**Introduction**

*There are lot of problem while designing the testing framework, faced by many people. Most of the common problems can be solved by using the existing design patterns mentioned in book ‘Gangs of Four’. This book contains classic design patterns were the authors has showed why, when and how to use every particular pattern in object oriented programming.*

**Need of patterns in our tests**

*Software development has lot of standardized approaches to make the development process efficient such as object oriented programming, domain-driven design, test-driven design and behavior driven design etc. Automation testing, since the very beginning, has been relatively new when it comes to processes and standards. But now it has gained lot of exposure in terms of standardization and has been under the process of continuous improvement and evolvement through design patterns. Automation testing is a process of developing software to test software. Hence, the test patterns are loosely similar to design patterns that are used in software development. Design patterns show how to design the test automation testware so that it will be efficient and easy to maintain. The most challenging part in test automation has always been the code maintenance. A lot of test automation projects have drowned or were scrapped due to the inability of the frameworks to cope up with the growing codebases. In order to keep the maintenance cost low, the automation engineers should strive to minimize the code that they reinvent or create from scratch by using existing functionality for common, generic, or repeated operations. The main reasons for leveraging design patterns in test automation is increasing of stability, maintainability, flexibility, reliability and clarity. All of them are important but different aspect can be reached by using different patterns.*

**Design Patterns**

*Following are the few design patterns we will discuss in the document:*

1. *Page Object*
2. *Page Factory*
3. *Strategy Pattern*
4. *Value Object Pattern*
5. *Builder pattern*
6. *Fluent chain of invocation*

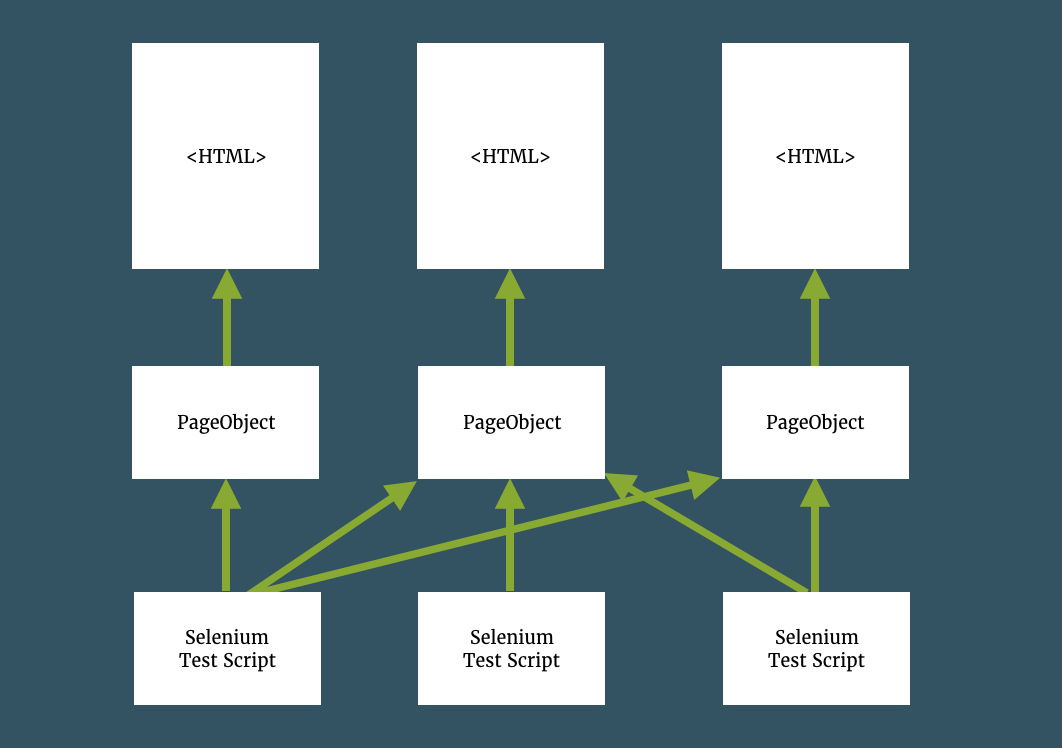
***Page Object Pattern***

*This is the most common and most popular design patterns among test engineers. The main problem it solves is the separation of technical details (e.g. user interface elements on the page/screen) and actual test logic of UI tests.*

*It also helps in code duplication, as we can reuse the page object in multiple test scripts to reuse the code.*

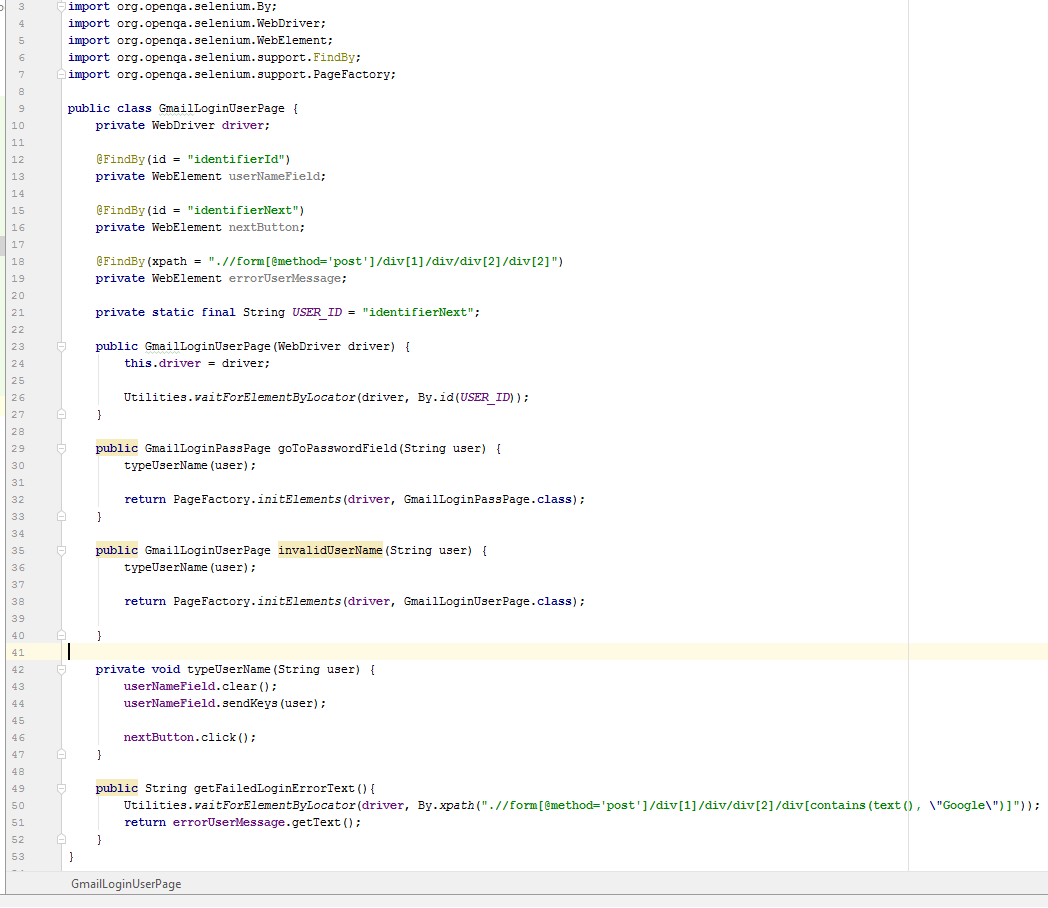
*Page object pattern makes the code more readable and understandable. It shows which page is user right now and prevents from doing actions not related to current page.*

*Page object pattern is not complete solution. If a project has a small amount of test (less than 25) and tester do not have plan to extend it, then consider not to use it as effort to create page pattern won’t pay-off.*



***Page Factory Pattern***

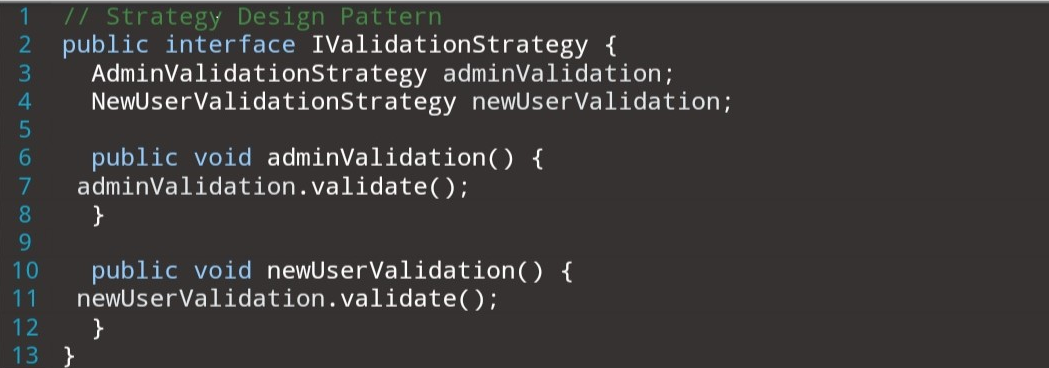
*Page factory pattern is an extension to Page Object pattern. It helps to encapsulate page’s attributes and methods even more by providing ‘@FindBy’ annotations. We can use ‘initElements’ method to initialize web elements. This static helper initializes all fields with FindBy annotations on the page, which will be found on it on each call. The main advantage is the fact that now directly with fields, buttons, windows etc. and no need to worry about the low level driver’s interactions.*



***Strategy Pattern***

*Strategy pattern is used whenever we want to have more than one implementations of the same action/sequence of actions, which is done differently. Depending on the context we could choose the implementation.*

*The easiest example is user validation. We might want to have two different implementations of this particular action. The first one would the admin user validation. The other one new user validation. In both the case validation is same only the strategy (implementation is different) in such scenarios strategy pattern is used.*

**

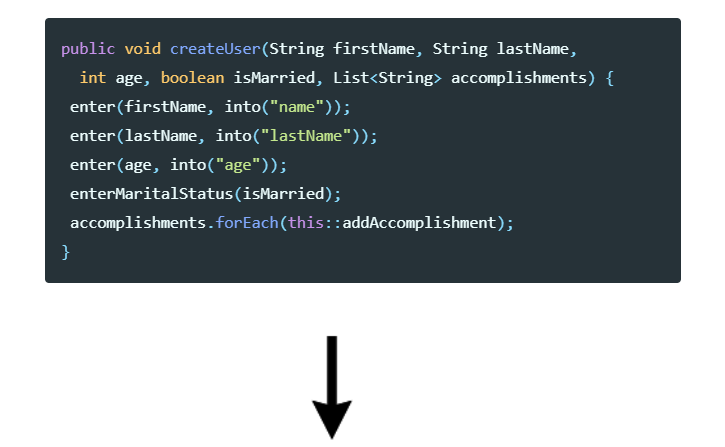
*In the above snapshot The IValidationStrategy interface is used to design two implementation of user validation, one for admin and another for new user. This way we segregate the validations process for admin and new user using strategy pattern.*

**

**

***Value Object Pattern***

*This pattern is used by the developers to customize the data injection into their unit test cases. Same can be leveraged by the tester to optimize the test data injection to the test cases. The main advantage of the value object pattern is that it is immutable (it cannot be changed once it’s created) and its main purpose is to deliver data from point A to point B avoiding side effects including modifications and extensions. Value Object can make our code more readable and significantly reduce amount of repeatable constructions.*

**

**

*If we have multiple objects which have some common logic (in the case above****createUser****accepts five parameters - first name, last name, age, marital status etc), it’s better to merge them in one entity. In this case User will be our Value Object, which aggregates all needed information about actual user into it.*

***Builder pattern***

*Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.*

*Earlier approach:*

*Consider an Address constructor for filling up the different address details based on different client requirement, the usual approach is to create constructor for each time*

// First Constructor Type: Used When there is only one Address

Address (String name, String streetAddress1, String city,

String state, String zipCode, String country) {

this.name = name;

this.streetAddress1 = streetAddress1;

this.city = city;

this.state = state;

this.zipCode = zipCode;

this.country = country;

}

// Second Constructor Type: Used When there is Address1 and Address2.

Address ( String name, String streetAddress1, String streetAddress2,

String city, String state, String zipCode, String country){

this.name = name;

this.streetAddress1 = streetAddress1;

this.streetAddress2 = streetAddress2;

this.city = city;

this.state = state;

this.zipCode = zipCode;

this.country = country;

}

*The builder pattern as defined in 2008 edition of book* ***Effective Java*** *by Joshua Block is*

*"Traditionally, programmers have used the telescoping constructor pattern, in which we provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters [...]   
  
"[...T]he telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it. The reader is left wondering what all those values mean and must carefully count parameters to find out. Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won’t complain, but the program will misbehave at runtime.  
  
"Luckily, there is a third alternative that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern. It is a form of the Builder pattern*

*Instead of making the desired object directly, the client calls a constructor (or static factory) with all of the required parameters and gets a builder object. Then the client calls setter-like methods on the builder object to set each optional parameter of interest. Finally, the client calls a parameter less build method to generate the object, which is immutable".*

*Above code can be converted to builder pattern as follows:*

public class Address {

private final String addressName;

private final String address1;

private final String address2;

private final String city;

private final String state;

private final String zip;

private final String country;

public static class Builder {

// Required Parameters

private final String name;

private final String address1;

private final String city;

private final String state;

// Optional Parameters

private String address2;

private String zip;

private String country;

}

public Builder (String name, String address1,

String city, String state) {

this.name = name;

this.address1 = address1;

this.city = city;

this.state = state;

}

public Builder setAddress2(String value) {

address2 = value;

return this;

}

public Builder setCountry(String value) {

country = value;

return this;

}

public Builder setZip(String value) {

zip = value;

return this;

}

public Address build() {

return new Address(this);

}

}

private Address(Builder builder) {

this.addressName = builder.name;

this.address1 = builder.address1;

this.city = builder.city;

this.state = builder.state;

this.address2 = builder.address2;

this.country = builder.country;

this.zip = builder.zip;

}

*The builder pattern can be called in the client program as :*

Address **fitbitBostonAddress** = new Address.Builder("Fitbit-Boston","One Marina Park Drive", "Boston", "MA").build();

Address **fitbitBostonMailingAddress** = new Address.Builder("Fitbit-Boston",

"One Marina Park Drive", "Boston", "MA").setAddress2("Suite 701").setZip("02210").build();

Address **fitbitBostonMailingAddressWithCountry** = new Address.Builder("Fitbit-Boston”, “One Marina Park Drive", "Boston", "MA").setAddress2("Suite 701").setZip("02210").setCountry("United States").build();

***Fluent chain of invocation***

*The next design pattern is fluent chain of invocation pattern, mostly used along with Page Object pattern. The problem it resolves is helping test developer to determine whether they can use the object or should switch to other one. For example, user is on the login page and pressing some button. By writing code in old fashion user wouldn’t be sure if he/she’s still on the login page or already on the home page.*

*And now just imagine if user had 50 similar methods in their test. He/she cannot be sure if he can invoke them right away, or they depend on some order, or even they cannot be invoked after one of them was executed.*

***Old fashion approach:***



***Chain of invocation approach:***

**References**

1. [*https://en.wikipedia.org/wiki/Design\_Patterns*](https://en.wikipedia.org/wiki/Design_Patterns)
2. *Book: Gang of four Design Patterns by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm*
3. *Book: Effective Java edition 2008*