

Automated Functional Testing of an ExtJS UI using Selenium RC in Java

By Lindsay Kay
Septemb2009

In this article I'm going to describe a Java framework I created at SMX (<http://www.smx.co.nz>) on top of Selenium RC for the purpose of performing automated functional testing of a complex rich user interface built with ExtJS (<http://www.extjs.com>).

Automated testing of an application that has an [ExtJS](#) user interface can be tricky, due to Ext's AJAX asynchronism, complex DOMs and randomly-generated default element IDs. However, our framework works around these difficulties effectively. In fact, our test suite has been easy to maintain within our agile process, and has recently proven itself during a major refactoring.

I'm going to assume that you have experience in Java, Selenium, ExtJS and TestNG. Although I won't publish the complete source code, which is the property of SMX, I'll share with you the essential techniques, which I hope you will find useful.

Overview

For easy maintenance, our test code has three layers:

1. **Test** – contains TestNG test classes,
2. **Action** – provides logical user operations on the application, and
3. **Document** – a proxy to the application user interface.

The **Document** layer contains a Java proxy for each Ext component type that we use, and we assemble these into containment hierarchies that mirror those within the UI. They have various methods that drive a Selenium proxy (a **com.thoughtworks.selenium.Selenium**) to fire events and queries at the DOM elements generated by their corresponding Ext components. The **Action** layer provides facade classes providing logical user actions that drive proxies in the Document layer, encapsulating the latter. Whenever we change UI layouts, the API of the Action layer tends to remain largely unchanged. The TestNG classes in the **Test** layer drive the facades in the Action layer, organising their logical operations into test cases. Thanks to the Actions layer, code at this level tends to be fairly intention-revealing with respect to use cases.

Now I'll describe these layers in more detail, working upwards, with a little background on how we arrived at the present design.

Document Layer

To interact with DOM elements, test code must fire Selenese commands at them, specifying the target elements with XPath's. A typical path might be:

```
click //div[5]//div[2]/td[@class="foo"]
```

The primary challenge was to determine these XPath's. We could use Selenium IDE to record a manual click, then cut and past the XPath into our test code, however the path

would be fragile, where we would need to repeat that process every time the path changes after a UI modification. However, if we know what ID Ext assigns to the element, then the element's XPath is trivial and stable. For example, if the UI has an Ext.Button and we know its ID is 'ext-gen1048', then the path could be something like:

```
//*[@id='ext-gen1048']
```

However by default, Ext assigns its own randomly-generated Ids. Since they change each time Ext renders the components, recording the XPaths with Selenium IDE and pasting into test code is futile.

We could have Ext assign our own non-random IDs, but we would need to share some kind of complicated hierarchical ID schema between the UI and test code, which would create a maintenance bottleneck.

Initial Technique: Finding DOM Elements with Structural XPaths

Initially, I took a close look at the HTML generated by Ext components and implemented for each component a Java proxy that had a portion of XPath, so that when the proxies were plugged together to mirror the UI hierarchy, the concatenated portions drilled down to resolve the DOM elements.

For example, an Ext.Window proxy had a portion like this:

```
//div[starts-with(@id, 'LoginWindow') and starts-with(@class, 'x-window')]
```

Note that we did configure Ext.Windows with IDs. An Ext.Button proxy had a portion like this:

```
//button[text()='Login']
```

so when the Button proxy is connected as a child of the Window proxy, the absolute XPath to the Ext.Button's DOM element is

```
//div[starts-with(@id, 'LoginWindow') and starts-with(@class, 'x-window')]//button[text()='Login']
```

The proxy classes shared a `com.thoughtworks.selenium.Selenium` and provided methods that would delegate to it. For example, the Button proxy had a click method:

```
class Button ... {
    ...
    private String getAbsolutePath() {
        return parent.getAbsolutePath() +
"//button[text()='Login']"; // Concatenate XPaths
    }

    public void click() {
        this.selenium.click( getAbsolutePath() );
    }
}
```

```
    ...  
}
```

One benefit was that we could encapsulate work-arounds for browser weirdness within the proxies. For example, some of them had switch statements to select variations of XPath expressions or events for different target browsers. Internet Explorer 7, for example did not support clicks on Ext tabs correctly, so a special work-around event was encapsulated within our Tab proxy for that target.

Theoretically, this technique was supposed to push XPath expressions down into the proxy layer once and for all, out of sight and mind, but in practice they required constant maintenance. Sometimes they ended up accidentally ambiguous, resolving to the wrong DOM element. Being snaky and complex, they also tended to break whenever the UI code changed or a new release of Ext was used. Internet Explorer 7 often reared its ugly head, with little failures such as not being able to reliably determine if elements on XPath expressions were visible or not.

With effort, the tests were maintainable, but too much of my time went into fixing broken XPath expressions. Also, when a test failed I had to laboriously verify a bunch of XPath expressions before actually raising a ticket against the application.

A Better Technique: Java Proxies containing JavaScript to Evaluate the IDs of their Ext Components

Recall that if we have the ID of a DOM element then we can form an unambiguous XPath expression that points directly to it. Since we can get the element ID from the corresponding Ext component, the winning technique was to have **Java proxies for Ext components** that contain portions of **JavaScript**. When these are plugged together they form absolute JavaScript expressions that they can evaluate through a `com.thoughtworks.selenium.Selenium` to obtain their Ext components, from which they can then get IDs and voila, have robust XPath expressions to their component's DOM elements. No need to assign IDs to all those Ext components, just a few easy ones like Windows and Fields.

It was tempting to have the proxies drive their Ext components purely through calls to their components' JavaScript functions, but that would not be testing the actual user interface. We do that sparingly, such as when browsers do not support certain Selenium events on certain HTML elements.

To synchronise test execution with asynchronously-loading AJAX widgets, the proxies repeatedly attempt to evaluate JavaScript expressions until they either succeed or time out. When a test creates a proxy, it is usually making an implicit assertion that the UI contains the corresponding widget. A test failure due to one of these timeouts therefore tends to imply that the UI is not in the expected state.

Sometimes a proxy asserts the *absence* of an Ext component or something in the DOM by asserting that a repeatedly-executed JavaScript expression, XPath or Selenese command will fail to resolve it within a certain time limit. This gives the target element a chance to disappear.

A skeleton of the basic component proxy class is shown below. This is subclassed for each Ext component type. A root proxy gets a `com.thoughtworks.selenium.Selenium`, which is shared among all sub-proxies for them to fire Selenese commands at their Ext components or DOM elements, and an absolute JavaScript expression to resolve its Ext component. Non-root proxies get a parent proxy, and a relative JavaScript expression that will be concatenated to that of the parent. The non-root proxies get their Selenium from their parents.

```
public class Component {
```

```

private Component parent;
private Selenium selenium;
private String expression;

/** Makes a proxy for a root Ext component;
 * selenium - Selenium proxy through which it can fire
Selenese commands,
 * expression - JavaScript that evaluates to the Ext
component.
 */
public Component(Selenium selenium, String expression) {
    this.parent = null;
    this.selenium = selenium;
    this.expression = expression;
}

/** Makes a proxy for an Ext component that is contained
within another;
 * parent - proxy for the container Ext component,
 * expression - JavaScript expression that evaluates this
proxy's component on
 * that of the container.
 */
public Component(Component parent, String expression) {
    this.parent = parent;
    this.selenium = parent.selenium;
    this.expression = expression;
}

/** Returns the ID of the Ext component, found with the
proxy's JS expression.
 * This is overridden in some
 * subclasses for where the expression to get the ID
varies.
 */
public String getId() {
    return selenium.getEval(this.getExpression() +
".getId()");
}

/** Returns an XPath to the Ext component, which contains
the ID provided by getId()
 */

```

```

public String getXPath() {
    return "//*[@id='" + getId() + "']";
}

/** Returns the absolute expression that resolves this
proxy's Ext component.
*/
public Expression getExpression() {
    return (parent != null) ? parent.getExpression() +
expression : expression;
}

//-----
----
// Methods to synchronise with AJAX
//-----
----

// Returns true as soon as expression evaluates, else
false on timeout
protected boolean waitUntilExpressionResolves(String expr)
{ .. }

// Returns true as soon as expression fails to resolve,
else timeout exception
protected void waitUntilExpressionNotResolves(String expr)
{ ... }

// Evaluates expressions and returns result.
protected String getEval(String expr) { ... }

// Immediately evaluates expression
protected void eval(String expr) { ... }

// Returns as soon as expression evals, else throws
exception on timeout
protected void waitForEvalTrue(String
expr) { ... }

//-----
-----
// Convenience methods to evaluate properties of the Ext
component.
//-----

```

```

-----

protected String getEvalStringProperty(String name) { ...
}
protected boolean getEvalPropertyExists(String name) { ...
}
protected boolean getEvalBooleanProperty(String name) {
... }
protected int getEvalIntegerProperty(String name) { ... }
protected double getEvalDoubleProperty(String name) { ...
}
}

```

Shown in the proxy above is one method, *close*, which closes the Ext.Window. That method obtains the XPath to the window's primary DIV element and fires a click through the Selenium at the close button, found relative to the primary DIV element. Recall that the XPath is generated within the Component base class from the ID of the Ext.Window, which is obtained through evaluation of the proxy's JavaScript expression. Note that the click method first logs the action. I prefix the log messages with the path of names of proxy classes leading down to this proxy in the containment hierarchy, which provides a nice trace of actions performed through the proxies.

Here's a proxy for an Ext.Button:

```

public class Button extends Component {

    public Button(Component parent, String text) {

        super(parent, ".findBy(function(component) {"
            + "return (component.isXType &&
component.isXType('button'))"
            + "&& (component.text && component.text == '" + text
+ "')"
            + "})[0]");
    }

    public boolean isEnabled() {
        return ( ! getBooleanEval("disabled"));
    }

    public void click() {
        log("click()");

        waitForEvalTrue(".disabled == false"); // Throws
exception on timeout

        getSelenium().click(getXPath());
    }
}

```

```
    }  
}
```

The proxy has a JavaScript expression to find the Ext.Button within it's container. Note how the click method waits for the Ext.Button to become enabled (if it is disabled) before firing a Selenese "click" at the HTML element.

As I mentioned above, logging at proxies is prefixed with the location of the proxy in the hierarchy. This click will log something like:

```
window["new.User"].button["Save"].click()
```

where the name of each path element is the proxy class name with the first letter decapitalised. The test report contains sequences of entries like this, terminated by stacktraces wherever a failure occurs. They can also be quickly scanned to verify test coverage.

To further illustrate expressions, the proxy for an Ext.Toolbar.Button is shown below. One of its constructors initialises the proxy as a child of an Ext.Tab proxy. An Ext.Tab holds an Ext.Toolbar.Button in an **items** array belonging to an Ext.ToolBar child component, so that constructor defines for the proxy a JavaScript expression to find the Ext.Tab accordingly.

```
public class ToolbarButton extends Component {  
    public ToolbarButton(Tab parent, String text) {  
        super(parent,  
            ".getTopToolBar().items.find(function(component)  
{" +  
            " return (component.isXType &&  
component.isXType('button'))" +  
            " && (component.text && component.text == '" +  
text + "');" +  
            " })[0]";  
        }  
        ....  
    }  
}
```

Domain-Specific Proxies

I sub-class the proxy classes to form domain-specific proxies with names that give some affordance to what they represent within the application. In our application there is the "New User" window, in which an administrator can create users. For that window I have the NewUserWindow proxy, which is both the container and factory for child proxies:

```
public class NewUserWindow extends Window {  
    public NewUserWindow(CSelenium selenium) {  
        super(selenium, "new.User");  
    }  
  
    public TextField getFirstNameField() {  
        return new TextField(this, "person.firstName");  
    }  
}
```

```

    }

    public TextField getLastNameField() {
        return new TextField(this, "person.lastName");
    }

    public TextField getEmailField() {
        return new TextField(this,
"primaryEmailAddress.emailAddress");
    }

    public TextField getPhoneNumberField() {
        return new TextField(this,
"primaryPhoneNumber.contactNumber");
    }

    public CreateButton getSaveButton() {
        return new Button(this, "Save");
    }

    public Button getCancelButton() {
        return new Button(this, "Cancel");
    }

    public NewUserWindow populateWith(User user) {
        getFirstNameField().setValue(user.getFirstName());
        getLastNameField().setValue(user.getLastName());
        getEmailField().setValue(user.getEMail());

        getPhoneNumberField().setValue(user.getPhoneNumber());
        return this;
    }
}

```

Note the populateWith method, which populates the window's widgets with the contents of a User data bean. Also, note that we manually specify Ext Ids for our Ext.Windows. Our application also has a launcher across the top of the desktop area, from where one can create resources, such as users. Below is our proxy for the launcher, which extends a Toolbar proxy and provides ToolbarButton proxy:

```

public class DesktopLauncher extends Toolbar {
    public DesktopLauncher(Selenium selenium) {
        super(selenium, "Ext.getCmp('desktop-launcher')");
    }
    ....
}

```

```

public ToolbarButton getNewUserButton() {
    return new ToolbarButton(this, "New User");
}
....
}

```

Actions Layer

The Actions layer provides a facade of logical actions on top of the Document Proxy Layer, hiding the latter. It is driven by the Test Layer, providing a level of indirection so that the component proxies can be refactored with minimal impact on test code. It also allows logical actions to be reused among different tests. For example, the actions for testing CRUD on one resource can be reused to set up fixtures for CRUD tests on another dependent resource.

```

public class UserExplorerActions {

    private Selenium selenium;

    public UserExplorerActions(Selenium selenium) {
        this.selenium = selenium;
    }
    ...

    public UserExplorerActions create(User user) {
        (new DesktopLauncher(this.selenium))
            .getNewUserButton()
                .click();

        (new NewUserWindow(this.selenium))
            .populateWith(user)
                .getSaveButton()
                    .click(user);
        return this;
    }
    ...
}

```

Tests Layer

At this layer we have TestNG classes which organise logical actions provided by the Actions Layer into tests. Here's the user CRUD test, showing just the user creation part that uses the action described above:

```

@TestState(state="userCrud", dependsOnState =
"startApplication")
public class UserCrudTest extends SMXTestBase {
    ...
    @Test(groups="selenium", dependsOnGroups = "system")
    public void testUserCrud() {
        User user = new User( ... );
        UserActions actions = new
UserActions(this.getSelenium());

        actions.createUser(user);
        ...
    }
    ...
}

```

Note that at the test level:

- We're thinking purely in terms of *use cases*, in this example 'create a user'
- All the UI structure has been abstracted away, revealing more of the business domain in the tests (see Martin Fowler's BusinessReadableDSL at <http://www.martinfowler.com/bliki/BusinessReadableDSL.html>)
- We can re-use the same actions in various tests.

See it on YouTube

Selenium tests are fun to watch. Hide them from your co-workers if you want them to get any work done. I posted a movie of our framework running user account CRUD tests at YouTube:

<http://www.youtube.com/watch?v=sDdaNEXxzMs>

Further reading

Patrick Lightbody, one of the Selenium developers, used some of these very techniques in testing the Nexus Maven repository manager. Read his article at:

<http://www.sonatype.com/people/2009/09/testing-nexus-with-selenium-a-lesson-in-complex-ui-testing-part-1/>