Selenium 2019

Page Objects and Beyond

About This Talk

The original version of this talk was presented as an internal talk at Celtx in 2019 to introduce the software development staff to key concepts for implemented automated testing using Selenium and the Page Object pattern.

It was subsequently revised and presented to members of the NDev software development community in St. John’s, Newfoundland and Labrador in September of 2019 as part of NDev meetup #40.

About Me: Mike Murphy-Burton

QA Automation Lead at Celtx

My first general test framework was xUnit for C#, VB

I’ve worked with a variety of tools for creating user-facing test tools:

- Selenium
- Fitnesse
- Robot
- Windows Journalling

A Brief History of Selenium

Testing tools

1990s and earlier: Largely bespoke, organization-dependent

1989-1998 - SUnit (Beck)

1996 - Extreme Programming (Beck, Cunningham, Jeffries)

1997 - JUnit

1998 - Mercury Interactive

2001 - Manifesto for Agile Software Development

2004 - Selenium

Selenium

Developed in 2004 at ThoughtWorks by Jason Huggins

The name is a joke at Mercury's expense (prior to its acquisition by HP)

As of 2.0, Selenium uses the WebDriver architecture, which entered W3C Recommendation as of 2018

Selenium

Java Example

```
class MyTestFixture {
     WebDriver driver;
     @BeforeEach
     void setup() {
          driver = new RemoteWebDriver(DesiredCapabilities.chrome());
     }

     @AfterEach
     void teardown() {
          driver.quit();
     }

     @Test
     void testClickElement() {
          WebElement el = driver.findElement(By.cssSelector("tag.class.otherclass"));
          el.click();
          new WebDriverWait(driver, timeout).until(d -> meetsCondition(d));
     }
}
```

Runners

JUnit

- Well-vetted and usually familiar to any Java dev
- Some support lacking (for example, repeat only on exceptions)
- However, large community tends to compensate

TestNG

- Heavily influenced by JUnit
- Growing adoption by automated testers
- More built-in support than JUnit
- Smaller overall community, so not as many options for plugging gaps (but with JUnit 5, this is less of a concern)

WebDriver and DesiredCapabilities

```
WebDriver createLocalChromeDriver() {
    return new ChromeDriver(new ChromeOptions());
}

WebDriver createRemoteDriver() {
    DesiredCapabilities capabilities = DesiredCapabilities.chrome();
    capabilities.setCapability("version", "65.0");
    Return new RemoteWebDriver(desiredCapabilities);
}
```

Chrome/Firefox/SafariDriver

Local driver binary files allow direct execution of tests against local browsers.

Test code has to know how to find the binary file!

Some mismatches in capabilities - Chrome’s experimental options, for example

May allow some otherwise unsupported calls - element.getPosition, for example

RemoteWebDriver

RemoteWebDriver allows the execution of tests against a Selenium server.

Running against BrowserStack, for example, requires using RemoteWebDriver.

Still need to specify browser name and version, but don't need to have a driver binary accessible to your test suite.

BrowserStack Example

```
DesiredCapabilities capabilities = DesiredCapabilities.chrome();
capabilities.setCapability("browserstack.user", "myusername");
capabilities.setCapability("browserstack.key", "abc123");
```

WebDriverWait

Because Selenium is automating systems from the user perspective, there's a lot of asynchronous behaviour and unpredictable timing involved.

WebDriverWait provides a fluent interface to define waits for specific states. It is often combined with **ExpectedConditions**, which provides common wait conditions, and **By**, which provides common locators.

Example:

```
new WebDriverWait(driver, 30).ignoring(NoSuchElementException.class)
     .until(ExpectedConditions.visibilityOfElementLocatedBy(By.cssSelector("div.class")));
```

Actions

Sometimes you need complex input, or find that executing a particular input command against an element is either flaky or non-functional.

Actions provide an alternate mode of executing input, and tend to be more stable in terms of outcomes.

Actions Example

```
Actions actions = new Actions(driver);
actions.moveToElement(driver.findElement(By.cssSelector("#threestateel")))
     .click().click()
     .build().perform();
```

The Page Object Pattern

The Problem with Selenium

UI/UX tests are inherently brittle. Every redesign of a page can break tests.

Attempts have been made to mitigate this issue by clever HTML+CSS design as well as the use of extended attributes like **data-qa**

Data-QA example:

```
<div class=”container etc” data-qa=”our.container.element”>
</div>

driver.findElement(By.cssSelector(“[data-qa=’our.container.element’]”));
```

The PageObject Approach

What if we don’t represent our UI directly in our tests, but instead focus on functional representation? That is, instead of this:

```
driver.findElement(By.cssSelector(“button.ok”)).click();
```

We did:

```
PageObject.init(driver, MyPage.class)
    .clickOK();
```

MyPage is a **PageObject** that represents a particular piece of a page or view.

Page Object Pattern

Page Objects provide a set of **actions** and a set of **accessors**. Actions allow the test developer to change page state by, for example, opening a dialog. Accessors, on the other hand, return information about the current state.

Action

```
myPage.addUser(newUser);
```

Accessor

```
List<UserData> users = myPage.getUsers();
```

Page Objects & Fluent Interfaces

Selenium tests consist of a series of actions interspersed with assertions against the view state. Page Object actions return PageObjects to allow fluent tests

Example

```
class MyPage {
     WebDriver driver;
     MyPage(driver) {
          this.driver = driver;
     }

     MyPage doSomething() {
          makeAThingHappen();
          Return this;
     }
}
```

Test Cadence and Single Responsibility

Fluent interfaces introduce the opportunity to create monolithic method chains.

```
PageObject.init(driver, MyPage.class)
     .doSomething().assertSomething()
     .doSomethingElse().assertSomethingElse
```

This violates the SRP, however - MyPage will change both when the design of the page changes and when the design of the test changes. A better pattern is:

```
MyPage sut = PageObject.init(driver,MyPage.class)
Data state = sut.doSomething().getState();
assertSomething(state)
Data nextState = sut.doSomethingElse().getState();
assertSomethingElse(nextState);
```

Flaky Tests & @Repeatable

Why Are Selenium Tests So Dang Flaky?

Fully integrated (or “system”) tests, particularly those involving a network, are prone to failure, BUT we still need to validate the system from a user’s perspective.

There are many reasons Selenium tests might fail:

- Network outage
- Slow performance
- JavaScript errors
- Uncaught ephemeral transition conditions

Some of these are “true” errors, but statistically you may see 1% or more “false positives” in a suite that has a few hundred test methods -> WASTED TIME!

@RepeatedTest and @RepeatedIfExceptionsTest

Prior to JUnit 5, there were a large number of extensions that could change the behaviour of test cases. Because of lifecycle changes in JUnit 5, many no longer work, and there aren't always updated options available.

JUnit 5 ships with **@RepeatedTest**, which will automatically repeat any given test N times. This translates into a factor of N inflation in run time, which is unacceptable for large, frequently-run suites.

Enter **artsok** and **@RepeatedIfExceptionsTest**

```
@RepeatedIfExceptionsTest(name = "My test - repetition {currentRepetition}, repeats = 3)
```

Interactive Demo

Page Object Limitations

PageObject extends the breakage horizon, but…

1. What if you radically change an interface component (toolbar vs dropdown menu)
2. Some actions don’t map cleanly (login)
3. Sometimes you want to skip directly to a specific state

Other tools and approaches - such as sub-UI testing frameworks like Cypress - handle one or more of these situations better than vanilla Page Objects

Further Study

Selenium 4

From a test dev perspective, this is a relatively minor upgrade

Key Changes

- WebDriver conforms to W3C standard
- Selenium Grid upgrades, particularly for containerized grid nodes.
- Significant changes to DesiredCapabilities for various browsers
- Replacement of getSize/getPosition with getRect

Selenium IDE

“Record” tests in browser

Captures multiple locators (but not data-qas)

Very ugly output (`driver.findElement(By.cssSelector(".class")).click();`)

Replay locally or remotely against multiple browsers

Very, very ugly output (`vars.put("win4279", waitForWindow(2000));`)

Supports further extension via WebExtension standard

Very, very, very ugly output (`js.executeScript("if(arguments[0].contentEditable === 'true') {arguments[0].innerText = <big lump of HTML>}", element);`)

WebDriverIO

Reimplementation of the WebDriver standard with advanced functionality

Runs on Node.JS

Native Applitools integration

Visual Regression Testing

Particularly as machine vision evolves, I think this is very likely to largely replace Selenium et al in the next 10 years

LOTS of options - BackstopJS, Wraith, Percy, Applitools

Recommended talk: Spot the Difference - https://www.youtube.com/watch?v=zCln-Cj_qyg