Day 4: Developing Automation

# KDT/Page Object Model

We will discuss two models for developing automation with Selenium. The first is the Page Object Model.

The Page Object Model is a system that attempts to map out web pages and pieces of functionality on those webpages into small segments of functionality. In the Page Object Model, you would have a class that represents a web page, and inside the class each function performs some action you can perform on the page. Each function in the Page Object Model also returns a Page Object.

For example, logging into a website would require you to enter a username and password and then click the submit button. In the Page Object Model, each one of those three actions would be segmented into their own function - one to enter a username, one to enter a password, and one to click the submit button. This allows you to build segments of code that are less dependent on each other, and if something on the website changes (for example, replacing the 'Username' box with a button), you would only have to change the code in one place.

Generally, the functions themselves don't expose the details of the piece of functionality they're working on. So, your method to login would read:

public LoginPage enterUsername(String username) {

//Enter the name into the box

}

public LoginPage enterPassword(String password) {

//Enter the password into the box

}

public LoginPage submitLogin() {

//Click the submit button

}

public HomePage login(String user, String pass) {

enterUsername(user);

enterPassword(pass);

return submitLogin();

}

This block of code wouldn't need to change even if the details of the UI changed, unless the type of PageObject that each of those functions returned was altered.

See further documentation on Page Object here: https://code.google.com/p/selenium/wiki/PageObjects

# Keyword-Driven Testing

An alternate method is the Keyword-Driven Testing model.

In the KDT model, you organize functions into keywords, which can be thought of as verbs that act on the web page elements. In the login page example, you might have a keyword that enters arbitrary text into an arbitrary textbox followed by a keyword that clicks an arbitrary button.

Keywords tend to obscure less of the functionality and are more flexible. The keyword to enter text into a textbox could be used on any textbox, as long as you wrote it to accept a textbox WebElement as input.

Your login code might look like this:

public static class EnterText(WebElement textbox, String text) {

//Enter the text into the textbox

}

public static class ClickButton(WebElement button) {

//Click the button

}

public static class Test() {

WebElement tbox = driver.findElement(/\*the textbox\*/)

WebElement tbox2 = driver.findElement(/\*the password textbox\*/

WebElement btn = driver.findElement(/\*the button\*/)

EnterText(tbox, username);

EnterText(tbox2, password);

ClickButton(btn);

}

Note the fact that these keywords can be easily reused, as long as you obtain all the necessary objects (WebElements etc.) within your main functions. However, when something changes on the web pages, you will have to dig through your main functions to edit those objects. Which helps lead into the next topic:

# Object Repository

An object repository is basically a place to store all of your objects in one place. This can be as simple as a file with a list of XPaths that correspond to titles which correspond to specific web elements. The simplest way to create an object repository is to use the built-in Java properties class (http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html).

Properties prop = new Properties();

FileInputStream in = new FileInputStream("elements.properties");

prop.load(in);

in.close();

driver.getElement(By.xpath(prop.getProperty("elementXPath");

Where "elementXPath" in your properties file might read:

elementXPath = "//body/div/div[1]"

This would find the element corresponding to that XPath, but you can use a properties file to keep all your XPaths stored in one place that's easy to organize, easy to read, and out of your regular code.

# Exercise 3:

This exercise will be an update on Exercise 1.

Step 1: Navigate to a random Wikipedia page using what you learned from Exercise 1.

Step 2: Pick a random category from the page's list of categories at the bottom of the page.

Step 3: Select a random article from the list of pages in the selected category.

Step 4: Output the compared differences between the categories of the original random page and the second one.

Step 5: Make this work in multiple browsers.