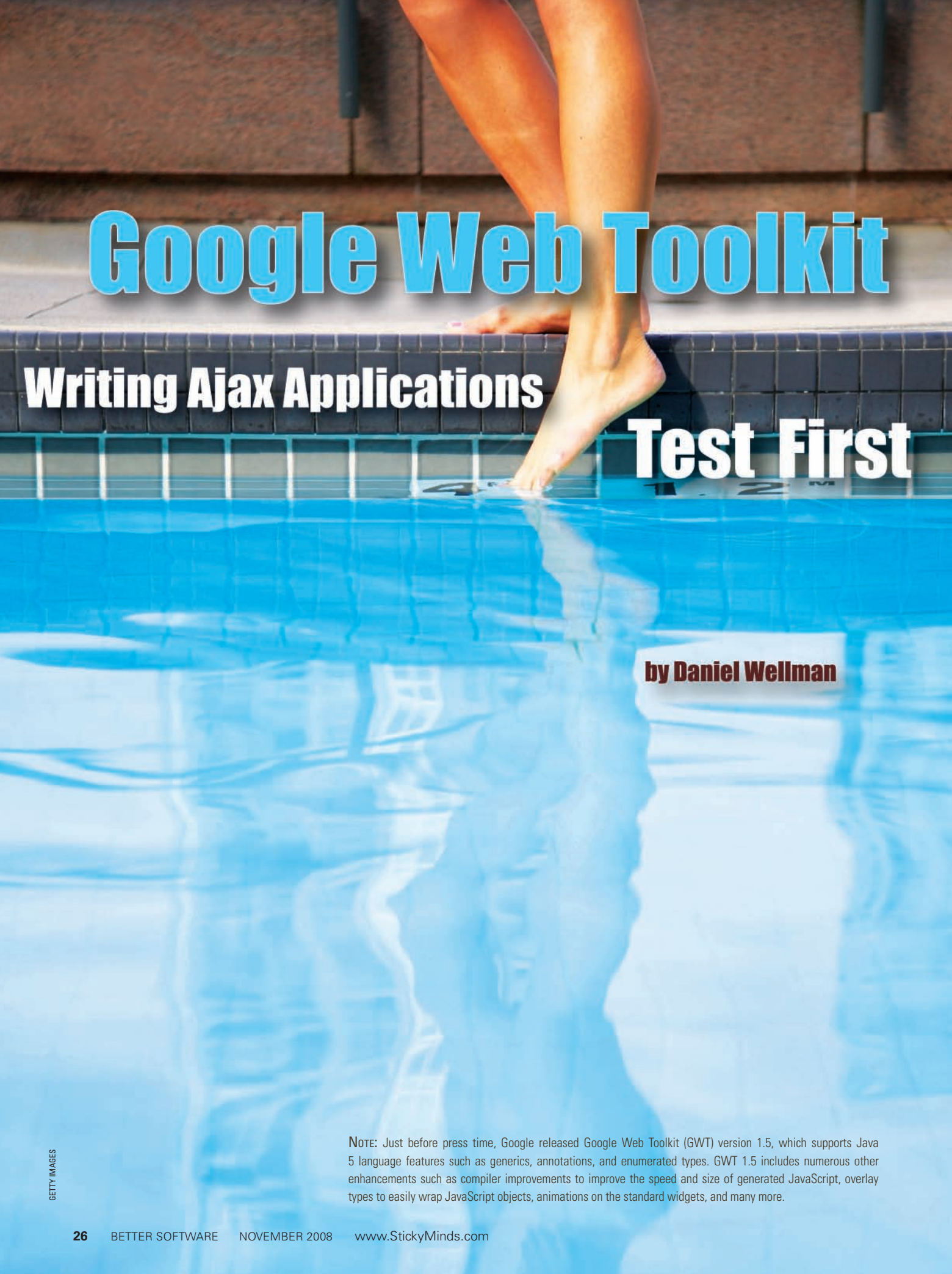

Google Web Toolkit

Writing Ajax Applications Test First

by Daniel Wellman

GETTY IMAGES

NOTE: Just before press time, Google released Google Web Toolkit (GWT) version 1.5, which supports Java 5 language features such as generics, annotations, and enumerated types. GWT 1.5 includes numerous other enhancements such as compiler improvements to improve the speed and size of generated JavaScript, overlay types to easily wrap JavaScript objects, animations on the standard widgets, and many more.

Rachel had successfully built her Web 2.0 Ajax application using Google Web Toolkit, and her boss and customers were ecstatic. Revenue had increased, which led to a steady stream of new business requirements.

But there was a catch: Bugs appeared, got fixed, and mysteriously reappeared. Rachel found her team spending an increasing amount of time fixing bugs while new feature development slowed to a crawl. She remembered that test-driven development was a great way to build applications in a manner that prevented defects from reappearing. She knew that Google Web Toolkit had been built with testability in mind, so Rachel downloaded JUnit and started writing tests ...

In the October 2008 issue of *Better Software* magazine, part one of this series introduced the Google Web Toolkit (GWT), a tool for building cross-browser Ajax applications written in Java. GWT compiles Java code into JavaScript and provides a component library that is cross-browser compatible and memory-leak proof. This means that you can focus on writing your application business logic instead of handling the accidental complexity of supporting multiple browsers.

Another core feature of GWT is *testability*, which means it's easy to unit test your application. This makes it possible to write GWT applications *test first*—an agile practice that helps build reliable and extensible applications. This article introduces GWT's testing infrastructure and demonstrates how to build an Ajax application test first.

GWT's Testing Infrastructure

Since a GWT application is almost entirely written in Java, you can test almost all of it using standard JUnit tests. However, GWT also includes a special subclass of JUnit's `TestCase` that can test code that requires JavaScript at run time. While all of your client-side Java code will ultimately be compiled to JavaScript, only some of it directly uses code implemented as JavaScript. For example, the code in listing 1 is from the GWT `HTMLTable` class.

This code sample demonstrates a method written in Java (`setStylePrimaryName`) that relies on code implemented directly in JavaScript, as indicated by the keyword "native" in the definition of `getCellElement`. As shown in listing 1, many of the GWT libraries include some native code—in particular, all widgets that manipulate the Domain Object Model (DOM). Thus, when your unit tests execute native JavaScript, you must be running in an environment where it can be executed, such as the hosted-mode browser provided by GWT.

To test native JavaScript code, GWT provides a subclass of JUnit's `TestCase` called `GWTTestCase`. This base class allows you to implement your JUnit test case as you normally would. In fact, `GWTTestCases` look almost identical to the standard JUnit `TestCase` shown in listing 2.

The only visible difference is that all `GWTTestCases` must override an abstract method called `getModuleName`, which returns a string containing the name of your GWT code module as defined in your application's module configuration XML file. (GWT applications can be grouped into reusable modules, each of which contains an XML file descriptor with information including source code location, dependencies, target browsers, etc.)

When you run your test, the GWT framework starts up an invisible (or "headless"), hosted-mode browser and then evaluates your test case against it. What this means is that all the facilities of the hosted browser are available to your test case. You can run native JavaScript functions, render widgets, or invoke asynchronous remote procedure calls. Furthermore, you can run your tests either as a hybrid of Java and JavaScript

```
public void setStylePrimaryName(int row, int column, String styleName) {
  UIObject.setStylePrimaryName(getCellElement(bodyElem, row, column),
                               styleName);
}

private native Element getCellElement(Element table, int row, int col) /*-{
  var out = table.rows[row].cells[col];
  return (out == null ? null : out);
}-*/;
```

Listing 1

```
public class MeetingSummaryLabelTest extends GWTTestCase {

    public String getModuleName() {
        return "com.danielwellman.booking.Booking";
    }

    // Add tests here
}
```

Listing 2

code ("hosted mode") or compile and run all your Java code as JavaScript ("Web mode"). The GWT team recommends that you run your tests both in hosted mode and Web mode, since there are a few subtle differences between Java and JavaScript [1], which could cause unexpected behavior.

Being able to test native JavaScript code in your Java JUnit tests is great, but there are some limitations. First, the normal browser-event mechanisms don't work as expected in test mode. For example, you can't programmatically click a button and expect the corresponding event handlers, such as `onClick()`, to fire. Selenium [2], the open source testing tool, can control a real browser and is a helpful alternative in this situation.

There are also performance considerations; the tests are slower than standard JUnit `TestCases`. Running a `GWTTestCase` forces a compilation of the source code in your module, which incurs an initial startup delay. Furthermore, each individual test method is wrapped by logic that starts up and shuts down the headless browser, which can take several seconds. Some testers would call these integration tests, not unit tests, since they involve other systems, cross language boundaries, and are slow to execute.

So when should you extend a standard JUnit `TestCase` vs. a `GWTTestCase`? In general, you should prefer standard JUnit `TestCases` because they run orders of magnitude faster than a `GWTTestCase`. If your code executes native JavaScript, however, or uses the libraries supplied with GWT, then your test must extend `GWTTestCase`. The upshot is that even if you simply instantiate a widget in the code being tested, you will have to test this using a `GWTTestCase`. You might try to find another design approach that avoids this native code requirement, such as moving the logic to another class.

GUI Design Patterns

To build a testable GUI application, there are several design patterns and techniques you can use. All of them focus on one core principle: Move as much logic as possible out of the view and into other, more easily testable layers. One common pattern is known as Model-View-Presenter, where a presenter object acts as a mediator between the view (GUI) and model objects and instructs the view layer to change states in response to user input or model changes. Martin Fowler has described a few variants of this pattern in his bliki [3], including Supervising Controller and Passive View. Both patterns push all logic- and event-handling code into the presenter, but they differ in how much the view knows about the model. Presenters hand the model objects directly to the view in the Supervising Controller pattern; then the view picks the appropriate information to display. In Passive View, the view layer knows nothing about the model objects, and the presenter communicates model details to the view in terms of primitives, such as strings and numbers. Michael Feathers's paper "The Humble Dialog" [4] provides a good introduction to the subject, and Martin Fowler's bliki is a good resource for the other variations.

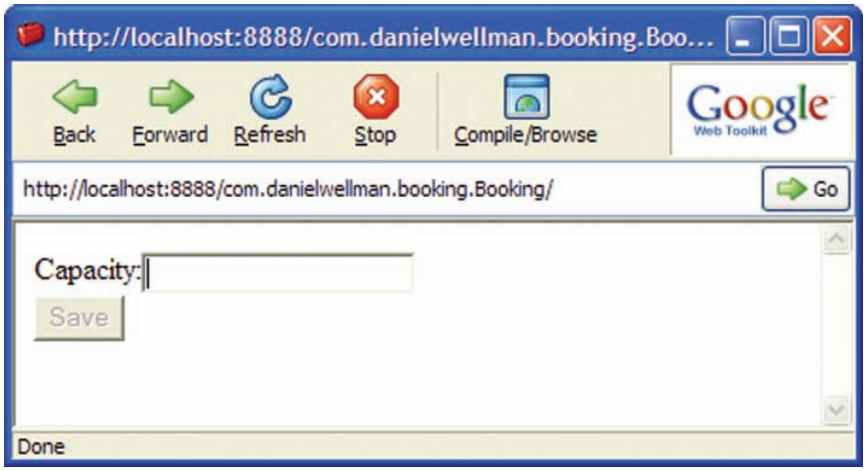

Figure 1: The first iteration of the UI for the booking application

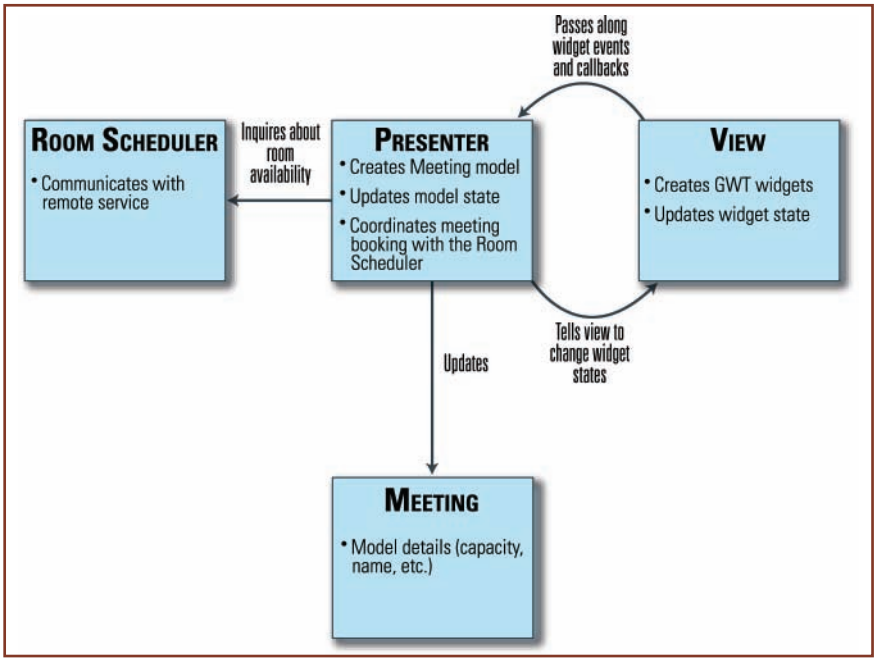

Figure 2: Object responsibilities and interactions for the booking application

Example

To illustrate some of these concepts, let's take a look at building a small portion of an application. For this example, suppose we're building an online application for booking meeting rooms at a conference center. A user will need to specify some details about the meeting,

```
@RunWith(JMock.class)
public class PresenterTest {

      Mockery context = new Mockery();

      @Test
      public void anUnavailableRoomDisablesTheSaveButton() {

            final MeetingView view = context.mock(MeetingView.class);
            final RoomScheduler scheduler = context.mock(RoomScheduler.class);

            final Meeting meeting = new Meeting();
            final Presenter presenter = new Presenter(meeting, view, scheduler);

            // The schedule service will reply with no available capacity
            context.checking(new Expectations() {
                  {
                        allowing(scheduler).canAcceptCapacityFor(meeting);
                        will(returnValue(false));

                        one(view).disableSaveButton();
                  }
            });

            presenter.requiredCapacityChanged(new FakeTextContainer("225"));

            assertEquals("Should have updated the model's capacity", 225,
                        meeting.getCapacity());
      }
}
```

Listing 3

including the expected capacity and date. The application will check with a scheduling back-end service to determine if the room is available. If it's not available, the Save button will dim and a message will be displayed. See figure 1 for a sample layout of this dialog.

After some quick drawing at a whiteboard, we come up with a rough sketch of the objects involved, as shown in figure 2.

Building the Presenter

The key to testing presenters is to keep in mind that they are plain old Java code and can be tested like any other Java code with JUnit. A mock-object library like JMock [5] can be used to test the interactions between the presenter and the view components.

Let's tackle a small slice of the following functionality: The user enters a meeting capacity that cannot be scheduled. First, the view will notify the presenter that the user changed the value of the capacity text field. The presenter will then ask the RoomScheduler service if it can accept a new meeting with the specified capacity. Finally, the presenter will tell the view to disable the Save button. Listing 3 shows a test for this scenario.

This is an interaction-based test using JMock to provide test doubles for the MeetingView and the RoomScheduler. We stub out the scheduler to reply that it cannot accept the capacity for the meeting and expect our view to be told to disable the Save button. Note here that the view ends up being fairly dumb; it does nothing but notify the presenter whenever the required capacity is changed.

This code requires that we specify an interface for our view:

```
public interface MeetingView {
    void disableSaveButton();
}
```

and for our service:

```
public interface RoomScheduler
{
boolean canAcceptCapacityFor(
            Meeting meeting);
}
```

The code that passes this test is fairly simple, as shown in listing 4.

The presenter is responsible for orchestrating the call to the remote service and instructing the view to disable the Save button. Note also that we're choosing to let the presenter maintain the state of the Meeting object, so that all UI events ultimately modify this object.

This is a very simple implementation, but it's far from the completed design. Our next test would probably check that setting an acceptable capacity enables the Save button and drives us to make either a new `enableSaveButton` method or a generalized `setSaveButtonAvailable` method on the view. We're still testing

```
public class Presenter {
    private Meeting meeting;
    private MeetingView meetingView;
    private RoomScheduler roomScheduler;

    public Presenter(Meeting meeting, MeetingView meetingView, RoomScheduler
roomScheduler) {
        this.meeting = meeting;
        this.meetingView = meetingView;
        this.roomScheduler = roomScheduler;
    }

    /**
     * Callback when the view's capacity text box changes
     *
     * @param textField the capacity TextBox widget
     */
    public void requiredCapacityChanged(HasText textField) {
        meeting.setCapacity(Integer.parseInt(textField.getText()));
        if (!roomScheduler.canAcceptCapacityFor(meeting)) {
            meetingView.disableSaveButton();
        }
    }

    protected Meeting getMeeting() {
        return meeting;
    }
}
```

Listing 4

```
package com.google.gwt.user.client.ui;

public interface HasText {

  /**
   * Gets this object's text.
   */
  String getText();

  /**
   * Sets this object's text.
   *
   * @param text the object's new text
   */
  void setText(String text);
}
```

Listing 5

```
public class FakeTextContainer implements HasText {
    private String text;

    public FakeTextContainer(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

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

Listing 6

plain Java objects that don't require any JavaScript, so these tests run quickly.

Note the argument to `requiredCapacityChanged` is of the type `HasText`. This turns out to be an interface that is part of the GWT libraries, as shown in listing 5.

This simple interface is used by many GWT components and allows manipulation of a widget's text contents, including the TextBox in our example. This interface is extremely useful for testing because we don't need to pass in a real TextBox. Thus, we avoid instantiating a text input in the DOM, requiring our test to extend `GWTTestCase` to run in a real browser. In listing 6, I've made a simple, fake implementation that wraps a string.

And finally, the view implementation is shown in listing 7.

As you can see, there's not much logic here. Most of the code is involved in setting up the event listeners and config-

```
public class MeetingViewWidget extends Composite implements MeetingView {
    private Button saveButton = new Button("Save");
    private TextBox capacityText = new TextBox();

    public MeetingViewWidget() {
        VerticalPanel mainPanel = new VerticalPanel();

        HorizontalPanel row = new HorizontalPanel();
        row.add(new Label("Capacity:"));
        row.add(capacityText);

        mainPanel.add(row);
        mainPanel.add(saveButton);

        // Start with the save button disabled
        saveButton.setEnabled(false);

        // Here the view is responsible for creating the model and presenter
        final Presenter presenter = new Presenter(new Meeting(), this,
                                                  new RemoteRoomScheduler());
        capacityText.addChangeListener(new ChangeListener() {
            public void onChange(Widget sender) {
                presenter.requiredCapacityChanged((HasText) sender);
            }
        });

        initWidget(mainPanel);
    }

    public void disableSaveButton() {
        saveButton.setEnabled(false);
    }
}
```

Listing 7

```
public AlternatePresenter(Meeting meeting, MeetingView meetingView,
                          RoomScheduler roomScheduler) {
        this.meeting = meeting;
        this.meetingView = meetingView;
        this.roomScheduler = roomScheduler;
        // Register to receive all widget callbacks to this presenter
        meetingView.registerPresenter(this);
}
```

Listing 8

uring the display widgets. So how do we test it in a `GWTTestCase`?

We don't. In fact, there's not much here that can be tested in an automated test. As stated earlier, event propagation won't work by default in a `GWTTestCase`, and the layout of widgets is often best checked visually. If you are building a widget library, then you might want to write `GWTTestCases` that test the widget through its API, which is what Google does with the widgets included in GWT, such as Button, TextBox, and Tree. However, these tests are slow (a sample widget test takes twelve seconds on my two-year-old workstation), and any complex logic could be moved into a simple presenter object, which could be tested in a plain old, fast JUnit `TestCase`. For more ideas for testing GWT widgets, see the StickyNotes for a link to my blog post on the subject.

Note here that the view is instantiating the model and presenter objects, which is one way of ensuring that the presenter is instantiated with a "live" view. You also could have some higher-level application object construct the view and pass it to the presenter, which would then need to register itself with the view so all the controls know where to send their events. This would look something like listing 8.

Testing Asynchronous Access to Remote Services

GWT provides a remote procedure call (RPC) mechanism that enables passing Java objects between the server and client using a server-side serialization library. `GWTTestCase` supports testing of these features by providing utility methods that facilitate writing asynchronous tests. Most of the infor-

mation available on `GWTTestCase` focuses on these RPC cases, and I recommend reading it for the full story. For a brief introduction, refer to the GWT documentation page titled "JUnit Integration" in the section "Asynchronous Testing" or, for a deeper example, review the book *GWT in Practice* [6].

Reflecting on this Design Approach

My team used the design approach I've described in this article on a project and found it worked well. A disadvantage of this design is that it relies on the views' correctly registering the callback events with the presenter. Since this logic was almost too simple to break, we accepted these limitations on our project. For end-to-end integration tests, we used Selenium to control an instance of Firefox and Internet Explorer. These tests filled in the cracks to ensure we had widgets properly wired into their corresponding presenters.

Testing GWT applications, like testing Swing or other desktop client applications, can be fairly succinctly summarized as follows: *Don't put logic in your view components.* If you find complicated logic in your view, see if it can be moved into the model objects or the presenters. When you need to test view component behavior, JavaScript, or remote server communication, use a `GWTTestCase`. If you have too many slow `GWTTestCases`, see if there's some design change that doesn't require testing inside a real browser. Let the tests help guide your design, and you'll be on your way toward making Ajax development fun and relatively painless. **{end}**

References:

[1] "Compatibility with the Java language and libraries" in the Google Web Toolkit online documentation. tinyurl.com/6d4tg6

[2] Selenium. selenium.openqa.org/

[3] Fowler, Martin. martinfowler.com/eaaDev/uiArchs.html

[4] Feathers, Michael. "The Humble Dialog Box." Object Mentor, inc., 2002. www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf

[5] JMock. jmock.org/

[6] Cooper, Robert and Collins, Charlie. *GWT in Practice.* Manning Publications, 2008.

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- Testing GWT widgets
- Infrastructure tips
- More information
- GWTTestCase gotchas