# ****How to build a maintainable Selenium framework?****

Below is the outline of the major steps in building a maintainable Selenium framework.

1. [Choose a programming language](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section1)
2. [Choose a unit test framework](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section2)
3. [Design the framework architecture](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section3)
4. [Build the SeleniumCore component](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section3.1)
5. [Build the SeleniumTest component](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section3.2)
6. [Choose a reporting mechanism](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section4)
7. [Decide how to implement CI/CD](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section5)
8. [Integrate your framework with other tools](https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/" \l "section6)

As the blog post progresses, we’ll also include some best practices that you can apply to your project. Most importantly, as you read, try to get hands on and apply the best practices as much as possible.

## ****Choose a programming language****

### If you can code…

Your programming language of choice has a impact to your framework design & productivity. Thus the very first question you should ask is: ****In what programming language do I want to write my tests?****

The most popular languages among the Selenium community are Java, Python and JavaScript. To decide which programming language you should pick, consider the below factors.

1. ****What programming language is being used to develop the web apps you need to test?****
2. ****Does your company have an in-house framework that you can reuse?****
3. ****Who will use your framework to write tests?****

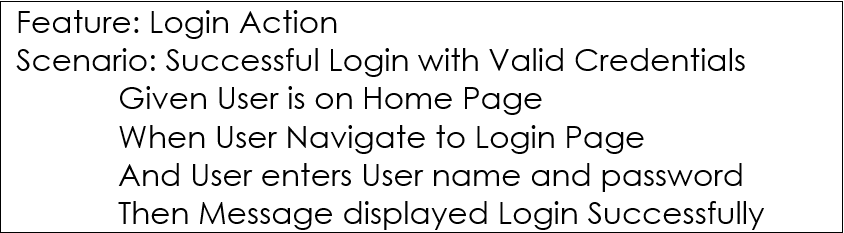
From our experience, ****Java**** is the safest choice if you start a new project from scratch since it is widely adopted by the community due to the fact that it works across platforms. Moreover, you can easily find code examples or troubleshooting tips if you get stuck. Java is also the top priority for each new release of Selenium.

### If you are not good at code…

The good news is: you can also write Selenium tests using the famous ****Behavior-Driven Development (BDD)**** method. But that would require some additional setup.

In brief, BDD helps boost the readability of your tests by structuring a test flow into ****Given****, ****When****, and ****Then**** (GWT) statements. As a result, not only test automation engineers with programming skills but also domain experts and business testers can understand the tests and contribute meaningfully to the process of test creation, test result debugging, and test maintenance.

The picture below shows an example of a test written in BDD.



Some tools that you can leverage if you choose BDD:

* [Cucumber](https://www.guru99.com/using-cucumber-selenium.html" \t "https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/_blank) (support for most major languages)
* [SpecFlow](http://toolsqa.com/specflow/specflow-tutorial/" \t "https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/_blank) (for C#)

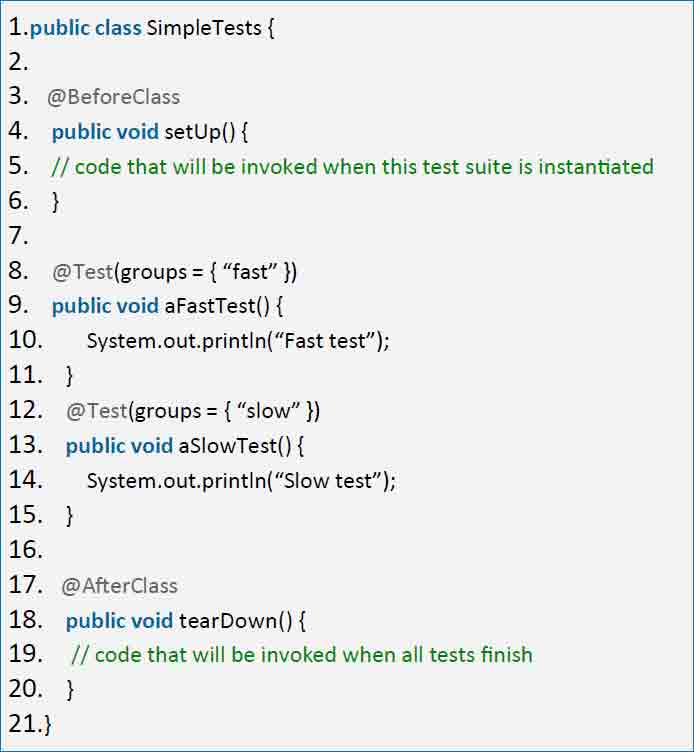
In our opinion, BDD is suitable for small or short-term projects. It’ll be hard to scale if you have to write a dozen of “And/And/And…” statements using the GWT syntax. A more mature method for your consideration is the **[Keyword-Driven Testing](https://www.testarchitect.com/resources/white-papers/the-art-of-keyword-driven-testing?_ga=2.223137880.2114611785.1562123736-312532252.1562123736)**method (KDT). Check out this blog post: [Keyword-Driven Testing: The Best Practices You Can’t Afford to Miss](https://www.logigear.com/blog/test-methods/keyword-driven-testing-the-best-practices-you-cant-afford-to-miss/)

## ****Choose a unit test framework****

Now we’ve selected the most suitable programming language, we now need to pick a unit test framework that we will build our framework upon. Since we already chose the Java language to write tests, I’d recommend ****TestNG**** since it offers several important benefits, such as:

* ****TestNG**** is similar to JUnit, but it is much more powerful than JUnit—especially in terms of testing integrated classes. And better yet, TestNG inherits all of the benefits that JUnit has to offer.
* ****TestNG**** eliminates most of the limitations of the older frameworks and gives you the ability to write more flexible and powerful tests. Some of the highlight features are: easy annotations, grouping, sequencing, and parameterizing.

The below code snippet shows an example of two TestNG tests. Both tests share the same ****setUp()**** and ****teardown()**** methods thanks to the ****@BeforeClass**** and ****@AfterClass**** annotations.

*Figure 1 – TestNG annotations*

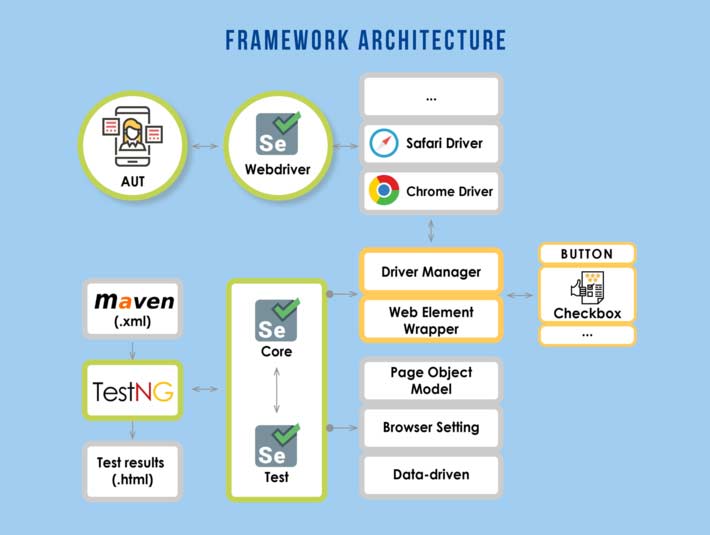
You can think of a test class as a logical grouping of some automated test cases that share the same goals, or at least the same area of focus.

For instance, you can group automated test cases that focus on verifying whether the app calculates the total price of a shopping cart correctly into a test class named ****TotalPriceCalculation****. These tests probably share the same initial setup of navigating to the ecommerce site under test and the tear down steps of clearing the items in the cart.

With TestNG, you can also group tests inside one test classes into sub-groups using the ****@Test**** annotations as demonstrated in the code snippet.

## ****Design the framework architecture****

Now, it’s time to take a look at our framework’s architecture. After many big and small Selenium projects at [LogiGear](https://www.logigear.com/), we’ve come up with a sustainable, maintainable, and scalable architecture shown in the diagram below. We highly recommend that you follow this architecture or at least the core principles behind it.

*Figure 2 – Framework Architecture*

The beauty of this architecture comes from the fact that there are two separate components called [1] ****SeleniumCore****, and [2] ****SeleniumTest****. We’ll explain those components in detail in the following sections. In brief, having two decoupled components simplifies test maintenance in the long run.

For instance, if you want to check whether an <input> tag is visible on screen before clicking on it, you can simply modify the “input” element wrapper and that change will be broadcasted to all test cases and page objects that interact with <input> tags.

Not having the tests and the element wrappers decoupled means you’ll have to update each and every test case or page object that are currently interacting with <input> tags whenever you want to introduce new business logics.

Now that we’ve had an overview of the framework, we’ll examine how to build each component in the upcoming sections of this post.

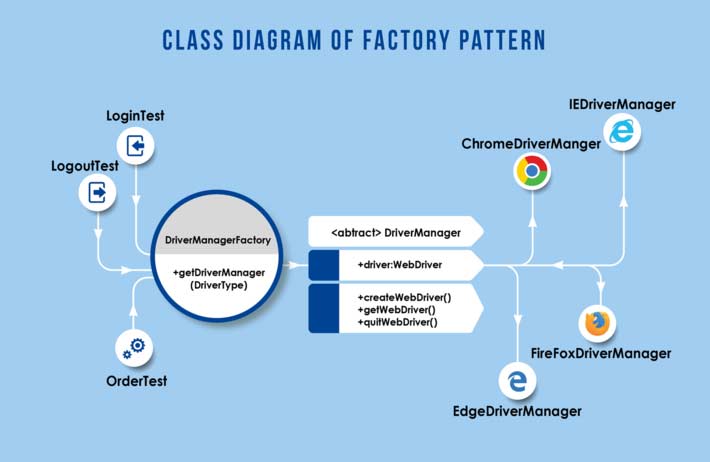
****PART 2****

### ****Build the SeleniumCore component****

****SeleniumCore**** is designed to manage the browser instances as well as element interactions. This component helps you to create, and destroy WebDriver objects.

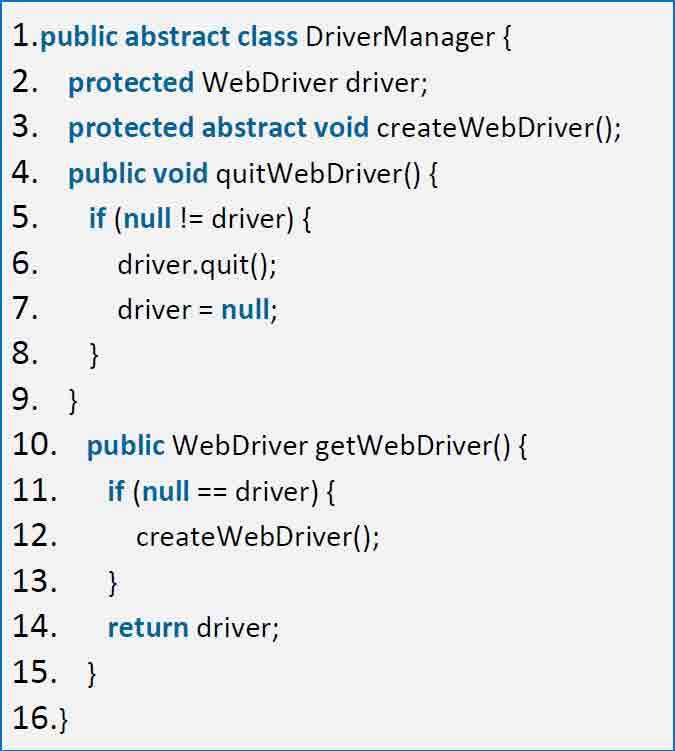
One WebDriver object, as its name suggests, “drives” a browser instance such as moving from web page to web page. Ideally, the test writers should not care about how the browser instances are created or destroyed. They just need a WebDriver object to execute a given test step in their test flow.

To achieve this kind of abstraction, we normally follow a best practice called the **[Factory](https://www.tutorialspoint.com/design_pattern/factory_pattern.htm)** design pattern. Below is a class diagram explaining how we use the Factory design pattern in our framework.

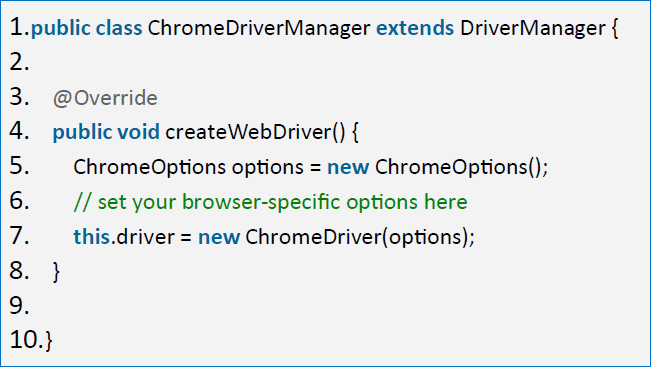
*Figure 3 – Class diagram of Factory pattern*

In the above diagram, LoginTest, LogoutTest and OrderTest are the test  
classes that “use” the ****DriverManagerFactory**** to “manufacture” ****DriverManager**** objects for them.

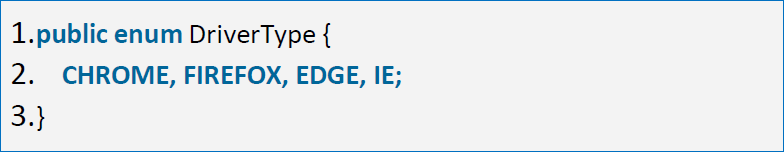
In the below code snippet, you will see that ****DriverManager**** is an abstract  
class, dictating that its implementations such as ChromeDriverManager, FirefoxDriverManager and EdgeDriverManager must expose a set of  
mandate methods such as ****createWebDriver(),**** ****getWebDriver(),**** and ****quitWebDriver()****.

*Figure 4 – DriverManager abstract class*

The below ****ChromeDriverManager**** implements the ****DriverManager****abstract class defined in the above snippet. Specifically, in the ****createWebDriver()**** method, we instantiate a new ****ChromeDriver**** with a set of predefined options. Likewise, we’ll do the same for ****FirefoxDriverManager****, ****EdgeDriverManager****, or any other browsers of your interest.

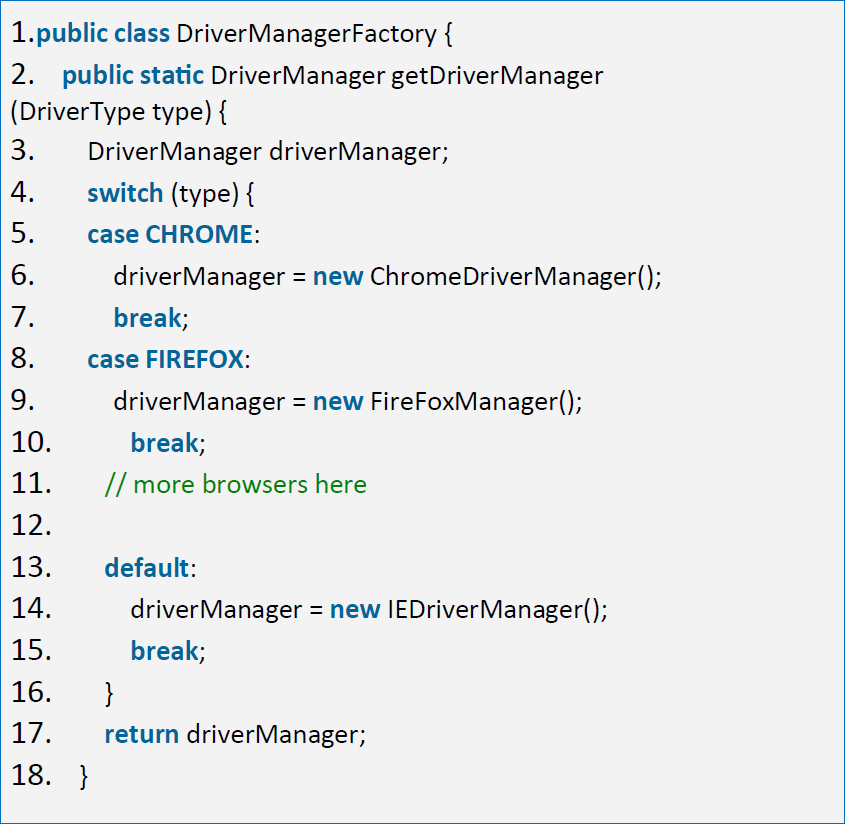
[](https://d1842wf5bcpgkm.cloudfront.net/blog/wp-content/uploads/code3.png)*Figure 5- ChromeDriverManager sample implementation*

To easily manage the browsers that our project focuses on, we define an enum called ****DriverType****which contains all browsers we ever want to test.

[](https://d1842wf5bcpgkm.cloudfront.net/blog/wp-content/uploads/code3.1.png)Figure 6 – DriverType enum

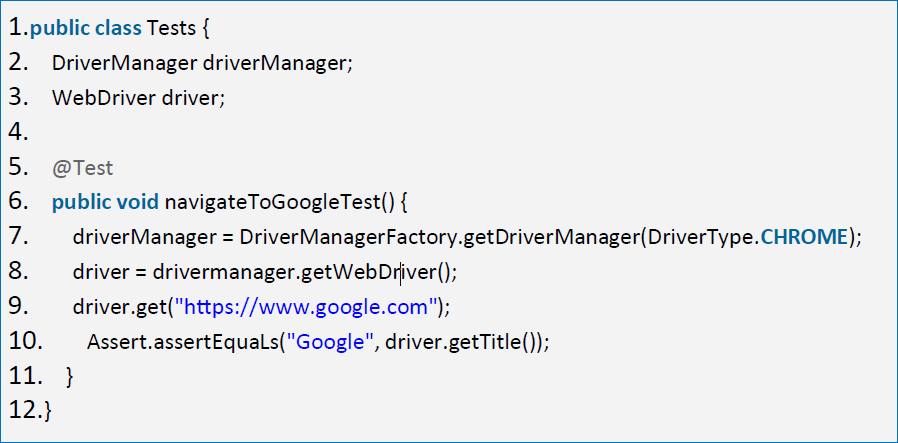
Like we previously mentioned, ****DriverManagerFactory**** is a factory that “manufactures” ****DriverManager**** objects. You invoke the ****getDriverManager()**** method of this class with your ****DriverType**** (described above) to receive a ****DriverManager****-type object.

Since ****DriverManager**** is an abstract class, you won’t receive an actual ****DriverManager****, just one of its implementations, such as ****ChromeDriverManager****, ****FireFoxDriverManager****, etc. The code snippet below demonstrates how to implement the ****DriverManagerFactory**** class.

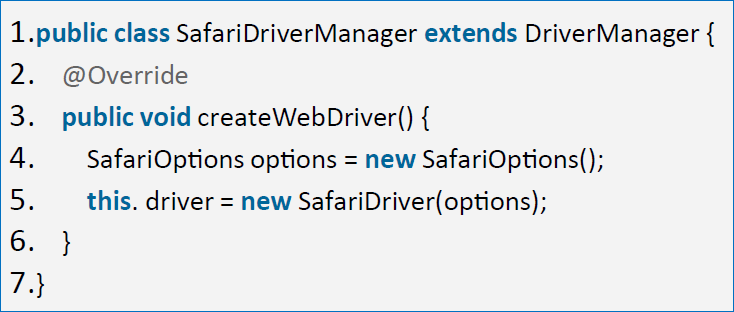
[](https://d1842wf5bcpgkm.cloudfront.net/blog/wp-content/uploads/code4.png)*Figure 7 – “Manufacture” DriverManager objects*

After understanding how a browser instance is created, we’ll now create a test using one of the above ****DriverManager**** objects. As you can see, the test writer doesn’t care whether the ****WebDriver**** for Chrome is called ****ChromeDriver**** or not. They only need to specify the simple ****CHROME**** string (one of the values in the ****DriverType**** enum) when they need a Chrome browser instance.

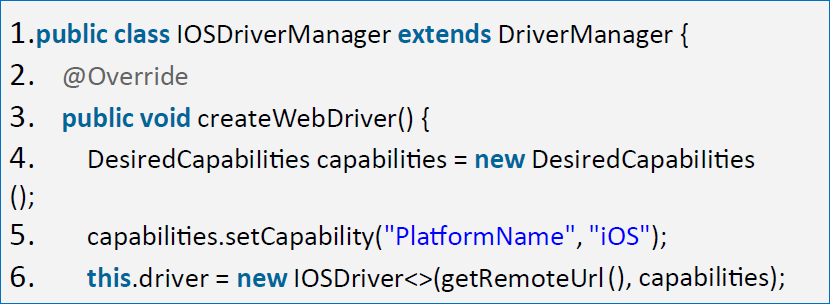
In the below test, we navigate to [www.google.com](http://www.google.com/" \t "https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/_blank) and verify that the site’s title is named “Google.” Not much of a test but it demonstrates how you we apply the aforementioned ****DriverManagerFactory****.

[](https://d1842wf5bcpgkm.cloudfront.net/blog/wp-content/uploads/code5-2.png)*Figure 8 – Sample test using DriverManagerFactory and DriverManager*

By using this Factory design pattern, if there is a new requirement to run tests on a new browser, say Safari for example, it should not be a big deal. We just need to create a ****SafariDriverManger****, which extends ****DriverManager**** exactly like the ****ChromeDriverManager**** we saw earlier. When it’s been created, test writers can simply create a ****SafariDriverManager**** using the new ****SAFARI**** value of the ****DriverType**** enum.

[](https://d1842wf5bcpgkm.cloudfront.net/blog/wp-content/uploads/code6.png)*Figure 9 – SafariDriverManager*

Similarly, it’s very easy to integrate with ****Appium**** when we need to run tests against a mobile native app or web app on mobile browsers. We can simply implement a new class so-called ****iOSDriverManager****.

[](https://d1842wf5bcpgkm.cloudfront.net/blog/wp-content/uploads/code7.png)*Figure 10 – IOSDriverManager*

### ****Build the SeleniumTest component****

Unlike the ****SeleniumCore**** component which plays the role of the foundation of the framework, ****SeleniumTest**** component contains all test cases that use the classes provided by ****SeleniumCore****. As we mentioned earlier, the design pattern we’ll apply here is called ****PageObject**** pattern (POM).

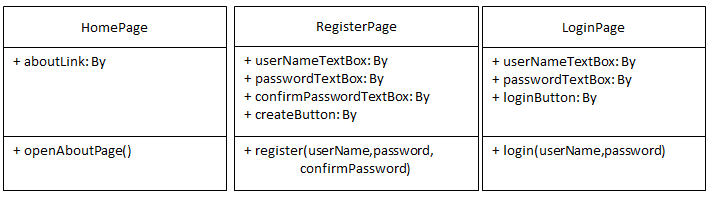
#### ****PAGEOBJECT PATTERN****

[Page Object Model](https://github.com/SeleniumHQ/selenium/wiki/PageObjects" \t "https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/_blank) (POM) has become the de-facto pattern used in test automation frameworks because it reduces duplication of code thus reduces the test maintenance cost.

Applying POM means we’ll organize the UI elements into pages. A page can also include “actions” or business flows that you can perform on the page. For instance, if your web app includes several pages called the Login page, Home page, Register page, etc., we’ll create the corresponding PageObjects for them such as ****LoginPage****, ****HomePage****, ****RegisterPage****, etc.

Thanks to POM, if the UI of any page changes, we will only need to update the PageObject in question once, instead of tiringly refactoring all tests that interact with that page.

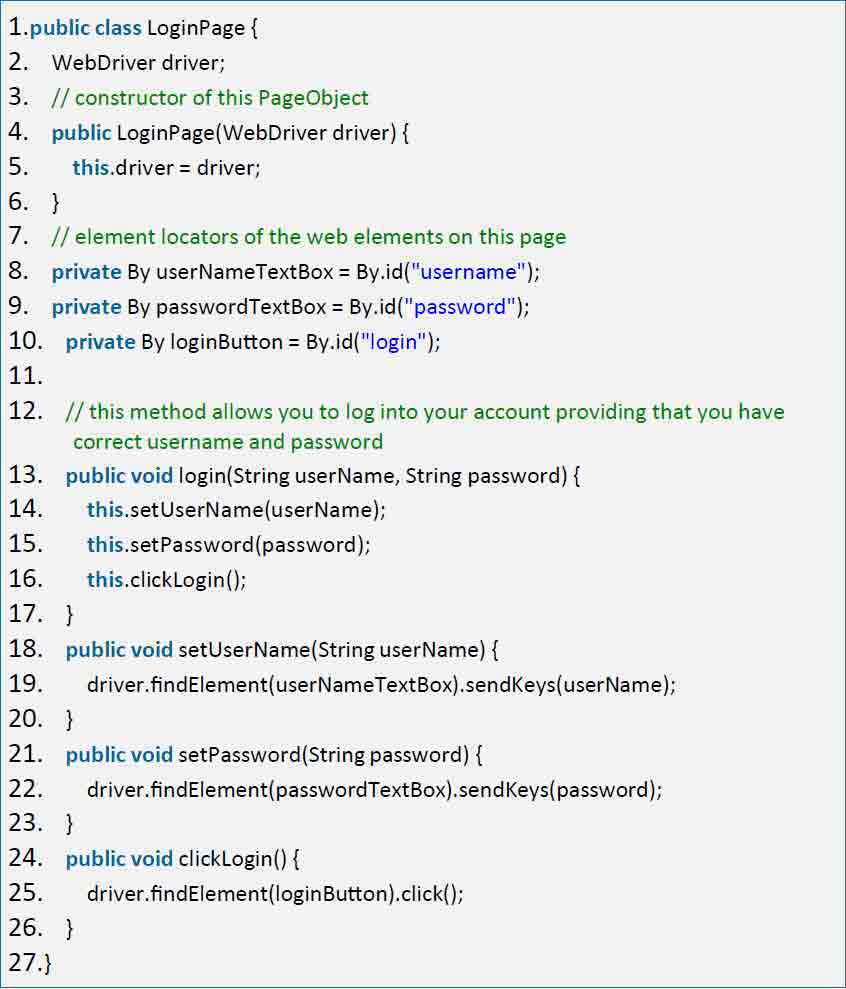
The picture below demonstrates how we usually structure PageObjects, their element locators as well as action methods. Note that although  ****RegisterPage**** and ****LoginPage**** both have ****userNameTextBox****and ****passwordTextBox****, these web elements are complete different. The ****userNameTextBox**** and ****passwordTextBox**** on the ****Register**** page are used to register a new account while the same set of controls on the ****Login**** page allow users to log into their accounts.

Figure 11 – Example of some Page objects

#### ****A SIMPLE PAGE OBJECT****

Let’s zoom into a specific Page object. In the below example, we see that the LoginPage contains several important pieces of information:

* A constructor that receives a WebDriver object and sets its internal WebDriver object to that object.
* The element locators that help the WebDriver object find the web elements you want to interact with. E.g. userNameTextBox
* Methods to perform on the Login page such as setUserName(), setPassword(), clickLogin(), and most importantly–l****ogin()**** method that combines all of the three methods above.

*Figure 12 – LoginPage page object*

#### ****HOW TO USE A PAGEOBJECT****

To interact with the Login page in our tests, we can simply create a new ****LoginPage**** object and call its action methods. Since we’ve abstracted away the web element definitions (locators) from the test writer, they are not required to know how to find an element, e.g. ****userNameTextBox****. They just  
call the ****login()**** method and pass in a set of username and password.

If the web element definitions happen to change, we do not need to update all of the tests interacting with this Login page.

[](https://d1842wf5bcpgkm.cloudfront.net/blog/wp-content/uploads/code9.png)Figure 13 – A sample test case to verify the Login page

As you might have already noticed, the goal of the test is to verify that the web app displays the correct error message (“*Invalid username or password*”) when a user tries to log in with an  
incorrect credential.

Note that, we have not included the ****getLoginErrorMessage()****action method in our previous code snippet since the implementation of this method could be complicated depending on how we design our web app. Normally, an error message would appear as a simple red-color string right next to the Login button.

In such a case, retrieving that error message would more straightforward. We’ll just need to define an element locator, e.g. ****errorMessageLabel = By.id(“errorMessage”))**** then create the ****getLoginErrorMessage()**** method using that locator.

*At this point, our Test Automation framework finally has a concrete foundation. We can now release it to the team so that everybody will contribute to the test development and test execution efforts. Part 3 will discuss how to add some more utilities to the framework to increase our productivity.*

****PART 3****

## ****Choose a reporting mechanism****

Hopefully we now scale up our volume of automated tests quickly and run them frequently enough to justify the upfront investment. As you run more and more tests, you’ll soon find that understanding test results will be difficult without a good reporting mechanism.

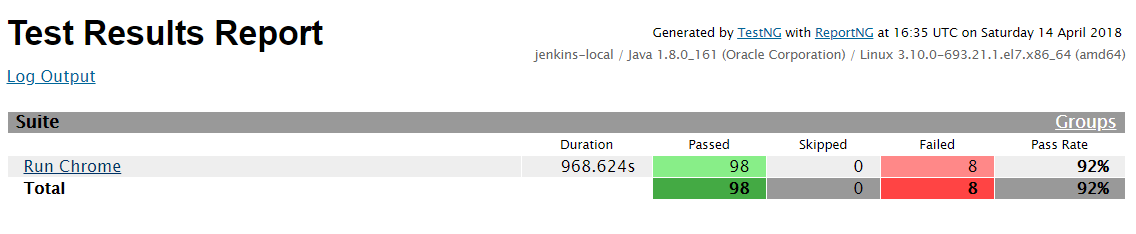
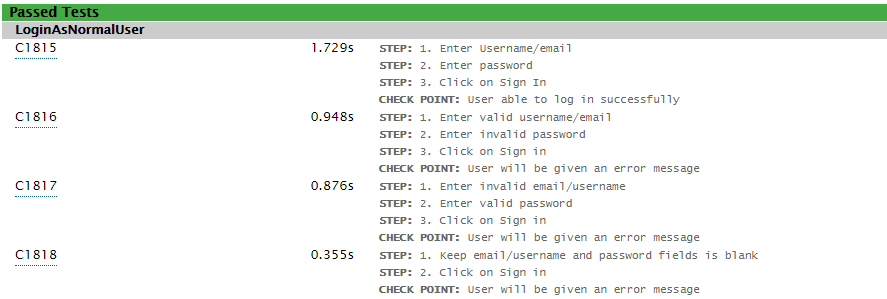
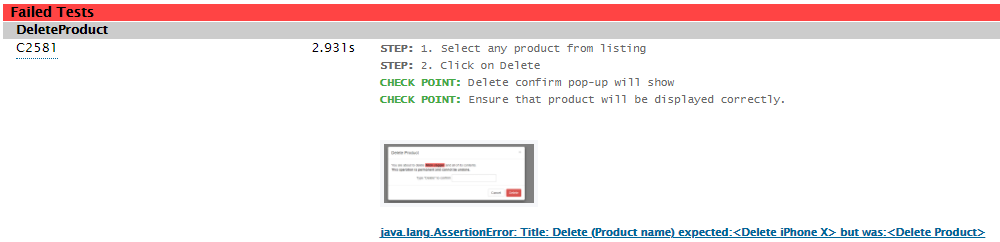
Let’s say we receive a failed test. How do we investigate the result timely enough to determine whether the failure is due to an AUT bug, an intentional design change on the AUT, or mistakes during test development and execution?

At the end of the day, test automation will be ****useless**** if we cannot get useful insights from the test results to take meaningful corrective actions. There are a lot of options available out there for logging your automated tests. Reporting mechanisms provided by testing frameworks such as Junit and TestNG are often generated in XML format, which can easily be interpreted by other software like CI/CD tools (Jenkins). Unfortunately, those XMLs are not so easy to read for us human beings.

Third party libraries such as ****ExtentReport**** and ****Allure**** can help you create test result reports that are human-readable. They also include visuals like pie charts and screenshots.

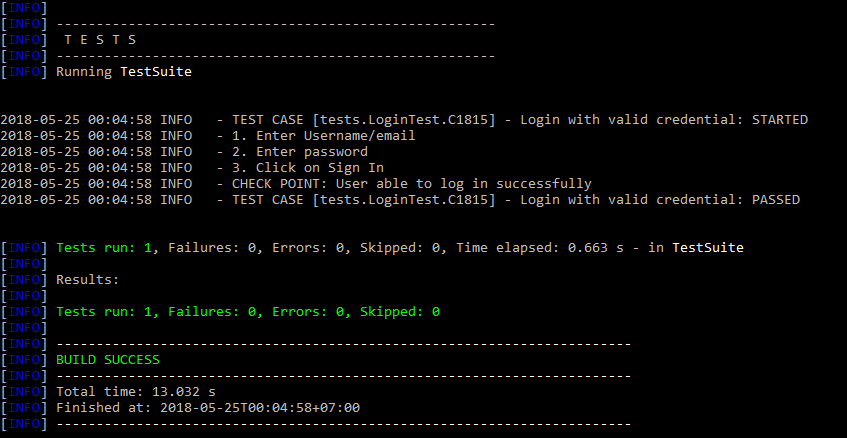
If you don’t like those tools, there is an open-source Java reporting library called ****ReportNG****. It’s a simple HTML plug-in for the ****TestNG**** unit-testing framework that provides a simple, color-coded view of the test results. The sweet spot is: setting up ReportNG is very easy.

A good report should provide detailed information such as: the amount of passed or failed test cases, pass rate, the execution time, and the reasons why test cases failed. The below pictures are example reports generated by ****ReportNG****.

*Figure 14 – Overall summary such as execution time, number of passed/failed/skipped**Figure 15 – Some passed test cases with detailed steps and check points*Figure 16 – A failed test with a screenshot showing what went wrong at the check point

## ****Decide how to implement CI/CD****

To complete your Selenium framework, there are a few other areas of concern that you might want to tackle.

* ****Build tools and dependency managers:**** Dependency managers help you manage the dependencies and libraries that the framework is using. Examples of these tools include Maven, Gradle, Ant, NPM, and NuGet. Invest in a dependency manager to avoid missing dependencies when you build your framework
* ****Build tools**** assist you in building the source code and dependent libraries, as well as in running tests. The below image illustrates how we use Maven to execute our tests (mvn clean test).  
  
* **[Version control](https://www.atlassian.com/git/tutorials/comparing-workflows" \t "https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/_blank)**:**** All Automation teams must collaborate and share source code with each other. Just like a software development project, source code of the tests and test utilities are stored in a source control system, also known as a version control system. Popular source control systems are GitHub, Bitbucket, and TFS. However, we recommend that your team set up an in-house source control system using Git if you don’t want to share your source code with the public.
* ****CI/CD integration:**** Popular CI systems include Jenkins, Bamboo, and TFS. In the world of ever-increasing demand on agility, you will soon find it useful to integrate your automated tests into DevOps pipelines so that your organization can speed up delivery and stay competitive. We’d recommend Jenkins since it’s free and very powerful.

## ****Integrate your framework with other tools****

Consider integrating with the following tools to add more value to your framework:

* ****AutoIt**** is a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting. It will help you in case you want to work with desktop GUI, like the download dialog of the browser.
* **[TestRail](http://docs.gurock.com/testrail-api2/start" \t "https://www.logigear.com/blog/test-automation/building-a-selenium-framework-from-a-to-z/_blank)** is a test case management (TCM) system that proves useful when your project has a large number of tests and related work items such as bugs and technical tasks. It’s best if our Selenium framework can automatically upload test results to TestRail after execution.
* ****Jira**** is a famous eco-system for software development and testing. Thus, consider integrating with Jira in some common scenarios such as automatically posting and closing Jira bugs according to Selenium test results.

# ****Conclusion****

Selenium is a powerful tool to perform functional and regression testing. In order to get the most benefit out of it, we should have a good framework architecture right from the start. Once you cement a strong foundation, anything you build on top of it is there to stay.

Hopefully after reading this ebook, you are now 100% ready to build a good framework architecture from scratch or upgrade your existing Selenium framework to the next level. From our 25 years of experience in Software Testing, investments in learning the best practices of designing a good framework architecture pay off exponentially in the long run. You won’t regret it.