Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 2 (bonus) — January 8, 2025

Patrick Lam *version 1*

These notes are extra material about using Selenium to test web frontends.

Selenium[1]

Selenium is a tool for (among other things) testing web applications. In its full generality, "Selenium automates browsers" and lets you programmatically drive Web browsers. For the purposes of this class, you can use Selenium to automate web application tests, testing multiple browsers and multiple platforms.

In Assignment 1, I asked you to use Selenium to test a web application. I've simplified the setup by not actually launching a browser; Selenium contains a "headless" (GUI-less) browser simulator called `HtmlUnit`. Selenium can also drive real browsers, and it's interesting to watch it do that. Furthermore, you can set up a bunch of different computers and drive them from a single Selenium script.

Selenium also includes record-replay functionality (the Selenium IDE, a Firefox add-on), but I won't discuss it.

Anatomy of a Selenium Test. There are many bindings for Selenium, but we'll talk about using Selenium from JUnit.

Like any other test, there is setup and teardown; for Selenium, you ask for the browser to be started and terminated.

```
1   @Before
2   public void openBrowser() {
3     baseUrl = System.getProperty("webdriver.base.url");
4     driver = new FirefoxDriver(); // could also be Chrome, IE,
          HtmlUnit, etc.
5     driver.get(baseUrl);
6   }
7
8   @After
9   public void saveScreenshotAndCloseBrowser() throws IOException {
10    driver.quit();
11  }
```

Note that we create a browser and ask it to load a page.

[1] `http://seleniumhq.org`

Now let's look at a particular test. Like any other test, it sets up the test state, calls upon the system under test to do the action, and checks the result.

```
1   @Test
2   public void testPageTitleAfterSearchShouldBeginWithDrupal() throws
        IOException {
3     assertEquals("The page title should equal Google at the start of
          the test.",
4                  "Google", driver.getTitle());  // (1)
5     WebElement searchField = driver.findElement(By.name("q")); //
          (2)
6     searchField.sendKeys("Drupal!"); // (3)
7     searchField.submit();            // (3)
8     assertTrue("The page title should start with the search string
          after the search.", // (4)
9               (new WebDriverWait(driver, 10)).until(new
                    ExpectedCondition<Boolean>() {
10                       public Boolean apply(Object d) {
11                           return ((WebDriver)d).getTitle().
                                toLowerCase().startsWith("drupal!");
12                       }
13                   })
14              );
15  }
```

Here's what's going on.

1. The initial `assertEquals` is based on an assumption that the test is called on `www.google.com`.
2. Next, the test searches for the appropriate input on the webpage. (For Java tests, we don't need to search for the method that we're calling. Here, we do.) Selenium includes various ways to search the Document Object Model.
3. Having found the appropriate field, the test sends input and submits the form. This is analogous to calling the methods on the System Under Test.
4. Finally, the test waits for the action to complete (with `WebDriverWait`). Once complete, it checks that the outputs are correct; in this case, it checks that the title of the resulting page starts with "drupal!".

Waiting. In the above example, the test simply waits until the page loads (or 10 seconds). Selenium tests can also wait for more sophisticated conditions, for instance until a certain element appears on the page. (This is relevant when the JavaScript is adding content to the page dynamically).

Example:

```
1 WebDriverWait wait = new WebDriverWait(driver, 10);
2 WebElement element =
3   wait.until(ExpectedConditions.elementToBeClickable(By.id("someid")
      ));
```

Page Objects

The above example hard-coded the "q" field on Google's page. But things change. We can fix that by introducing a level of indirection.

References:

`http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#chapter06-reference`
`https://martinfowler.com/bliki/PageObject.html`

A page object should abstractly represent the actions that a user can take on a page and allow the caller to query the state of the page (if necessary). The Selenium documentation suggests a page object for a sign-in page:

```
1  public class SignInPage {
2    private WebDriver driver;
3
4    public SignInPage(WebDriver driver) {
5      this.driver = driver;
6      if(!driver.getTitle().equals("Sign in page")) {
7        throw new IllegalStateException("This is not sign in page,
             current page is: "
8        +driver.getLocation());
9      }
10   }
11
12   public HomePage loginValidUser(String userName, String password) {
13     driver.type("usernamefield", userName);
14     driver.type("passwordfield", password);
15     driver.click("sign-in");
16     driver.waitForPageToLoad("waitPeriod");
17
18     return new HomePage(driver);
19   }
20 }
```

The documentation suggests that the only verification that should occur on a page object is testing that the driver points to the right page. Everything else should be done in the tests.

The sign-in page object also exposes the sole functionality of the page, which is to sign in. It then returns the page that gets loaded after sign-in.

Note that the caller does not need to know about the identity of the username and password fields. In fact, let's say that a new version of the app included Facebook OAuth login. It would suffice to change the `SignInPage` to use the new login functionality; no other parts of the test suite would need to change.