Selenium Bootcamp 3: How to Write Maintainable Tests

Selenium Bootcamp is a series of articles prepared by Selenium guru Dave Haeffner for Sauce Labs. Dave also authors the Elemental Selenium website, which includes tips for using Selenium, and where you can sign up for his weekly email on the topic of Selenium testing.

One of the biggest challenges with Selenium tests is that they can be brittle and challenging to maintain over time. This is largely due to the fact that things in the application you're testing change -- causing your tests to break.

But the reality of a software project is that *change is a constant*. So we need to account for this reality somehow in our test code in order to be successful.

Enter Page Objects.

A Page Objects Primer

Rather than write your test code directly against your app, you can model the behavior of your application into simple objects and write your tests against them instead. That way when your app changes and your tests break, you only have to update your test code in one place to fix it.

With this approach, we not only get the benefit of controlled chaos, we also get reusable functionality across our suite of tests and more readable tests.

An Example

Part 1: Create A Page Object And Update Test

Let's take our login example from earlier, create a page object for it, and update our test accordingly.

First we'll need to create a package called `pageobjects` in our `src/tests/java` directory. Then let's add a file to the `pageobjects` package called `Login.java`. When we're done our directory structure should look like this.

```
pom.xml
src
       test
           java
               pageobjects
                  Login.java
               tests
                   TestLogin.java
```

And here's the code that goes with it.

```
// filename: pageobjects/Login.java
package pageobjects;
import org.openqa.selenium.By; import org.openqa.selenium.WebDriver;
public class Login {
private WebDriver driver;
By usernameLocator  = By.id("username");
By passwordLocator  = By.id("password");
By loginFormLocator = By.id("login");
By successMessageLocator = By.cssSelector(".flash.success");

public Login(WebDriver driver) {
    this.driver = driver;
    driver.get("http://the-internet.herokuapp.com/login");
}

public void with(String username, String password) {
    driver.findElement(usernameLocator).sendKeys(username);
    driver.findElement(passwordLocator).sendKeys(password);
    driver.findElement(loginFormLocator).submit();
}

public Boolean successMessagePresent() {
    return driver.findElement(successMessageLocator).isDisplayed();
}

}
```

At the top of the file we specify the package where it lives and import the requisite classes from our libraries. We then declare the class (e.g., `public class Login`), specify our field variables (for the Selenium instance and the page's locators), and add three methods.

The first method (e.g., `public Login(WebDriver driver)`) is the constructor. It will run whenever a new instance of the class is created. In order for this class to work we need access to the Selenium driver object, so we accept it as a parameter here and store it in the `driver` field (so other methods can access it). Then the login page is visited (with `driver.get`).

The second method (e.g., `public void with(String username, String password)`) is the core functionality of the login page. It's responsible for filling in the login form and submitting it. By accepting strings parameters for the username and password we're able to make the functionality here dynamic and reusable for additional tests.

The last method (e.g., `public Boolean successMessagePresent()`) is the display check from earlier that was used in our assertion. It will return a Boolean result just like before.

Now let's update our test to use this page object.

```
// filename: tests/TestLogin.java
package tests;
import org.junit.Test; import org.junit.Before; import org.junit.After;
import static org.junit.Assert.*; import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver; import pageobjects.Login;
public class TestLogin {
private WebDriver driver;
private Login login;

@Before
public void setUp() {
    driver = new FirefoxDriver();
    login = new Login(driver);
}

@Test
public void succeeded() {
    login.with("tomsmith", "SuperSecretPassword!");
    assertTrue("success message not present",
            login.successMessagePresent());
}

@After
public void tearDown() {
    driver.quit();
}
}
```

Since the page object lives in another package, we need to import it (e.g., `import pageobjects.Login;`).

Then it's a simple matter of specifying a field for it (e.g., `private Login login`), creating an instance of it in our `setUp` method (passing the `driver` object to it as an argument), and updating the test with the new actions.

Now the test is more concise and readable. If you save this and run it (e.g., `mvn clean test` from the command-line), it will run and pass just like before.

Part 2: Write Another Test

Creating a page object may feel like more work than what we started with initially. But it's well worth the effort since we're in a much sturdier position (remember: controlled chaos) and able easily write follow-on tests (since the specifics of the page are abstracted away for simple reuse).

Let's add another test for a failed login to demonstrate.

First, let's take a look at the markup that gets rendered when we provide invalid credentials:

```
<div id="flash-messages" class="large-12 columns">
  <div data-alert="" id="flash" class="flash error">
    Your username is invalid!
    <a href="#" class="close">x</a>
  </div>
</div>
```

Here is the element we'll want to use.

```
class="flash error"
```

Let's add a locator to our page object along with a new method to perform a display check against it.

```
 //filename: pageobjects/Login.java ... By successMessageLocator = By.
cssSelector(".flash.success"); By failureMessageLocator = By.cssSelector(".
flash.error"); ... public Boolean successMessagePresent() { return driver.
findElement(successMessageLocator).isDisplayed(); }
public Boolean failureMessagePresent() {
    return driver.findElement(failureMessageLocator).isDisplayed();
}

}
```

Now we're ready to add another test to check for a failure condition.

```
java //filename: tests/TestLogin.java ... @Test public void failed() {
login.with("tomsmith", "bad password"); assertTrue("failure message wasn't
present after providing bogus credentials", login.failureMessagePresent());
} ...
```

If we save these changes and run our tests (`mvn clean test`) we will see two browser windows open (one after the other) testing for successful and failure login scenarios.

Why Asserting False Won't Work (yet)

You may be wondering why we didn't just check to see if the success message wasn't present in our assertion.

```
java @Test public void failed() { login.with("tomsmith", "bad password");
assertFalse("success message was present after providing bogus
credentials", login.successMessagePresent()); }
```

There are two problems with this approach. First, our test will fail. This is because Selenium errors when looking looking for an element that's not present on the page -- which looks like this:

```
org.openqa.selenium.NoSuchElementException: Unable to locate element:
{"method":"css selector","selector":".flash.error"}
Command duration or timeout: 123 milliseconds
For documentation on this error, please visit: http://seleniumhq.org
/exceptions/no_such_element.html
Build info: version: '2.43.1', revision:
'5163bceef1bc36d43f3dc0b83c88998168a363a0', time: '2014-09-10 09:43:55'
System info: host: 'asdf', ip: '192.168.1.112', os.name: 'Mac OS X', os.
arch: 'x86_64', os.version: '10.10.1', java.version: '1.8.0_25'
Driver info: org.openqa.selenium.firefox.FirefoxDriver
Capabilities [{applicationCacheEnabled=true, rotatable=false,
handlesAlerts=true, databaseEnabled=true, version=34.0.5, platform=MAC,
nativeEvents=false, acceptSslCerts=true, webStorageEnabled=true,
locationContextEnabled=true, browserName=firefox, takesScreenshot=true,
javascriptEnabled=true, cssSelectorsEnabled=true}]
Session ID: b6648aef-5be5-e542-add1-265ed2a35a65
...
```

But don't worry, we'll address this in the next chapter.

Second, the absence of a success message doesn't necessarily indicate a failed login. The assertion we ended up with is more concise.

Part 3: Confirm We're In The Right Place

Before we can call our page object finished, there's one more addition we should make. We'll want to add an assertion to make sure that Selenium is in the right place before proceeding. This will help add some resiliency to our test.

As a rule, you want to keep assertions in your tests and out of your page objects. But this is the exception to the rule.

```
 // filename: pagesobjects/Login.java ... import static org.junit.Assert.
assertTrue;
public class Login { ... public Login(WebDriver driver) { this.driver =
driver; driver.get("http://the-internet.herokuapp.com/login"); assertTrue
("The login form is not present", driver.findElement(loginFormLocator).
isDisplayed()); } ... ```
```

After importing the assertion library we put it to use in our constructor (after the Selenium command that visits the login page). With it we're checking to see that the login form is displayed. If it is, the tests using this page object will proceed. If not, the test will fail and provide an output message stating that the login form wasn't present.

Now when we save everything and run our tests (e.g., `mvn clean test` from the command-line), they will run just like before. But now we can rest assured that the tests will only proceed if the login form is present.

Outro

With Page Objects you'll be able to easily maintain and extend your tests. But how you write your Page Objects may vary depending on your preference/experience. The example demonstrated above is a simple approach. Here are some additional resources to consider as your testing practice grows:

- https://code.google.com/p/selenium/wiki/PageObjects
- https://code.google.com/p/selenium/wiki/PageFactory (a Page Object generator/helper built into Selenium)
- https://github.com/yandex-qatools/htmlelements (a simple Page Object framework by Yandex)

Now that you understand how to write maintainable tests with page objects, our next installment will dive into writing resilient tests.