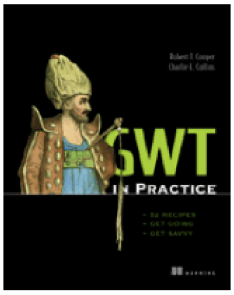
Published on *Javalobby* (http://java.dzone.com)
Design patterns and GWT
By *Volume4*
Created *2008/07/21 - 3:41pm*

Related Reading



GWT in Practice

Robert Cooper and
Charles Collins
May, 2008 | 376 pages
ISBN: 1-933988-29-0
Manning Publications

We have a calculator EntryPoint implementation that places a CalculatorWidget on the RootPanel, so we now need to provide this widget. This will be our own widget, composed of basic GWT components. CalculatorWidget will contain the view for our calculator application, a simple data and logic layer as the model, and an intermediary controller between these layers. This, of course, means we'll be using MVC and GWT together - keep in mind that all of these components will end up distributed to the client.

MVC will be the overarching pattern for the macro level of our application. Additionally, we'll use some other common object-oriented (OO) patterns on other levels because of the general benefits these approaches provide. For example, we'll use the Observer pattern for event handling as part of the mechanism of communication between our model and our view. We'll also use the Strategy pattern within the logic and operations parts of our model. We'll provide more detail about each of these patterns, and the reasoning behind their use in each situation, as we come to those portions of the application.

The most important pattern we need to consider is MVC itself. This pattern is the most useful we have found for creating GWT applications, while still keeping them manageable and testable.

MVC and GWT

[1] MVC is a familiar pattern to many developers. The pattern was first described in the late 70s by Trygve Reenskaug, who was working on Smalltalk at the time, and it has since been adopted, in various forms, by a wide range of programmers using many languages. The pattern is intended to separate responsibilities and allow for reuse of the logic and data portion of an application' the model. The view layer represents the model graphically, as the UI. Delegation between these layers is handled by an intermediary that is invoked by the user's input, the controller. The basic MVC pattern is shown in figure 1.

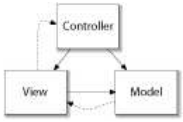


Figure 1 A basic model-view-controller architectural pattern diagram. The solid lines indicate a direct association, and the dashed lines indicate an indirect association, such as through an observer/observable relationship.

The key concepts to take away from MVC are the following: the model should be completely unaware of the view and controller, and the pattern can be applied on many different levels within an application.

In MVC, the model can be equipped to alert observers of changes within it, using the Observer pattern (something we'll look more closely at in the section on communicating by observing events). This is shown by the dashed line in figure 1. In this way, any other layer observing the model can react to events coming from the model. Nevertheless, the model does not directly update the view, and this distinction is important. The model is the reusable part in all incarnations of MVC; if the model had any direct knowledge of the view, that tenet would break down. How the view and controller are linked together is often more variable.

The view typically has a reference to the controller, and may invoke controller methods directly, based on user input. The controller ideally will have no direct references back to the view, and should simply update the model. If the view is an observer of the model, this effectively updates the view without direct controller/view coupling. Alternatively, it's also common to see situations in which the controller and view are treated as a pair, with direct references from the controller to the view.

There are many ways to divide up the responsibilities, and the various approaches have their pros and cons. The important part, though, is that the separation of responsibilities makes the code clearer, and often also makes testing easier. MVC can be applied with GWT, and it can happen on many different levels. An overall application may have a master domain, or model, and an overarching controller. Individual widgets can have their own encapsulated MVC, and small elements, such as a button, can repeat the pattern on their own level.

The possible variations of MVC, and many somewhat similar GUI architectural patterns, such as Presentation Abstraction Control (PAC) and Model View Presenter (MVP), are very interesting but are beyond our scope here. Fortunately, entire tomes on the subject exist. For our purposes, weâ€™ll turn to concrete examples of MVC that we have found work well with GWT. To do this, we need to get into code and implement the pattern.

Creating a widget

The functional specifications for a calculator (at least a simple one) are fairly universal; they include a display, some buttons,

and some mathematical operations defined by the likes of Pythagoras and Euclid. For our calculator, we'll make use of one GWT TextBox for the display and many GWT Buttons for, well, the buttons. We'll also use a layout derived from a Panel and a Grid to control the overall placement of the UI items. In terms of MVC, view elements are where all widgets begin, starting with the layout. The model and controller for our calculator will be implemented as separate classes that are referenced by the widget.

With GWT, you use widgets to create your application. These components are built from layout panels, input elements, events, data objects, and various combinations thereof. This is where GWT really departs from typical web applications and may seem somewhat foreign to those accustomed to standard server-side Java development. To create an application, you use widgets that are capable of being inserted or removed from the screen, in a single page, on the client side.

For our purposes, we'll extend the GWT-provided VerticalPanel component to build our view, and then include references to our separate controller and model, as shown in the class diagram in figure 2.

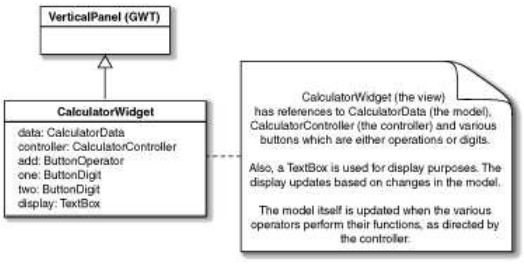


Figure 2 CalculatorWidget top-level class diagram showing the extension of GWT's VerticalPanel as the view, and the included model and controller references

It's worth noting that figure 2 does not display all the attributes or operations of the CalculatorWidget class. We have included just a representative sample of attributes to keep it concise. The code for the CalculatorWidget will be displayed in three parts in listings 1, 2, and 3. Listing 1 addresses the beginning of the class, showing what the class needs to import and how the class is derived from, and makes use of, other existing GWT classes.

Listing 1 CalculatorWidget.java, part one:

```
package com.manning.gwtip.calculator.client;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.Grid;
import com.google.gwt.user.client.ui.TextBox;
//. . . remainder of imports omitted
public class CalculatorWidget extends VerticalPanel {
        private TextBox display;
        public CalculatorWidget(final String title) {
                super();
                final CalculatorData data = new CalculatorData();
           final CalculatorController controller =
new CalculatorController(data);
//...
```

Within the first part of the CalculatorWidget class there are several important items. First of all, we're making use of some existing GWT client.ui classes to compose our view . Our entire class is, in fact, a subclass of VerticalPanel . This is significant in that we inherit all of the hierarchy of a GWT UI Widget automatically with this approach.

Directly extending a panel in this manner is simple (in this case, intentionally so), but it's also limiting. It's often better to use a GWT Composite. We're using direct subclassing here to keep things very basic, and we'll move on to Composite in due course.

In listing 1 we then go on to define a simple constructor, and within it we create an instance of CalculatorController . CalculatorController itself will be addressed in the section on controlling the action, after we complete the CalculatorWidget class. The controller we'll be using is a client-side component, which itself contains a reference to the data portion of our model, which is a single CalculatorData object. That takes us to part two of the CalculatorWidget class: layout and buttons. These view elements are displayed in listing 2.

Listing 2 CalculatorWidget.java, part two:

```
VerticalPanel p = new VerticalPanel();
p.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
p.setStyleName(CalculatorConstants.STYLE_PANEL);
Grid g = new Grid(4, 5);
g.setStyleName(CalculatorConstants.STYLE_GRID);
final Button zero = new ButtonDigit(controller, "0");
g.setWidget(3, 0, zero);
final Button one = new ButtonDigit(controller, "1");
g.setWidget(2, 0, one);
final Button two = new ButtonDigit(controller, "2");
g.setWidget(2, 1, two);
//. . .
final Button divide = new ButtonOperator(controller, new OperatorDivide());
```

```
g.setWidget(0, 3, divide);
final Button multiply = new ButtonOperator(controller, new OperatorMultiply());
g.setWidget(1, 3, multiply);
//. . .
final Button clear = new Button(CalculatorConstants.CLEAR);
clear.addClickListener(new ClickListener() {
                public void onClick(Widget sender) {
                                controller.processClear();
                }
        });
        clear.setStyleName(CalculatorConstants.STYLE_BUTTON);
        g.setWidget(2, 4, clear);
//...
```

In this second part of the CalculatorWidget class, we add components to our widget and add event handlers for those components. First, we create a GWT Grid , a component made of rows and columns that ultimately will be implemented as an HTML table. We'll use this to lay out our calculator buttons. Then we add our two different types of buttons: ButtonDigit and ButtonOperator . The digits on our calculator are implemented by our own class, ButtonDigit, which extends the GWT Button class. The operators are similarly implemented by our own ButtonOperator class. Once again, for the sake of brevity, we have not included the code for all the fairly self-explanatory buttons and operators.

We have yet to create ButtonDigit and ButtonOperator, which we'll do next, but the important thing to know at this point is that these classes both implement a ClickListener. This listener gets the control when the buttons are clicked, and it invokes a controller method. Listeners are common in component-oriented development and enable an event-driven approach. Various listeners, such as ClickListeners, OnChangeListeners, KeyboardListeners, and more are available in GWT. By using listeners in this way, the controller is notified to perform an appropriate action when any button or operator is pressed on our calculator.

> **About KeyboardListeners**
>
> In this example, we have not attached KeyboardListeners to our calculator, for brevity's sake. But, obviously, including them would be very useful for a calculator. They can be enabled by setting a default focus element and then attaching the listeners.

Finally, we also add a standard GWT Button for both our calculator's CLEAR and EQUALS functions. These buttons directly invoke methods on our controller instance without our own Button subclass. This is done with regard to CLEAR and EQUALS because these are special cases in terms of calculator buttons, as we shall see when we get to the controller in the next section. Before moving on, though, we need to implement the remainder of the CalculatorWidget, as shown in listing 3.

Listing 3 CalculatorWidget.java, part three:

```
// . . .

display = new TextBox();
data.addChangeListener(new CalculatorChangeListener() {
        public void onChange(CalculatorData data) {
                        display.setText(
                        String.valueOf(data.getDisplay()));
                        }
                });
        display.setText("0");
        display.setTextAlignment(TextBox.ALIGN_RIGHT);
        p.add(display);
        p.add(g);
        this.add(p);
        }
}
```

The final part of the CalculatorWidget class creates a GWT TextBox for our calculator's display area , and then adds that component and our Grid to an earlier created VerticalPanel, which is in turn added to the current instance of itself. This ensures that all of the items are visible when CalculatorWidget is used .

Getting back to event handling, we have also introduced a custom listener' our own CalculatorChangeListener interface . This is implemented as an anonymous inner class, and it connects our display widget (the view) to the data it represents (the model), through events.

These events are a manifestation of the Observer pattern. We'll now take a closer look at this relationship, because it's what facilitates the separate responsibilities in our MVC-powered GWT calculator.

Communicating by observing events

We need to briefly elaborate on the event handling we have just introduced via CalculatorChangeListener. The general pattern, Observer, is how our calculator's model and view will be connected. It's also how default GWT components themselves generally communicate.

In addition to MVC, many developers will be accustomed to event handling. Some non-GUI programmers may also have realized the power of this approach and may be using it on the server side. If you're already familiar with events, you may find our initial examples here primitive. This is because we're trying to simplify and demonstrate the concepts, rather than to create the most efficient or streamlined code. (For example, we're not using such standard Java event idioms as PropertyChangeSupport and related constructs.).

This approach should bring those who have not yet encountered event-based and asynchronous programming, especially in relation to a GUI, up to speed quickly. We'll bring the pattern, itself, to the forefront with our first manually driven example. Listening for, and notifying of, events are all aspects of the Observer pattern. Figure 3 illustrates this pattern.

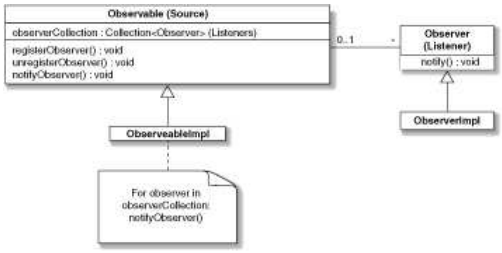


Figure 3 The Observer pattern, which is used in GWT to connect the model and view

In our example, the CalculatorChangeListener is used to register our view component (our CalculatorWidget itself) with the data portion of our model (CalculatorData). This is basically the first half of an observer/observable relationship between these components. Our change listener interface is shown in listing 4.

Listing 4 CalculatorChangeListener.java

```
public interface CalculatorChangeListener {
        public void onChange(CalculatorData data);
}
```

CalculatorChangeListener has a single callback method, onChange() . This is a fairly typical basic listener interface. Our CalculatorData model component, as we'll see in a bit, makes itself observable (available for listeners to listen to) by implementing another simple interface, CalculatorChangeNotifier. This change notifier interface is shown in listing 5.

Listing 5 CalculatorChangeNotifier.java

```
public interface CalculatorChangeNotifier {
        public void addChangeListener(
final CalculatorChangeListener listener);
}
```

CalculatorData, as shown in listing 6, therefore carries out the other half of the observer/observable relationship and notifies any listeners that come along when things change.

Listing 6 CalculatorData.java: the data portion of the model layer:

```
public class CalculatorData implements CalculatorChangeNotifier {
private String display;
private double buffer;
private boolean initDisplay;
private boolean lastOpEquals;
private CalculatorChangeListener listener;
public CalculatorData() {
        this.clear();
}
public void addChangeListener(
final CalculatorChangeListener listener) {
        this.listener = listener;
}
public void clear() {
        this.display = "0";
        this.buffer = 0.0;
        this.initDisplay = true;
        this.lastOpEquals = false;
        if (listener != null) listener.onChange(this);
}
public double getBuffer() {
        return buffer;
}
public void setBuffer(double buffer) {
        this.buffer = buffer;
        listener.onChange(this);
}
public String getDisplay() {
        return display;
}
public void setDisplay(String display) {
        this.display = display;
        listener.onChange(this);
}
//...
```

CalculatorData allows a single listener to register , and then, when its own setters are invoked, it notifies that listener of the change. Again, keep in mind that we have simplified the approach here. In the real world, as is the case with GWT's own components that use this same pattern, you will likely see more than one listener attached to an observable in a collection,

and hierarchies of specialized interfaces or abstract classes used for both observable and observer support (and you will need support to remove and clean up observers).

Along with the event mechanism, you probably have also noticed by now that the CalculatorWidget makes references to CalculatorConstants. This is a simple constants class that defines CSS styles and String constants such as: ADD, SUBTRACT, SQRT, and EQUALS. Now that we have our CalculatorWidget in place and our event handling taken care of, we'll go back to the remainder of our example. This means we need to address ButtonDigit and ButtonOperator, where weâ€™ll meet the other half of our model, the logic.

Operator strategy

As we saw in the code in the 'Creating a widget'□ section, CalculatorWidget makes use of several custom button types. The buttons of a calculator handle the numeric and operator input. Buttons are, of course, view components, but the buttons we'll create to fill these roles are also backed by the logic portion of our model, our calculator's operators. We have used our own specific Java types, ButtonDigit and ButtonOperator, so that we can easily distinguish the logical types of buttons pressed, and so that we can further encapsulate the logic for each operation. Listing 7 shows our ButtonDigit implementation.

Listing 7 ButtonDigit.java

```
public class ButtonDigit extends Button {
        public ButtonDigit(
          final CalculatorController controller,
          final String label) {
                super(label);
                this.addClickListener(new ClickListener() {
                                public void onClick(Widget sender) {
                                        controller.processDigit(label);
                                }
                        });
                this.setStyleName(CalculatorConstants.STYLE_BUTTON_DIGIT);
        }
}
```

ButtonDigit is a very straightforward extension of the GWT Button class. It simply includes a CalculatorController object reference in its constructor , and then invokes the controller's processDigit() method each time a ButtonDigit instance is clicked . This is a repeat of the pattern we're using in the outer CalculatorWidget, using the same controller reference. This invocation is achieved via a ClickListener.

The primary point in this example, especially for server-oriented developers, is that extending the basic UI classes is not only possible, but desirable. Using OOP techniques to extend functionality allows for much cleaner separation and reuse of code. This is in contrast to HTML-based development, where the <input type="button"> represents an opaque instruction. This fairly simple example just passes on a value, but you should note that the MVC pattern is present on multiple levels.

In listing 8 we see this pattern again with the ButtonOperator class. In this case, MVC comes into play on a micro level, while it's also used on a macro level with the entire outer Widget. ButtonOperator has a model, represented by the label; a view, managed by the parent class of Button, tweaked by the designation of a style to the button; and a controller layer represented by an AbstractOperator instance, which translates actions up to the larger scope controller for the calculator widget.

Listing 8 ButtonOperator.java

```
public class ButtonOperator extends Button {
        public ButtonOperator(final CalculatorController controller,
                final AbstractOperator op) {
                super(op.label);
                this.addClickListener(new ClickListener() {
                                public void onClick(Widget sender) {
                                        controller.processOperator(op);
                                }
                        });
                this.setStyleName(CalculatorConstants.STYLE_BUTTON_OPERATOR);
        }
}
```

ButtonOperator works in much the same way as ButtonDigit does. It extends the GWT Button class and includes a CalculatorController object reference in its constructor . Then it invokes the controller's processOperator() method each time a ButtonOperator instance is clicked . The difference with ButtonOperator is that it includes a reference to an AbstractOperator class in its constructor.

Now that we have operator buttons, we obviously need operations to back these view components. To complete these components, we'll use the Strategy pattern to encapsulate the logic. Rather than a monolithic if/else logic block, we'll delegate each operation to a specified instance of an operator class (each an implementation of AbstractOperator) and let those classes update our calculator's model. Using our operators in this manner makes our calculator easier to understand and allows for greater flexibility and extensibility. With this approach, we can add new operators later without affecting our existing logic. Figure 4 shows the Strategy pattern.

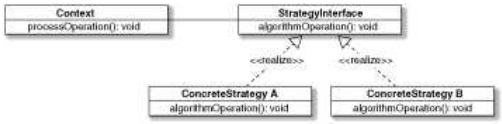


Figure 4 The Strategy pattern used in the GWT calculator example to implement operators

The only exceptions to this strategic structure for operators are the CLEAR and EQUALS operations, which are handled directly by the calculator controller, not by operator subclasses. Doing it this way requires less code and makes things a bit clearer. (We have some global state information in the controller that CLEAR and EQUALS rely upon. A purist implementation could put this information in our model and make separate operators for CLEAR and EQUALS as well, but that would complicate the other operations just for the sake of the pattern, and we think that would be overkill in this instance.)

These concepts may seem slightly off the beaten GWT path, but they serve to demonstrate the important point that you can use many OO design principles within the context of the toolkit. This flexibility is one of the advantages of GWT. In addition, this approach provides a much more robust calculator at the end of the day.

To handle the individual operators for our calculator, we need to implement the AbstractOperator and the subclasses that will extend it. As we saw in listing 8, ButtonOperator includes a reference to AbstractOperator. This is because each operator needs to do something different within a calculator application; each has a unique behavior. AbstractOperator, the straightforward beginning of the hierarchy, is shown in listing 9.

Listing 9 AbstractOperator.java

```
public abstract class AbstractOperator {
        public String label;
        AbstractOperator(final String label) {
                this.label = label;
        }
        public abstract void operate(
                final CalculatorData data);
}
```

AbstractOperator defines a single abstract method, operate(), which takes CalculatorData as input and updates it accordingly . We further divide operators with abstract types that determine whether or not the operator being implemented is binary or unary. BinaryOperator and UnaryOperator are as follows:

```
public abstract class BinaryOperator extends AbstractOperator {
        BinaryOperator(final String label) {
                super(label);
        }
public abstract class UnaryOperator extends AbstractOperator {
        UnaryOperator(final String label) {
                super(label);
        }
}
```

With our operator abstractions in place, we'll now implement a simple concrete operator for addition. (We'll not explicitly cover all the concrete operators used in our CalculatorWidget in the text for the sake of brevity). Listing 10 displays OperatorAdd.

Listing 10 OperatorAdd

```
public class OperatorAdd extends BinaryOperator {
        public OperatorAdd() {
                super(CalculatorConstants.ADD);
        }
        public void operate(final CalculatorData data) {
                data.setDisplay(String.valueOf(data.getBuffer() +
                           Double.parseDouble(data.getDisplay())));
                data.setInitDisplay(true);
        }
}
```

The binary addition operator adds the current buffer to the current display value, and then updates the display . Binary operators in this context are basically responsible for updating the display, based on the values in the CalculatorData object. Also, this operator sets the initDisplay status to true ; this indicates that, when the next digit is entered, the display should start over rather than append digits to any possible existing value. Now, with all of the other aspects of our CalculatorWidget in place, we can move on to the controller.

Controlling the action

The calculator's controller handles the interaction between the other calculator components. The controller is called upon by the various buttons to perform actions such as invoking an operator or otherwise manipulating the model or internal state.

The GWT CalculatorWidget controller component is significant for several reasons beyond just the separation of responsibilities derived from the MVC pattern itself. We'll write this component in Java, and it will be compiled into JavaScript by GWT along with our UI components, yet this has nothing to do with the interface. GWT allows you to create not only your UI but also any logic and data representations with client-side intentions. You can create classes, like we're about to do with

our CalculatorController and have already done with our CalculatorData, which end up in the client browser as completely non-view-related items.

In the CalculatorController, which is presented in listing 11, notice that there are no references to GWT-related classes.

Listing 11 CalculatorController.java

```
public class CalculatorController {
        CalculatorData data;
        AbstractOperator lastOperator;
        Private double prevBuffer;
        public CalculatorController(
                   final CalculatorData data) {
                this.data = data;
        }
        public void processClear() {
                data.clear();
                lastOperator = null;
        }
        public void processEquals() {
                if (lastOperator != null) {
                        if (!data.isLastOpEquals()) {
                                prevBuffer = Double.parseDouble(data.getDisplay());
                        }
                        lastOperator.operate(data);
                        data.setBuffer(prevBuffer);
                        data.setLastOpEquals(true);
                }
        }
        public void processOperator(
                                final AbstractOperator op) {
                if (op instanceof BinaryOperator) {
                        if ((lastOperator == null) || (data.isLastOpEquals())) {
                                data.setBuffer(Double.parseDouble(data.getDisplay()));
                                data.setInitDisplay(true);
                        } else {
                                lastOperator.operate(data);
                        }
                                lastOperator = op;
                } else if (op instanceof UnaryOperator) {
                        op.operate(data);
                }
                data.setLastOpEquals(false);
        }
        public void processDigit(final String s) {
                if (data.isLastOpEquals()) {
                        lastOperator = null;
                }
                if (data.isInitDisplay()) {
                        if (data.isLastOpEquals()) {
                                data.setBuffer(0.0);
                        } else {
                                data.setBuffer(Double.parseDouble(data.getDisplay()));
                        }
                        data.setDisplay(s);
                        data.setInitDisplay(false);
                } else {
                        if (data.getDisplay().indexOf(
                                   CalculatorConstants.POINT) == -1) {
                                data.setDisplay(data.getDisplay() + s);
                        } else if (!s.equals(CalculatorConstants.POINT)) {
                                data.setDisplay(data.getDisplay() + s);
                        }
                }
                data.setLastOpEquals(false);
        }
}
```

CalculatorController is instantiated with a new instance of CalculatorData . Along with the data object member, our controller also has four main actions: equals() and clear(), which are used for the calculator operations of the same names, and processOperator() and processDigit() , which are used when their respective operator buttons are clicked.

Controller actions, such as the pressing of a digit or an operator, update the model, which fires the corresponding events. When the data in the model changes, in response to actions from the controller, components that are registered to listen will be notified. Our view, CalculatorWidget, is one such component. The controller also contains a reference to the lastOperator so that it can make decisions about what to do in the chain of operations it provides. Figure 5 shows an overview of the classes involved in our now-complete MVC-enabled calculator project.

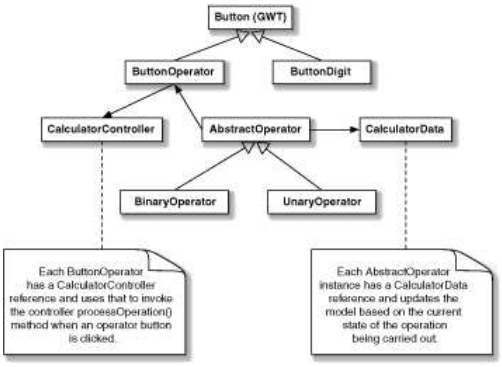


Figure 5 Overview of the classes involved with the GWT calculator example

This use of CalculatorController allows our client-side application, all by its lonesome, to handle state, logic, delegation, and the use of a separate model, along with the view. It's important to remember this expression of the MVC pattern, and where the responsibilities lie. Java developers accustomed to the request-response cycle will be used to the view being rendered in a single, generally procedural action based on a single-state model. In GWT, as in well-designed desktop applications, the model can change independently of a singular view layer. This is notably different from a more transient view that performs a single operation and then goes away. Dealing with the state of the view, not just of the model, is something you will need to keep in mind during your development.

Now we have an event-driven GWT calculator in the form of a reusable CalculatorWidget. In addition, our code is fairly resilient to change and is extensible because we have used an OO approach with operators responsible for their own individual operations.

> This article is excerpted from Chapter 1 of GWT in Practice, by Robert Cooper and Charlie Collins, and published in May 2008 by Manning Publications.

Source URL: http://css.dzone.com/news/design-patterns-and-gwt

Links:
[1] http://www.manning.com/affiliate/idevaffiliate.php?id