Selenium, использование POM


```
Selenium –  ,           (open source):

Selenium WebDriver
Selenium RC
Selenium Server
Selenium Grid
Selenium IDE
```

Называть просто словом Selenium любой из этих пяти продуктов, вообще говоря, неправильно, хотя так часто делают, если из контекста понятно, о каком именно из продуктов идёт речь, или если речь идёт о нескольких продуктах одновременно, или обо всех сразу.

Selenium WebDriver

Selenium WebDriver – это программная библиотека для управления браузерами. Часто употребляется также более короткое название WebDriver.

Иногда говорят, что это «драйвер браузера», но на самом деле это целое семейство драйверов для различных браузеров, а также набор клиентских библиотек на разных языках, позволяющих работать с этими драйверами.

Это основной продукт, разрабатываемый в рамках проекта Selenium.

Как уже было сказано, WebDriver представляет собой семейство драйверов для различных браузеров плюс набор клиентских библиотек для этих драйверов на разных языках программирования:

- AndroidDriver,
- ChromeDriver,
- FirefoxDriver,
- InternetExplorerDriver,
- iPhoneDriver
- OperaDriver

Selenium RC

Selenium RC – это предыдущая версия библиотеки для управления браузерами. Аббревиатура RC в названии этого продукта расшифровывается как Remote Control, то есть это средство для «удалённого» управления браузером.

Эта версия с функциональной точки зрения значительно уступает WebDriver. Сейчас она находится в законсервированном состоянии, не развивается и даже известные баги не исправляются. А всем, кто сталкивается с ограничениями Selenium RC, предлагается переходить на использование WebDriver.

Selenium Server

Selenium Server – это сервер, который позволяет управлять браузером с удалённой машины, по сети. Сначала на той машине, где должен работать браузер, устанавливается и запускается сервер. Затем на другой машине (технически можно и на той же самой, конечно) запускается программа, которая, используя специальный драйвер RemoteWebDriver, соединяется с сервером и отправляет ему команды. Он в свою очередь запускает браузер и выполняет в нём эти команды, используя драйвер, соответствующий этому браузеру.

Selenium Grid

Selenium Grid – это кластер, включающий в себя несколько Selenium-серверов. Он позволяет организовать распределённую сеть, позволяющую параллельно запускать много браузеров на разных машинах.

Selenium Grid работает следующим образом: имеется центральный сервер (hub), к которому подключены узлы (node). Работа с кластером осуществляется через hub, при этом он просто транслирует запросы узлам. Узлы могут быть запущены на той же машине, что и hub или на других.

Сервер и узлы могут работать под управлением разных операционных систем, на них могут быть установлены разные браузеры. Одна из задач Selenium Grid заключается в том, чтобы «подбирать» подходящий узел по типу браузера, версии, операционной системы и другим атрибутам, заданным при старте браузера.

Selenium IDE

Selenium IDE – плагин к браузеру Firefox, который может записывать действия пользователя, воспроизводить их, а также генерировать код для WebDriver или Selenium RC, в котором выполняются те же самые действия. В общем, это «Selenium-рекордер».

Тестировщики, которые не умеют (или не хотят) программировать, используют Selenium IDE как самостоятельный продукт, без преобразования записанных сценариев в программный код. Это, конечно, не позволяет разрабатывать достаточно сложные тестовые наборы, но некоторым хватает и простых линейных сценариев.

Паттерн POM в Selenium Webdrive

Page Object Model – это шаблон проектирования, который широко используется в автоматизированном тестировании и позволяет разделять логику выполнения тестов от их реализации.

POM помогает решить следующую проблему:

Рассмотрим простой пример логина на сайт:

```
public class NoPOMTestAutoQALogin {

@Test(priority=0)
public void test_Home_Page_Appear_Correct(){
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://autoqa.pp.ua/wp-login.php");
//Find user name and fill user name
driver.findElement(By.id("user_login")).sendKeys("subscriber");
//find password and fill it
driver.findElement(By.id("user_pass")).sendKeys("2016subscriberpasssword2016ok");
//click login button
driver.findElement(By.id("wp-submit")).click();
String homeText = driver.findElement(By.xpath("//div[@id='profile-page']/h2")).getText();
//verify login success
Assert.assertTrue(homeText.toLowerCase().contains("profile"));
}
}
```

Как можно заметить, все, что мы делаем, – это находим элементы на странице и делаем над ними определенные действия.

Это небольшой скрипт. Его поддержка выглядит простой, но с увеличением количества тестов, добавляя все больше строк кода, – она становится сложнее.

Наибольшей проблемой поддержки есть то, что 10 разных скриптов используют тот же самый код. И изменения одного элемента (например, изменение верстки сайта девелоперами), приведет к изменению всех 10 тестов.

Более лучший подход в поддержке тестов – это создание отдельного класса, который будет находить элементы на странице, заполнять или проверять их. Класс может использоваться в коде тестов, использующих элементы этого класса. В будущем в случае изменений нам необходимо будет только сделать правки в одном классе, а не во всех 10 тестах.

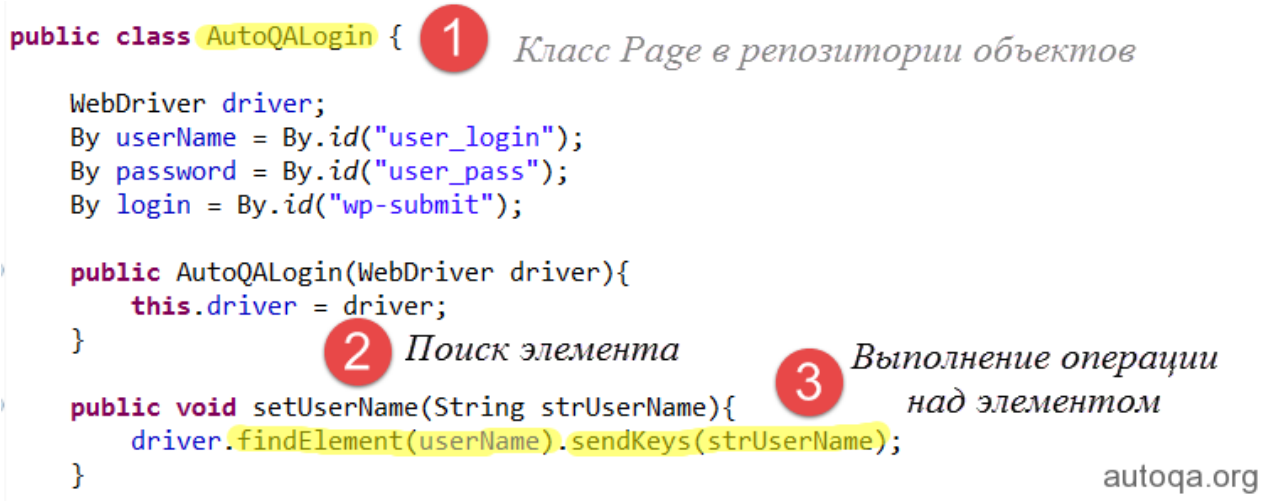
Этот подход называется **Page Object Model(POM)**. POM паттерн делает код более читабельным, поддерживаемым и повторно используемым.

Преимущества POM

1. Page Object Patten объявляет элементы отдельно от реализации теста. Эта концепция делает наш код более понятным.
2. Вторым преимуществом есть то, что независимость класса объектов от реализации теста позволяет использовать этот репозиторий в разных целях и с разными для выполнения тестов. Например, мы можем интегрировать POM с **TestNG**/**JUnit** для функционального тестирования, а также с **JBehave**/**Cucumber** для приемочного тестирования.
3. Кода становится меньше и он более оптимизирован. Его можно повторно использовать.
4. **Методы** получают более реальные имена и отображают выполненное действие на UI, например**,** *gotoHomePage()*.

Пример использования POM

Ниже представлена базовая структура Page object model (POM), где все элементы и методы вынесены в отдельный класс. Операции, такие как проверка должны быть отделены от тестового метода.



Рассмотрим на примере

Мы будем работать над 2 страницами:

1. Страница входа
2. Главная страница

Согласно этому мы создадим 2 POM класса:

Класс **AutoQALogin:**

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;


public class AutoQALogin {

WebDriver driver;
By userName = By.id("user_login");
By password = By.id("user_pass");
By login = By.id("wp-submit");

public AutoQALogin(WebDriver driver){
this.driver = driver;
}

//Set user name in textbox
public void setUserName(String strUserName){
driver.findElement(userName).sendKeys(strUserName);
}

//Set password in password textbox
public void setPassword(String strPassword){
driver.findElement(password).sendKeys(strPassword);
}

//Click on login button
public void clickLogin(){
driver.findElement(login).click();
}

/**
* This POM method will be exposed in test case to login in the application
* @param strUserName
* @param strPasword
* @return
*/
public void loginToAutoQA(String strUserName,String strPasword){
//Fill user name
this.setUserName(strUserName);
//Fill password
this.setPassword(strPasword);
//Click Login button
this.clickLogin();
}
}
```

Класс **AutoQAHomePage**:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class AutoQAHomePage {

WebDriver driver;
By homePageName = By.xpath("//div[@id='profile-page']/h2");

public AutoQAHomePage(WebDriver driver){
this.driver = driver;
}

//Get the Page name from Home Page
public String getHomePageDashboardName(){
return driver.findElement(homePageName).getText();
}
}
```

Класс тестов **TestAutoQALogin** :

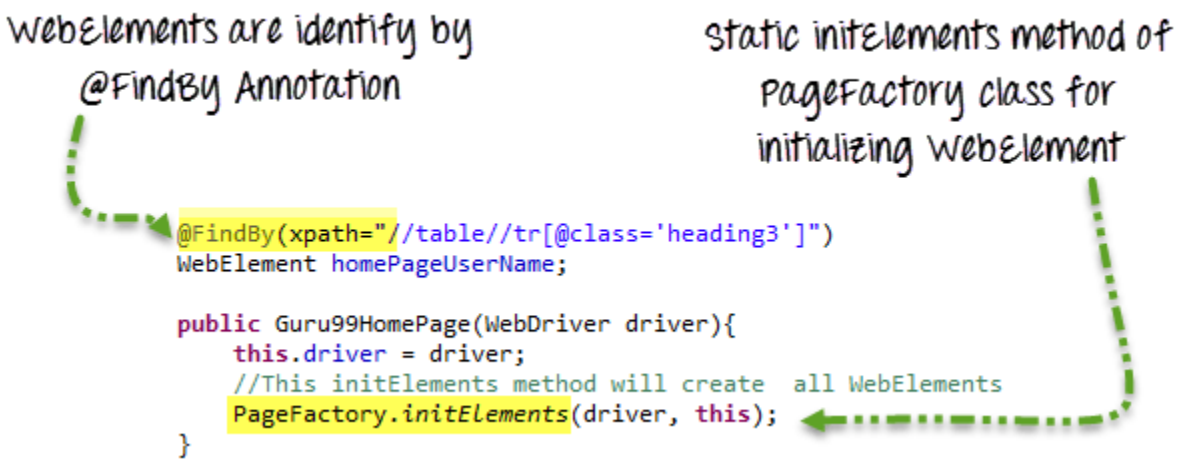
```
import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.Assert;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import PageFactory.AutoQAHomePage;
import PageFactory.AutoQALogin;

public class TestAutoQALogin {

WebDriver driver;
AutoQALogin objLogin;
AutoQAHomePage objHomePage;

@BeforeTest
public void setup(){
driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://autoqa.pp.ua/wp-login.php");
}

/**
* This test case will login in http://autoqa.pp.ua/wp-login.php
* Login to application
* Verify the home page using Dashboard message
*/
@Test(priority=0)
public void test_Home_Page_Appear_Correct(){
//Create Login Page object
objLogin = new AutoQALogin(driver);
//login to application
objLogin.loginToAutoQA("subscriber", "subscriberpass");
// go the next page
objHomePage = new AutoQAHomePage(driver);
//Verify home page
Assert.assertTrue(objHomePage.getHomePageDashboardName().toLowerCase().contains("profile"));
}
}
```

Конечный проект выглядит следующим образом:

Page Factory

Page Factory - это встроенная концепция объектной модели страницы из библиотеки Selenium, но она очень оптимизирована.

Здесь мы также следуем концепции разделения объектов страниц и методов тестирования. Кроме того, с помощью класса PageFactory мы используем аннотации **@FindBy** для поиска WebElement, а также метод **initElements** для инициализации веб-элементов.



@FindBy может принимать в качестве **атрибутов tagName ,partalLinkText, name, linkText, id, css, className, xpath** .