Wat is een Page Object Model (POM)?
Wanneer je een training Selenium WebDriver volgt, wordt vaak dit “design pattern” toegepast als zijnde een good practice. Zo ook bij de Selenium WebDriver training die door Polteq wordt gegeven. Maar de vraag is eigenlijk: is dit de enige oplossing?

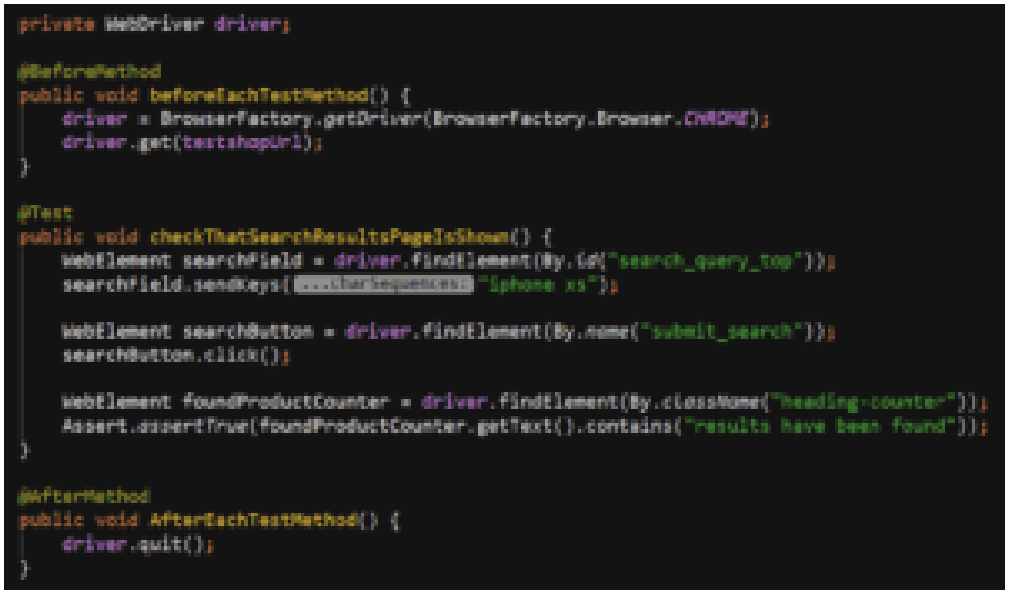
Om wat meer kader te scheppen, gaan we even terug naar de start van een beginnend testautomatiseerder die graag tests voor een website wil automatiseren. Als voorbeeld nemen we een testshop waarin we producten kunnen zoeken en bestellen. Hieronder een (Java) voorbeeld waarin de volgende stappen worden uitgevoerd:

- de browser wordt opgestart;
- de testshop pagina wordt geopend;
- zoek “iphone xs”;
- controleer of de zoekresultatenpagina verschijnt;
- sluit de browser.

```
@Test
public void checkThatSearchResultsPageIsShown() {

    WebDriver driver = BrowserFactory.getDriver(BrowserFactory.Browser.CHROME);

    driver.get(testshopUrl);

    WebElement searchField = driver.findElement(By.id("search_query_top"));
    searchField.sendKeys("iphone xs");

    WebElement searchButton = driver.findElement(By.name("submit_search"));
    searchButton.click();

    WebElement foundProductCounter = driver.findElement(By.className("heading-counter"));
    Assert.assertTrue(foundProductCounter.getText().contains("results have been found"));

    driver.quit();
}
```

Dit is een simpel voorbeeld van een test met Selenium WebDriver waarin alle testcode in één testmethode staat. Nadeel van deze aanpak is dat wanneer de “assert” (check) aan het einde faalt, de browser niet gesloten zal worden. Dit is niet gewenst, want een geautomatiseerde test moet altijd de browser sluiten.

Dit wordt opgelost door gebruik te maken van de mogelijkheden van het gebruikte testframework. Testframeworks beschikken over de mogelijkheid om stukjes testcode vóór en ná elke test uit te voeren. De code ná elke test wordt altijd uitgevoerd, ook als de check faalt. Dat heeft als voordeel dat de browser dus altijd gesloten wordt.

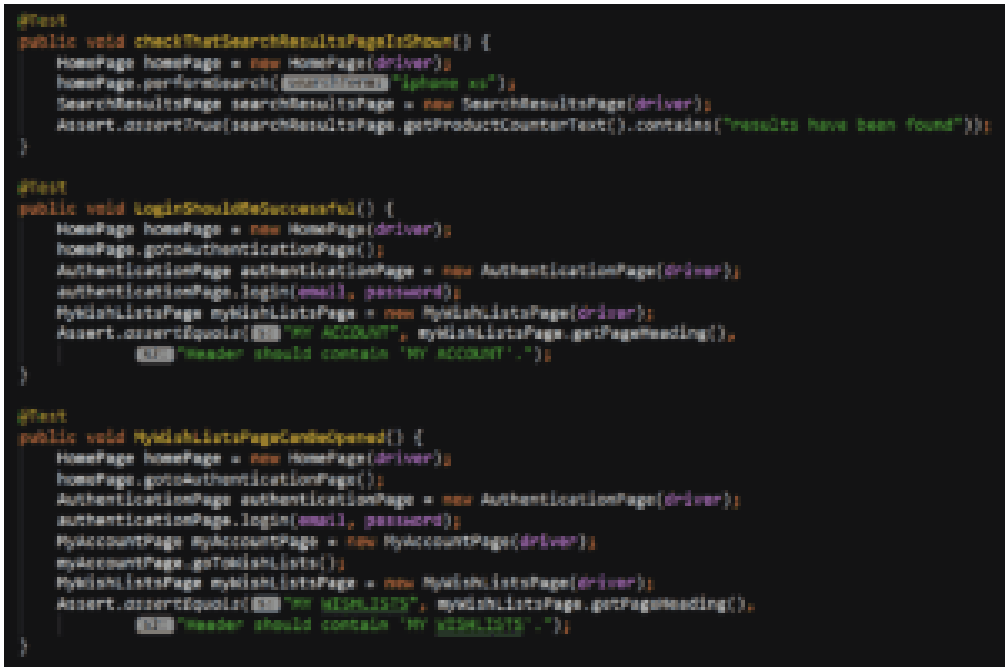


```
private WebDriver driver;

@BeforeMethod
public void beforeEachTestMethod() {
    driver = BrowserFactory.getDriver(BrowserFactory.Browser.CHROME);
    driver.get(testshopUrl);
}

@Test
public void checkThatSearchResultsPageIsShown() {
    WebElement searchField = driver.findElement(By.id("search_query_top"));
    searchField.sendKeys("Iphone xs");

    WebElement searchButton = driver.findElement(By.name("submit_search"));
    searchButton.click();

    WebElement foundProductCounter = driver.findElement(By.className("heading-counter"));
    Assert.assertTrue(foundProductCounter.getText().contains("results have been found"));
}

@AfterMethod
public void AfterEachTestMethod() {
    driver.quit();
}
```

Wanneer we nu naar de testcode kijken dan zien we dat er in de testcode veel locators staan. Locators zijn bijvoorbeeld “search_query_top” en “submit_search”. Stel we hebben 10 tests, dan zouden die ook in 10 tests voor kunnen komen. Wanneer een locator wijzigt naar aanleiding van een aanpassing van een ontwikkelaar, dan zouden dus ook 10 tests aangepast moeten worden om weer de goede locator te gebruiken. Dat is niet wenselijk en wordt opgelost door gebruik te maken van het Page Object Model. In het kort betekent dit dat pagina’s in een aparte class worden beschreven waardoor de locators maar op één plek staan. Een aanpassing aan een locator hoeft dan nog maar op één plek te gebeuren.

In de testcode kun je zien dat er gebruik gemaakt wordt van pagina-objecten waarmee je acties op een web pagina uit kunt voeren. De daadwerkelijke Selenium code en locators staan in het page object

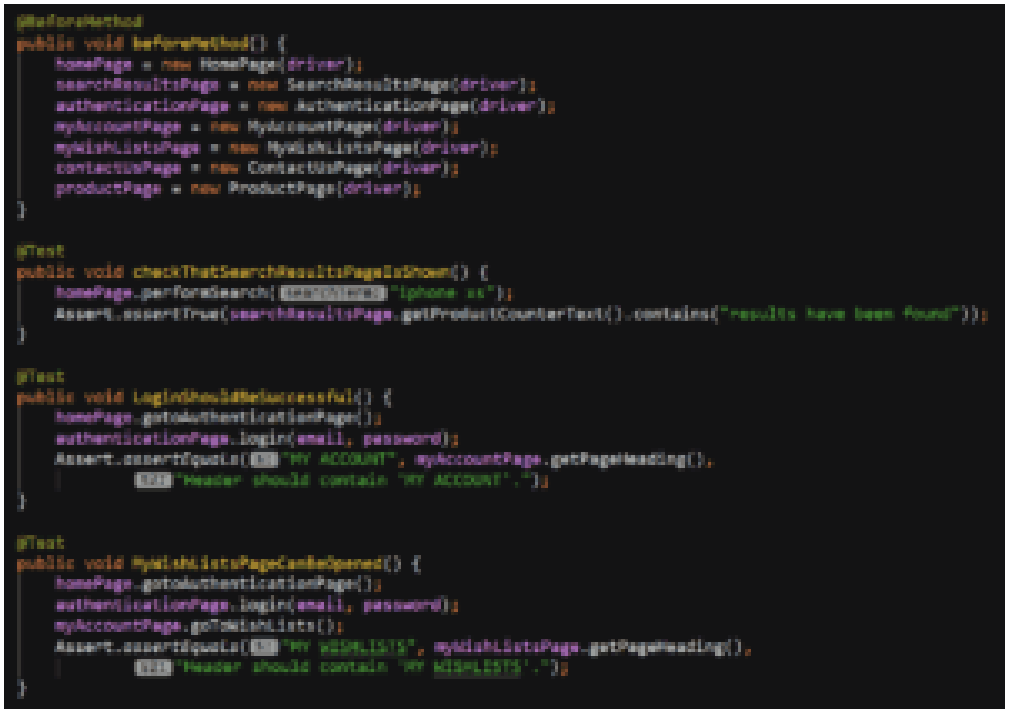
Wanneer we dan vervolgens meerdere tests hebben, dan ziet dat er uit als:



Dit zijn drie tests die elk hun eigen page objects instantiëren en gebruiken. Nette testcode

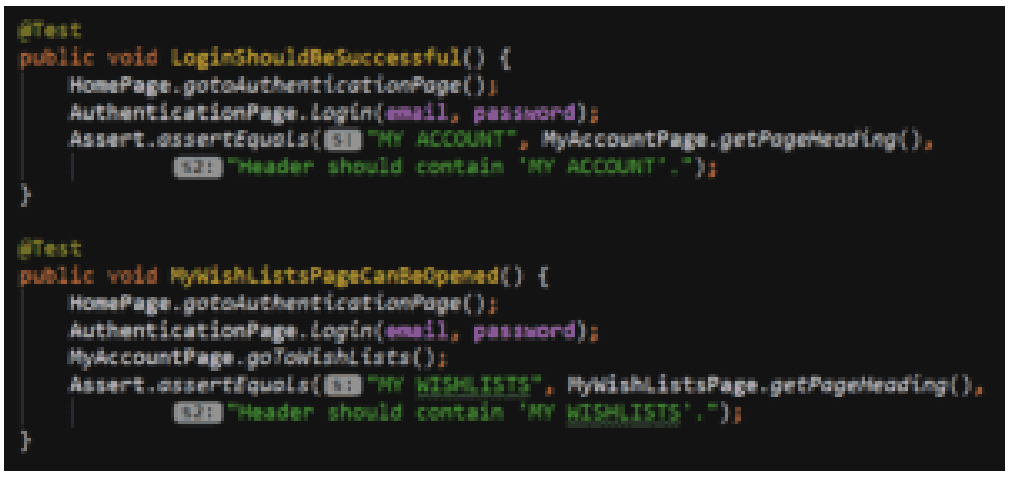
waarin geen Selenium code meer te zien is.

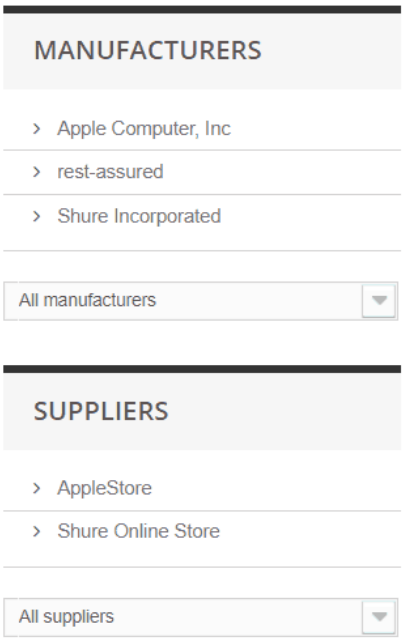
Een aanpak om het instantiëren van de pagina objecten te verminderen, is door het instantiëren in de “BeforeMethod” te zetten. Dan hoeft het maar één keer gedaan te worden



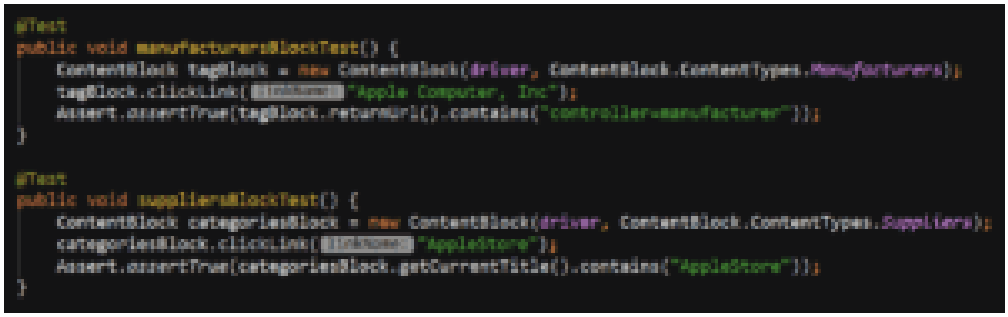
Hierdoor zien de tests er dan weer veel compacter uit en zijn beter leesbaar. Nadeel is echter dat er voor een test pagina’s geïnstantieerd worden die niet gebruikt worden in de test.

Als vergelijkbare oplossing zou je kunnen kiezen voor een static Page Object Model. Dit betekent dat de pages niet meer geïnstantieerd hoeven te worden en dat dit dus niet meer in de before hoeft te gebeuren. De afbeelding hieronder laat zien hoe dit er uit ziet.





Dit ziet er bijna hetzelfde uit als de vorige oplossing met het verschil dat er nu rechtstreeks naar de class verwezen wordt (HomePage) in plaats van de instantie (homePage).
Deze aanpak raden we af om te gebruiken omdat er veel nadelen aan vast zitten. Het is dan bijvoorbeeld niet meer mogelijk om de tests parallel uit te voeren.
Wanneer we echter naar het testframework WebdriverIO voor JavaScript kijken, dan lijkt het er echter op dat daar wél gebruik gemaakt wordt van een static POM.



Ook hier wordt verwezen naar, zoals het lijkt, de class “HomePage” in plaats van de instantie “homePage”. Voor deze combinatie geldt echter dat de combinatie JavaScript en WebdriverIO ervoor zorgen dat instantiëren niet expliciet nodig is. Tests worden door WebdriverIO steeds in een eigen “sessie” gestart. Hierdoor is parallel testen zonder instantiëren mogelijk.

Tot nu toe hebben we het steeds over complete pagina's gehad voor de Page Objects. Het kan echter ook voorkomen dat er op een pagina repeterende blokken staan.
Dit is een voorbeeld van een "MANUFACTURERS" en "SUPPLIERS" blok. Beide staan op één pagina en hebben dezelfde soort opbouw qua HTML code. In dit geval gaan we dit niet voor die pagina helemaal uitschrijven in één page object, maar gaan we gebruikmaken van Page Blocks. Een generiek stukje page object alleen voor de beschrijving van het blok.

In de testcode in de afbeelding hieronder kun je zien dat dezelfde class ContentBlock gebruikt wordt als beschrijving van een stukje page. Beide tests verwijzen echter naar een ander gedeelte van de pagina. Je hoeft dus niet altijd een complete pagina te beschrijven in je Page Object.

```
public AuthenticationPage openAuthenticationPage() {
    authenticationLink.click();
    return new AuthenticationPage(driver);
}
```

Als laatste bespreken we de Fluent Page Objects. Deze aanpak kom je ook tegen als je op zoek gaat naar het Page Object Model. In een Fluent Page Object wordt er in het Page Object rekening gehouden met waar de uitgevoerde actie terecht komt.
Als voorbeeld nemen we het selecteren van de login link op de homepage. Deze actie zorgt ervoor dat je op de inlogpagina terechtkomt (AutenticationPage).

```
public AuthenticationPage openAuthenticationPage() {
    authenticationLink.click();
    return new AuthenticationPage(driver);
}
```

De openAuthetication Page is een actie (methode) die je op de homepage uit kunt voeren. Deze methode retourneert aan het eind een nieuwe instantie van de AuthenticationPage.

Wanneer we dan de volgende "normale POM" test bekijken (afbeelding hieronder), kan dat met Fluent Page Objects beschreven worden als in de tweede afbeelding hieronder wordt getoond.

```java
@Test
public void LoginShouldBeSuccessful() {
    homePage.openAuthenticationPage();
    authenticationPage.login(email, password);
    Assert.assertEquals( m: "MY ACCOUNT", myAccountPage.getPageHeading(),
            s2: "Header should contain 'MY ACCOUNT'.");
}
```

```java
@Test
public void LoginShouldBeSuccessful() {
    myAccountPage = new HomePage(driver)
            .openAuthenticationPage()
            .login(email, password);

    Assert.assertEquals( m: "MY ACCOUNT", myAccountPage.getPageHeading(),
            s2: "Header should contain 'MY ACCOUNT'.");
}
```

Vanuit het aanmaken van een nieuwe homepage wordt direct “authentication page” geopend, waarna direct wordt ingelogd. De login methode geeft weer de accountpagina (MyAccountPage) terug die verschijnt als je inlogt.

Zo zie je dat je met het Page Object Model verschillende aanpakken kunt hanteren. Hierboven zijn er enkele beschreven en er zullen nog veel meer aanpakken mogelijk zijn. Welke je uiteindelijk kiest, hangt af van welke (soort) website je aan het automatiseren bent.

Erik Haartmans en Martijn Thiele