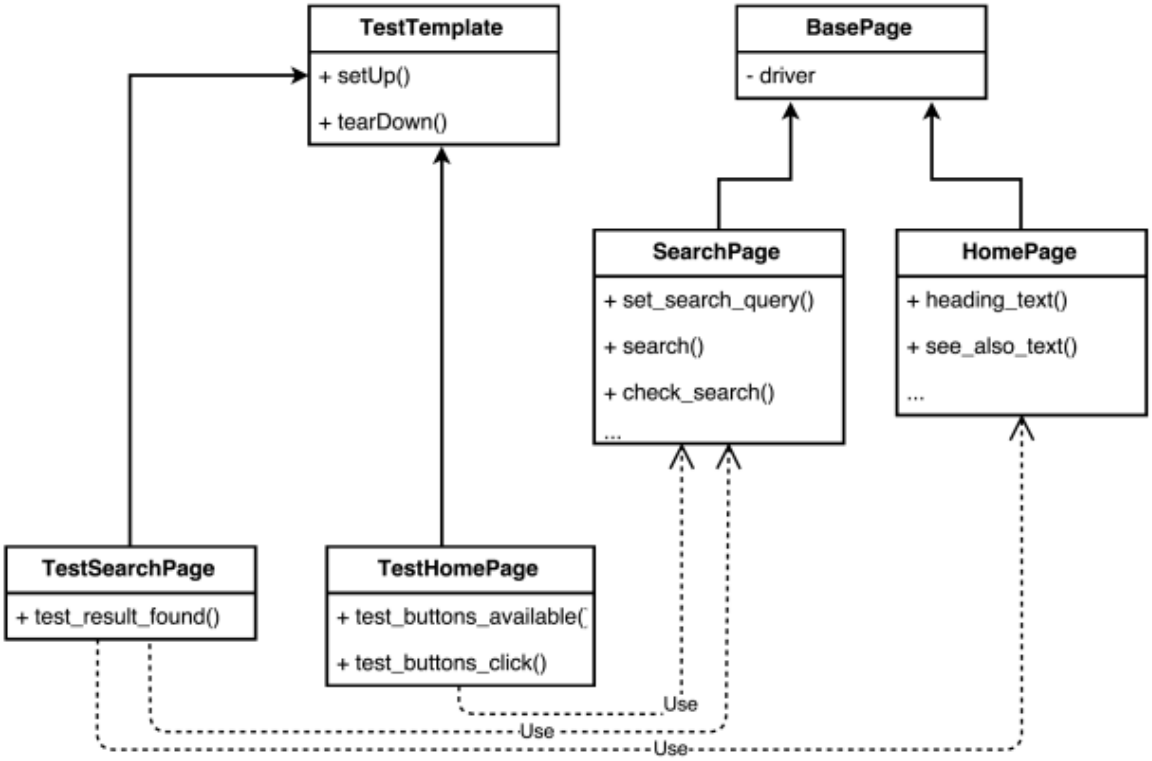
From a messy script to page object pattern in Python

atechnovel.com/2018/01/13/from-messy-automation-script-to-page-object-pattern-in-python

View all posts by paulbodean88 January 13, 2018



The purpose of this article is to help you understand how to organize a messy ‘record and play’ code into a maintainable one using the most popular design pattern in software testing ‘page object pattern’.

The transition from manual to automation testing is the logical next step for each software testers. But when you start your journey without any prior programming experience, it might be pretty challenging especially when you have to design a reusable automation framework.

Before starting I would recommend picking up a programming language, e.g. Python, learn its basics and afterward start to consider the applicability of design patterns.

About design patterns, my favorite definition is “a general reusable solution to a commonly occurring problem for different projects”. When it comes to automation testing, you also have to write code, even if the outcome is not a wonderful application which millions of people will use, it’s still an app, which actually prevents those millions of users to encounter different bugs in production. So, following some programming best practices is ideal also when designing an automation framework, and design patterns are a piece of them.

A page is an object-oriented class that serves as an interface to a page of your application under test. In this class, you will store ids, actions, and validators specific to the UI elements from a page. As a benefit when writing tests, you don’t have to care anymore about the automation setup e.g. the connectivity with Selenium driver or about those UI

changes which usually make the maintenance a hell. As a result, writing tests has never been so easy. You will see what I'm talking about in the next sections. This article, tutorial I would say is divided into three sections: environment configuration, write a messy script and the refactoring based on page object pattern.

Step1 – configuration

Related to the configuration, make sure you fix some dependencies.

- Install Python3
- Install virtualenv
- Choose an IDE e.g. Pycharm
- Chrome driver:
    - You can install it globally: brew install *'chromedriver'*. It works on macOS
    - Download it and send the path to your scripts

```
chromedriver = "/path/to/chromedriver"
os.environ["webdriver.chrome.driver"] = chromedriver
driver= webdriver.Chrome(chromedriver)
```

Create a virtual environment for your project:

```
$virtualenv -p python3 page_object_venv
```

Create a Python project using Pycharm IDE. and activate your virtual environment.

You can activate it or even create a new one using the IDE: Preferences… -> Project -> Project Interpreter -> Settings – > Add Local or CreateVirtualEnv

Step2 – Create a simple messy script

For this tutorial, we are going to automate the Wikipedia webpage https://en.wikipedia.org/wiki/Main_Page.

```
from selenium import webdriver

# Create a Chrome instance
driver = webdriver.Chrome()
driver.get("https://en.wikipedia.org/wiki/Main_Page")
driver.find_element_by_id("pt-createaccount").click()
driver.find_element_by_id("pt-login").click()
# Set your query
driver.find_element_by_id("searchInput").send_keys("Design patterns")
driver.find_element_by_id("searchButton").click()

# get the text for the main sections of the article
heading = driver.find_element_by_id("firstHeading").text
see_also = driver.find_element_by_id("See_also").text
domain_articles = driver.find_element_by_id("Domain-specific_articles").text
reading = driver.find_element_by_id("Further_reading").text
ref = driver.find_element_by_id("References").text
external_links = driver.find_element_by_id("External_links").text

def check_text(text_id, expected):
    if text_id == expected:
    print("Test passed")
else:
    print("Failed: Actual value", text_id, "Expected value", expected)
    # Check the text for some items from our list
    check_text(heading, "Design pattern")
    check_text(see_also, "See also")
    check_text(domain_articles, "Domain-specific articles")
    check_text(reading, "Reading")
    check_text(ref, "References")
    check_text(external_links, "External links")

# close the page
driver.quit()
```

Here we've accessed the Wikipedia page, clicked on some buttons from the main page, search for a string "Design patterns", and validate some text values from the result page.

Step3 – apply page object pattern

Now let's start to organize our code in page object pattern style. There are 4 things to consider when doing it:

1. Build a base page, which will be inherited by all the other pages. Here we will keep all the common code specific to selenium
2. Implement the pages
3. Create a test template page -> inherited by all the tests. It's like a blueprint of a test
4. Make some tests

To get a clear picture of this implementation please find below the class diagram of page object pattern applied on Wikipedia website:

```
Base page

class BasePage(object):

    def __init__(self, driver):
        self._driver = driver
```

Next, implement the pages from our framework. Based on the already implemented script we will create two separated objects: The Wiki home page and the Search result one.

Wiki homepage:

```
from src.page_object_pattern.base_page import BasePage

class HomePage(BasePage):
    SEARCH_CONTAINER = 'searchInput'
    SEARCH_BUTTON = 'searchButton'
    CREATE_ACCOUNT = 'pt-createaccount'
    LOGIN = 'pt-login'

def set_search_query(self, query: str):
    self._driver.find_element_by_id(HomePage.SEARCH_CONTAINER).send_keys(query)

def check_search(self):
    return self._driver.find_element_by_id(HomePage.SEARCH_BUTTON).is_displayed()

def search(self):
    self._driver.find_element_by_id(HomePage.SEARCH_BUTTON).click()

def check_login(self):
    return self._driver.find_element_by_id(HomePage.LOGIN).is_displayed()

def login(self):
    self._driver.find_element_by_id(HomePage.LOGIN).click()

def check_create_account(self):
    return self._driver.find_element_by_id(HomePage.CREATE_ACCOUNT).is_displayed()

def create_account(self):
    self._driver.find_element_by_id(HomePage.CREATE_ACCOUNT).click()
```

Search result page:

```
from src.page_object_pattern.base_page import BasePage

class SearchPage(BasePage):
    HEADING = 'firstHeading'
    SEE_ALSO = 'See_also'
    DOMAIN_ARTICLES = 'Domain-specific_articles'
    READING = 'Further_reading'
    REF = 'References'
    EXTERNAL_LINKS = 'External_links'

def heading_text(self):
    return self._driver.find_element_by_id(SearchPage.HEADING).text

def see_also_text(self):
    return self._driver.find_element_by_xpath(SearchPage.SEE_ALSO).text

def domain_articles_text(self):
    return self._driver.find_element_by_xpath(SearchPage.DOMAIN_ARTICLES).text

def reading_text(self):
    return self._driver.find_element_by_xpath(SearchPage.READING).text

def ref_text(self):
    return self._driver.find_element_by_xpath(SearchPage.REF).text

def external_links_text(self):
    return self._driver.find_element_by_xpath(SearchPage.EXTERNAL_LINKS).text
```

Now it’s time to create the test template page. Usually, each test class must contain two methods: setUp and tearDown. In order to avoid the code duplication, we will create a base test class that will be inherited by all the other tests.

```
import unittest
from selenium import webdriver

class TestTemplate(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.get("https://en.wikipedia.org/wiki/Main_Page")

def tearDown(self):
    self.driver.quit()
```

Here we’ve created the connection with the browser, we open a web page and also close the session once the test is done. Using this approach when a new test is triggered the page will be opened and closed automatically.

It’s time to see how our tests are looking. Also, the only thing we care about is writing tests.

TestHomePage

```
from src.page_object_pattern.home_page import HomePage
from src.page_object_pattern.test_template import TestTemplate

class TestHomePage(TestTemplate):

    def test_buttons_available(self):
    main_page = HomePage(self.driver)
    assert main_page.check_create_account()
    assert main_page.check_login()
    assert main_page.check_search()

def test_buttons_click(self):
    main_page = HomePage(self.driver)
    main_page.login()
    main_page.search()
```

Test search class

```
from src.page_object_pattern.home_page import HomePage
from src.page_object_pattern.search_page import SearchPage
from src.page_object_pattern.test_template import TestTemplate

class TestSearchPage(TestTemplate):

    def test_result_found(self):
        home_page = HomePage(self.driver)
        home_page.set_search_query("Design patterns")
        home_page.search()
        result = SearchPage(self.driver)
        assert result.heading_text() == "Design pattern"
```

What we've done? Using some UI element from Wikipedia page we've managed to implement the backbone of a solid automation framework using a popular design pattern: page object. Now you can extend the coverage with other functionalities following the same approach. Even if the number of pages will increase a lot, what you have to do only two things: create new object pages, write tests. If you want to start from scratch do not forget the following 4 logical steps:

1. Create the Base Page
2. Create a Simple page
3. Create the Test Template
4. Write a simple Test

That's all about page object pattern. Going further we will cover other design patterns applicable to automation testing.

You can also get the source code from Github.