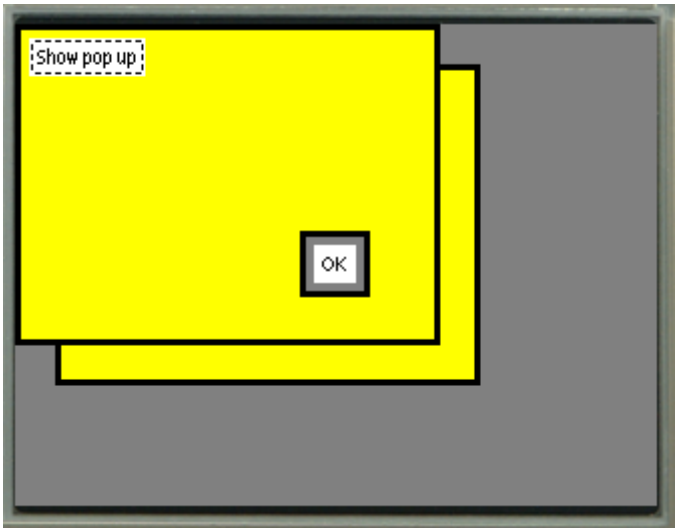

Figure 12.3. Example 14 source code

13 Document History

Date	Revision	Description
November 19th, 2013	A	First public release
April 08th, 2014	B	Change prerequisites to run the example 3

Headquarters
11, rue du chemin Rouge
44373 Nantes Cedex 3
FRANCE
Phone: +33 2 40 18 04 96
www.is2t.com