

Lab Module 2 – Build a Domain Specific Language

Introduction

Writing Web UI Automation Test is quite intuitive and easy to write once you've seen a few examples using Selenium to locate elements, perform action and make some assertions. However, maintaining these UI Automated test – *which are brittle by nature* – can be harder over time to the point where it becomes impractical to maintain.

In this module we will explore a few patterns to introduce a strong separation of concerns between the scripts - *which are the specifications or scenarios of what the tests should do* – and specialised objects abstracting the detailed interaction with the Pages using Selenium of our Application or System Under Test (SUT).

Pre-requisites

Docker for Windows/Mac/Ubuntu, Visual Studio 2015, 2017 or 2019 and a GitHub account.

Objectives

After completing this lab, you will be able to:

- Refactor existing brittle UI Selenium Tests with a lightweight DSL using Page Objects
- Understand how to apply SOLID, DRY and DAMP principle to maximise, optimise maintainability & readability of your test harness
- Although this example focuses on Web UI Automation, these principles apply for any type of interfaces whether Desktop or Mobile.

Scenario

Our new e-commerce website is about to release to production and some UI Selenium tests were written in a rush due to tight deadline by a contractor who left the organisation. You are now in charge on maintain these and extends these as an Automation Quality Engineer.

Setting up

Cloning out Application Under Test

All the demos use the open source ASP.NET Core 2.2 reference application eShopOnWeb.

1. To set it up and getting it running, please execute the following commands:

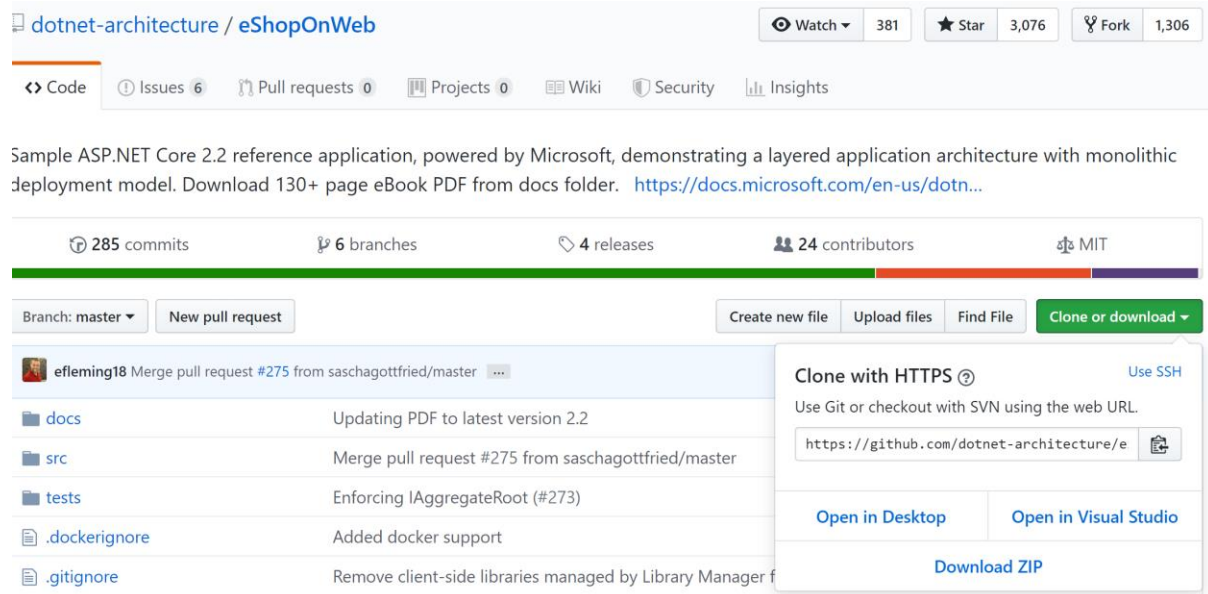
```
git clone https://github.com/dotnet-architecture/eShopOnWeb
cd eShopOnWeb
docker-compose build
docker-compose up
```

You can now access the application from <http://localhost:5106>

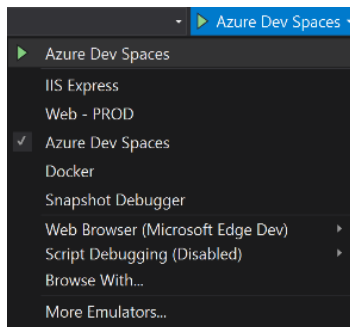
2. Alternatively, if you have an Azure Subscription, the solution can be deployed in Azure DevTest AKS environment. If you don't have a AKS DevTest Cluster run the following Azure CLI command using [Azure Cloud Shell](#)

```
az aks create -g <resourceGroupName> -n <AKSClusterName> --location
<region> --disable-rbac --generate-ssh-keys
```

Then clone the repository and open eShopOnWeb.sln with Visual Studio



And finally deploy using *Azure DevTest* from the list of execution environments as below:

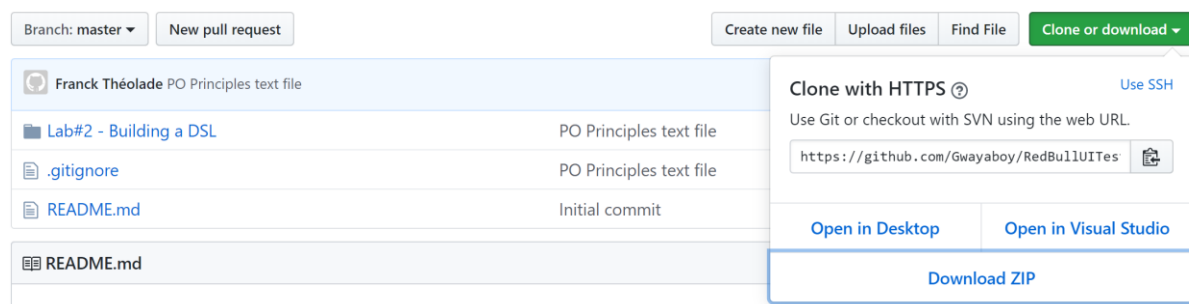


Cloning the Labs Repository

Download and clone the repository

```
git clone https://github.com/Gwayaboy/RedBullUITestingWorkshop
```

Or clone navigate to the URL above and choose clone with Visual Studio:



Exercise 1: Refactoring BrittleTest with Page Objects

Introduction

In this exercise we will:

- 1) Refactor Test fixtures to facilitate Page Object creation with factories and simple fluent API.
- 2) Break down long running *BrittleTest* user journey into separate functional flows
- 3) Encapsulate the page interaction into Page Objects

Tasks 1.1: Refactoring Plumbing code to facilitate Page Object Creation

*The Page Object pattern is as a way of encapsulating the interaction with an individual page in your application in a single object. It makes heavy use of Object-Oriented principles to enable code reuse and improve maintenance. Your test scripts are very procedural, and the details are handled in Page Objects. Scripts are the **What** and Page Objects are the **How**.*

1. Expand the Step 1 folder under the solution and notice there is a *IntroducingPageObjects.txt* file. You can read that file at the end of this exercise to gain more perspective on the purpose and principles of Page Object pattern.
2. Build the solution to restore all NuGet package dependencies and run `Registered_User_Can_buy_a_cup_of_T` test method of `BrittleTests` from the TestExplorer
3. First, let's add create a Page subfolder under Step 1 and add a new abstract `Page` class at the root of the solution, this will be our base page object which all our page will inherit from. This page will encapsulate selenium's `IWebDriver` instance
4. Let's add a protected `WebDriver` Property and a couple of public properties for our pages like `Title` and `Url`.

```
public abstract class Page
{
    protected IWebDriver WebDriver { get; private set; }

    public string Title => WebDriver.Title;

    public string Url => WebDriver.Url;
}
```

5. In our base `Page` we will also need a static internal generic method which will be responsible for navigating and creating our initial page object.
We will also guard our method against null `WebDriver` and empty or null start-up URL

```
internal static TPage NavigateToInitial<TPage>(IWebDriver driver, string url)
where TPage: Page, new()
{
    if (driver == null)
        throw new ApplicationException("Please provide web driver to proceed");

    if (string.IsNullOrEmpty(url))
        throw new ApplicationException("Please provide with a start-up url");

    driver.Navigate().GoToUrl(url);

    return new TPage { WebDriver = driver };
}
```

Notice we have placed some constraints on the generic `TPage` type to only return concrete children from `Page` with parameter-less constructors.

- Let's now introduce our first (empty) page object which will encapsulate our landing page. Under Step 1 folder, add a Pages subfolder and add in that folder our [HomePage](#) Page class which inherits from our base Page.
- Then we will need a factory class which will be responsible for creating WebDriver instance and navigating to an initial page. At the root of the solution add a Configuration folder and within it a [BrowserHost](#) class as follow:

```
public class BrowserHost
{
    private IWebDriver _webdriver;

    private BrowserHost(IWebDriver webDriver)
    {
        _webdriver = webDriver;
        _webdriver.Manage().Window.Maximize();
    }

    public TPage NavigateToInitial<TPage>(string url)
        where TPage : Page, new()
    {
        return Page.NavigateToInitial<TPage>(_webdriver, url);
    }

    public static BrowserHost Chrome()
    {
        var options = new ChromeOptions();
        options.AddArgument("test-type");
        var directory = Directory.GetCurrentDirectory();

        return new BrowserHost(new ChromeDriver(directory, options));
    }
}
```

We also take advantage here to maximise the actual browser's window (or centralise any other global operation we need to do on the web driver such explicit wait time, etc.)

- We also need to add to our Page base class another (non-static) generic and protected method to facilitate navigating from page to page when performing an action on an element from a given page such as clicking or submitting a form

```
protected TPage NavigateTo<TPage>(By byLocator, Action<IWebElement>
performAction = null) where TPage : Page, new()
{
    var webElement = WebDriver.FindElement(byLocator);
    var action = performAction ?? (e => e.Click());
    action(webElement);

    return new TPage { WebDriver = WebDriver };
}
```

You can see that the method above has 3 goals:

- Finding a web element by the specified locator passed as an argument*
- Performing an action on the web element found (which will be by default clicking on it) the action can also be specified as an argument*
- Creating and returning the resulting (strongly typed) page with the current web driver*

9. Next, Let's make `BrowserHost` implement `IDisposable` and implement the disposable pattern as follow:

```
#region IDisposable Support
private bool disposedValue = false; // To detect redundant calls

protected virtual void Dispose(bool disposing)
{
    if (!disposedValue && disposing && (_webdriver != null))
    {
        _webdriver.Quit();
        _webdriver.Dispose();
        _webdriver = null;
    }
    disposedValue = true;
}

// This code is added to correctly implement the disposable pattern.
public void Dispose()
{
    Dispose(true);
}
#endregion
```

10. Finally, we will introduce a `FunctionalUITestClass` abstract base class to take away some of the set up and tear down boiler plate code for creating or closing gracefully an instance of the `WebDriver` through `BrowserHost`. All our UI functional test classes will be inheriting from that class to help reducing noise to a minimum.

Please note, we have `Browser` protected property which holds an instance of `BrowserHost`. This base class also have a mandatory constructor which accepts a delegate factory function to create instance of `BrowserHost` on test initialise.

```
public abstract class FunctionalUITest
{
    private readonly Func<BrowserHost> _browserHostFactory;
    protected BrowserHost Browser { get; private set; }

    public FunctionalUITest(Func<BrowserHost> browserHostFactory)
    {
        _browserHostFactory = browserHostFactory;
    }

    [TestInitialize]
    public virtual void RunBeforeEachTests()
    {
        Browser = _browserHostFactory();
    }

    [TestCleanup]
    public virtual void RunAfterEachTests()
    {
        if (Browser != null)
        {
            Browser.Dispose();
            Browser = null;
        }
    }
}
```

Also please note that this currently will be creating and disposing of the WebDriver for each test method. This is optimised for isolating test methods from each other as we start from a clean browser session each time.

This behaviour can be modified by creating a different base class and using the `[ClassInitialize]` and `[ClassCleanup]` attributes for class level initialisation once before and after all test methods.

Lastly please notice we are using the Disposable pattern to gracefully dispose of the webdriver and inform garbage collection to free up the memory `BrowserHost` was occupying

Tasks 1.2: Breaking Down BrittleTest and Plumbing code to facilitate Page Objects

We can now start refactoring our functional test about buying a Mug of T by breaking down `BrittleTest` which mix a few things functional flows

- Search for a Mug<T>
- Add to Basket and Head to Checkout
- Checkout, Place Order and Verify Order has been placed

Using our Page Objects DSL, we will focus on confirming you can add a Mug<T> to the basket (providing it exists in our product catalogue), and head to checkout. Then we will introduce separate test for searching for an item through the product catalogue and lastly proving there's many items in the basked we can place an order.

1. Let's start by adding a new `AddingToBasketAndCheckingOutTests` Test class inheriting from `FunctionalUITest` to represent our "Add to Basket and Head to Checkout" scenario.

```
[TestClass]
public class AddingToBasketAndCheckingOutTests : FunctionalUITest
{
    public AddingToBasketAndCheckingOutTests() : base(BrowserHost.Chrome)
    { }
}
```

Our new test class represents a set of scenarios around Adding known items to basket and heading to checkout and can specify through `FunctionalUITest` in a fluent way which browser to use.

2. Add a new `Can_add_selected_item_basket()` test method and the following arrange statement to navigate to our initial `HomePage`.

```
[TestMethod]
public void Can_add_selected_item_basket()
{
    //Arrange
    var homePage =
        Browser.NavigateToInitial<HomePage>("http://localhost:5106");
}
```

The original UI Test filters items by type first and rely on finding an item by its index (second result of the search). What if there was of finding the item directly with some kinds of unique id for it?

After a few conversations with the developers, we quickly found out that each product has a unique (hidden) id which can be used to directly to select an item from the list of available products on the landing page. This item can then be added to the basket by submitting the form that hidden web element belongs to.

3. Another principle of Page Objects is that any action on the actual web page results of either staying on the same page or navigating to another page.
In this case adding the selected item to the basket will redirect to the [BasketPage](#).

In that sense, let's now Add an act statement for adding the known item to the basket by its product Id.

```
//Act  
var actualPage = homePage.AddToBasketByProductId(2);
```

The [HomePage](#) doesn't have yet an `AddToBasketByProductId(int productId)` method. Neither does is there a [BasketPage](#) yet.

We can generate the method for us by using ReSharper or Visual Studio's keyboard shortcut Ctrl + . contextual menu and implement the [HomePage](#)'s method as follow:

```
public class HomePage : Page  
{  
    public BasketPage AddToBasketByProductId(int productId)  
    {  
        var selector = ($"//*[@type=\"hidden\"][@value={productId}]");  
        return NavigateTo<BasketPage>(By.XPath(selector), e => e.Submit());  
    }  
}
```

Our method above make use of the base [Page](#)'s `NavigateTo<TPage>` method for

- *finding the hidden input web element with XPath selector where its value matches the specified product id*
- *submitting the form that element belongs to.*

The [BasketPage](#) doesn't exists yet but we are driving the implementation of our SDL based on test scripts demands so that we don't over-complicated with just what is needed.

4. Next let's generate our [BasketPage](#) using the same contextual menu (Ctrl + .) and ensure it inherits from our base [Page](#)

```
public class BasketPage : Page { }
```

5. Last (but not the least), to complete our test scenario, we need to make a few assertions whether we have selected the correct item, added to the basket and headed to the basket page. Our full test scenario looks as below

```

[TestMethod]
public void Can_add_selected_item_basket()
{
    //Arrange
    var homePage =
        Browser.NavigateToInitial<HomePage>("http://localhost:5106");

    //Act
    var actualPage = homePage.AddToBasketByProductId(2);

    //Assert
    Assert.IsInstanceOfType(actualPage, typeof(BasketPage));
    Assert.IsTrue(actualPage.Url.EndsWith("/Basket"));
    Assert.AreEqual(actualPage.Title, "Basket - Microsoft.eShopOnWeb");
}

```

6. Let's build our solution to make sure we don't have compilation error and run the test to verify our test method passes, selecting the Mug<T> item, adding it to the basket and ending up at the basket page.