


Organizing Automated Tests with Page Object Model

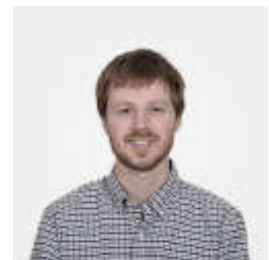
 willowtreeapps.com/ideas/organizing-automated-tests-with-page-object-model



Ashby Bowles

Senior Software Testing Analyst

In the world of test automation, Selenium Webdriver has become the standard tool for testing web applications. Selenium's success comes from its multi-platform support, ability to test across a wide variety of browsers, and it's free. For the most part, you can write your tests in the language you feel most comfortable with, but it is a good idea to use one that already has a test framework such as: Java, C#, Python, or Ruby.



Testers often automate their first test cases with relative ease. You find the web page elements, you click or type as a user would, and then you assert something is true or false.

```

class loginTests(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.get('http://somewebapp.com/')
        self.driver.implicitly_wait(5)

    def tearDown(self):
        self.driver.close()

    def test_login_incorrect_email(self):
        email = self.driver.find_element_by_Id('email_Id')
        email.send_keys('test@email.com')
        password = self.driver.find_element_by_Id('password_Id')
        password.send_keys('password123')
        submitButton =
self.driver.find_element_by_xpath('//*[@id="app-
wrapper"]/div/div[4]/div/form/input[3]')
        submitButton.click()
        loginError =
self.driver.find_element_by_xpath('//*[@id="app-wrapper"]/div/div[3]')
        assert loginError.is_displayed()

```

But as the web application continues to grow and more functionality needs to be tested, the test suite can start to become a mess of unreadable, fragile code (especially when you have to locate web elements using lengthy xpaths). Fortunately, a design pattern has emerged to help us write robust tests that are easy to read and maintain.

Enter Page Object Model Page Object Model (POM) is an organized approach to testing web applications wherein pages are represented in the tests. The page objects contain the elements and behaviors of the associated page. This design works well because it is clear what is being tested and limits the user to actions available on that page.

By separating the tests from the underlying data, not only is it now easier to write test cases, but you also only need to modify code in one place to update all tests. If the login error message changed to a different location, you would only need to update the LoginPage class and all the tests would still run fine. There is extra setup required to model the pages, but that time is quickly recovered when you get to creating your test suite.

```
class loginTests(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.get('http://somewebapp.com/')

    def tearDown(self):
        self.driver.close()

    def test_login_incorrect_password(self):
        login_page = page.LoginPage(self.driver)
        login_page.login('test@email.com', 'password123')
        assert login_page.login_error_displayed()
```

Compare this test to the one at the top. It's shorter and reads like a test case would, so you know what is being tested without knowing anything about the code. This is all possible because of the initial design and planning that went into mapping the login page

```
from selenium.webdriver.common.by import By
```

```
class BasePage(object):  
    def __init__(self, driver):  
        self.driver = driver  
    self.driver.implicitly_wait(5)  
    self.timeout = 30
```

```
class LoginPage(BasePage):
```

```
    email = (By.ID, 'email_Id')  
    password = (By.ID, 'password_Id')  
    loginError = (By.XPATH, '//*[@id="app-wrapper"]/div/div[3]')  
    submitButton = (By.XPATH, '//*[@id="app-wrapper"]/div/div[4]/div/form/input[3]')
```

```
    def set_email(self, email):  
        emailElement = self.driver.find_element(*LoginPage.email)  
        emailElement.send_keys(email)
```

```
    def login_error_displayed(self):  
        notifcationElement = self.driver.find_element(*LoginPage.loginError)  
        return notifcationElement.is_displayed()
```

```
    def set_password(self, password):  
        pwordElement = self.driver.find_element(*LoginPage.password)  
        pwordElement.send_keys(password)
```

```
    def click_submit(self):  
        submitBtn = self.driver.find_element(*LoginPage.submitButton)  
        submitBtn.click()
```

```
    def login(self, email, password):  
        self.set_password(password)  
        self.set_email(email)  
        self.click_submit()
```

Because the login page is being represented as a class, writing the actual tests will take less effort. Using POM structure allows you to write tests using page actions without having to worry about the logic behind them. Now that you have a LoginPage object, you would be able to write Login Page tests quick and easy, for example:

```

def test_login_incorrect_email(self):
    login_page = page.LoginPage(self.driver)
    login_page.login('test123@email.com', 'password')
    assert login_page.login_error_displayed()

def test_login_blank_password(self):
    login_page = page.LoginPage(self.driver)
    login_page.login('test@email.com', '')
    assert login_page.login_error_displayed()

def test_login_blank_email(self):
    login_page = page.LoginPage(self.driver)
    login_page.login('', 'password')
    assert login_page.login_error_displayed()

```

The more the better POM really starts to shine when you have a lot to do. Combinations of data input to verify, multiple steps that need to be completed, or different pages that you need to navigate to. You start to see this in Login method of the LoginPage. Multiple steps are being wrapped together to make a singular cohesive action of logging in.

Often tests will need to span several pages in order to verify the result. Tests are able to navigate through pages in POM by passing the webdriver into the new page object. By modifying the above Login method to return a HomePage object:

```

def login(self, email, password):
    self.set_password(password)
    self.set_email(email)
    self.click_submit()
    return HomePage(self.driver)

```

You can now start writing tests for the next page, where all the necessary steps to get to the Home Page will be one succinct line of code.

```

def test_somethingOnHomePage(self):
    loginpage = page.LoginPage(self.driver)
    homepage = loginpage.login(email,password)
    homepage.doSomethingOnHomepage()
    assert homepage.somethinghappened()

```

The POM structure also provides help with debugging. The more exception handling and logging you add to your page class, the easier it is to pinpoint where exactly a test went wrong. This may seem like a redundant point, but it really highlights the two advantages that come with using page objects:

Readability

Writing tests shouldn't have to include any kind of exception handling or low-level webdriver commands. These tests are easy to read and to write, even to a tester who is new to the project. In fact, if you are writing the tests in an IDE, you will have auto-complete on all of the different actions a page can do.

Maintainability

Creating a page class with exception handling on every component means you get them for free every time you bring the page into a test. Keeping tests and data separate from each other is essential to having a stable test suite. Modifying data in one place allows the test to adapt to a changing environment.