Use of Java Design Patterns while developing Automation Framework using Selenium

Agenda for this document: To design frameworks for Selenium by using Java Design Patterns.

Why use Design Patterns in Selenium Framework:
As selenium is an open source tool, so Its does not provide a framework along with it unlike other paid tool. So whoever is using Selenium has to write a framework for it.
Now, If you design your framework by using Java Design Patterns It can have multiple strengths:

1.Well **structured** code base
2.It will be **generic** and **robust**.
3. Framework **re-usability** and **maintainability** will be very high
4. **Enhancements** can be done with very **little code changes without disturbing the existing Flows**.
5. **Debugging** of Scripts/framework (if any) will be very **easier**.
6. **Framework can be used** with ease by the people **who are new to Automation**.

What Type of Design Patterns will be useful for Selenium Framework:
First of all it depends on the requirement for the framework as it varies from Company to company and sometimes project to project, but then also there are some generic scenarios where we can use design patterns like **Factory Design Pattern, Abstract Design Pattern, Singleton Design Pattern** in the framework.

Which Area of a Generic Framework can be developed using Java Design Patterns:

Scenarios:

1. We are having a requirement to use Local WebDriver, Grid, ThreadDriver, HeadLess Driver, EventDriver, RemoteDriver with Sauce Labs Implementation etc.

so there we can use **Factory Design Pattern** and along with the above feature if you need multiple browsers like FireFox, Chrome, Safari, IE etc etc. then we can use **Abstract Factory Design Pattern**.

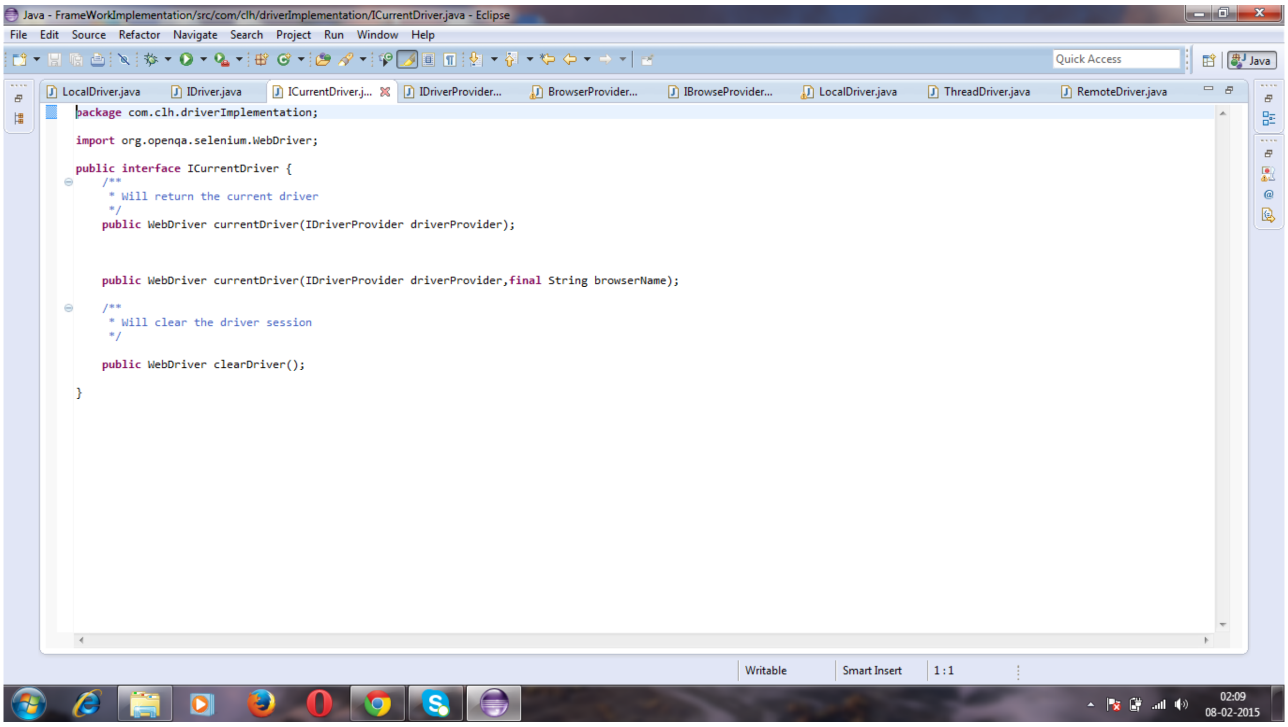
2. If you want to support Data Handling using xls file ,properties file, csv files etc etc there we can use the Factory Design Pattern.

3. We can configure the configuration Class using **Singleton Design Pattern**.

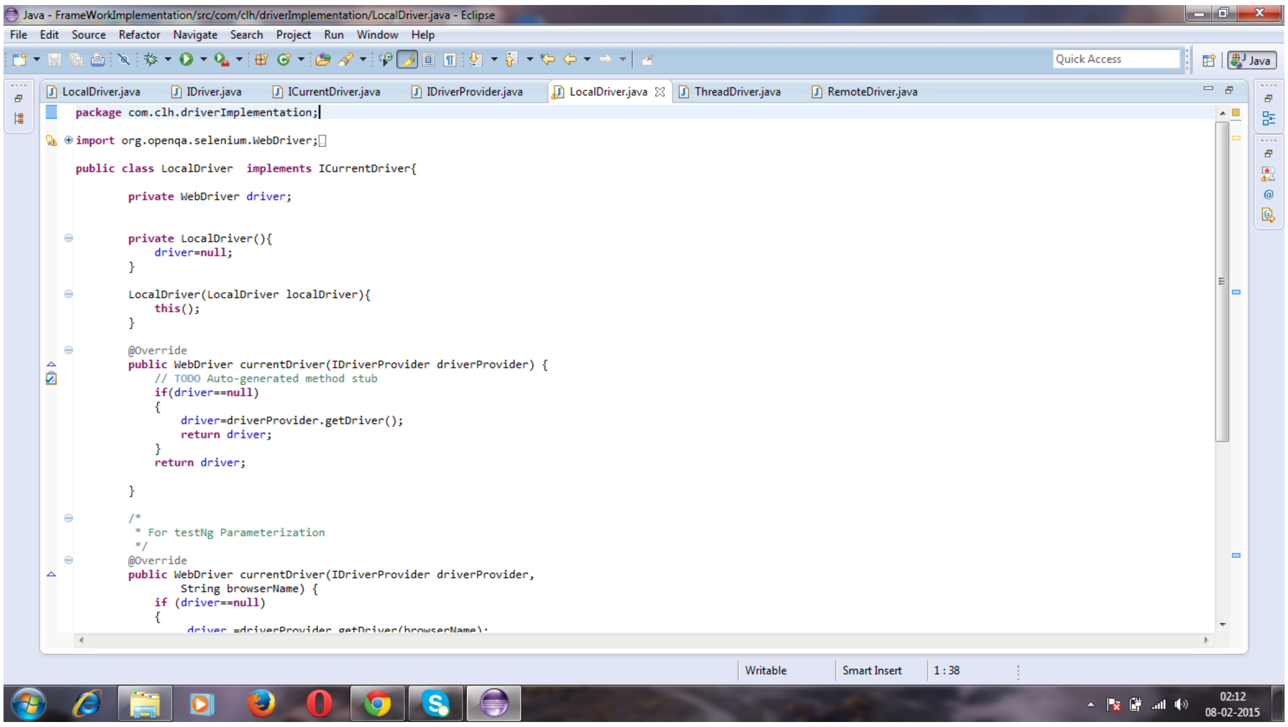
HOW:

Find the example for different driver implementation using **Factory** Design Pattern:

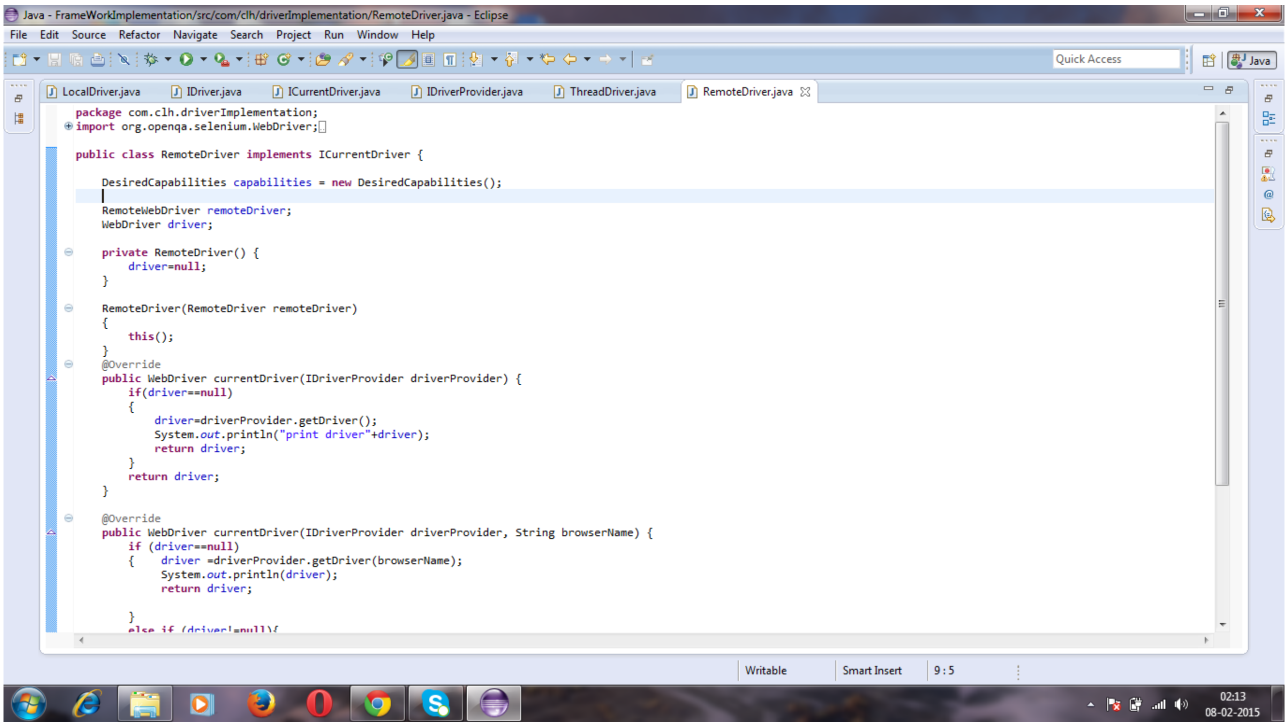
We are Creating an Interface ICurrentDriver which is Implemented by all type of Driver Classes such as ThreadDriver, LocalDriver, RemoteDriver because the implementations are different for each of them.



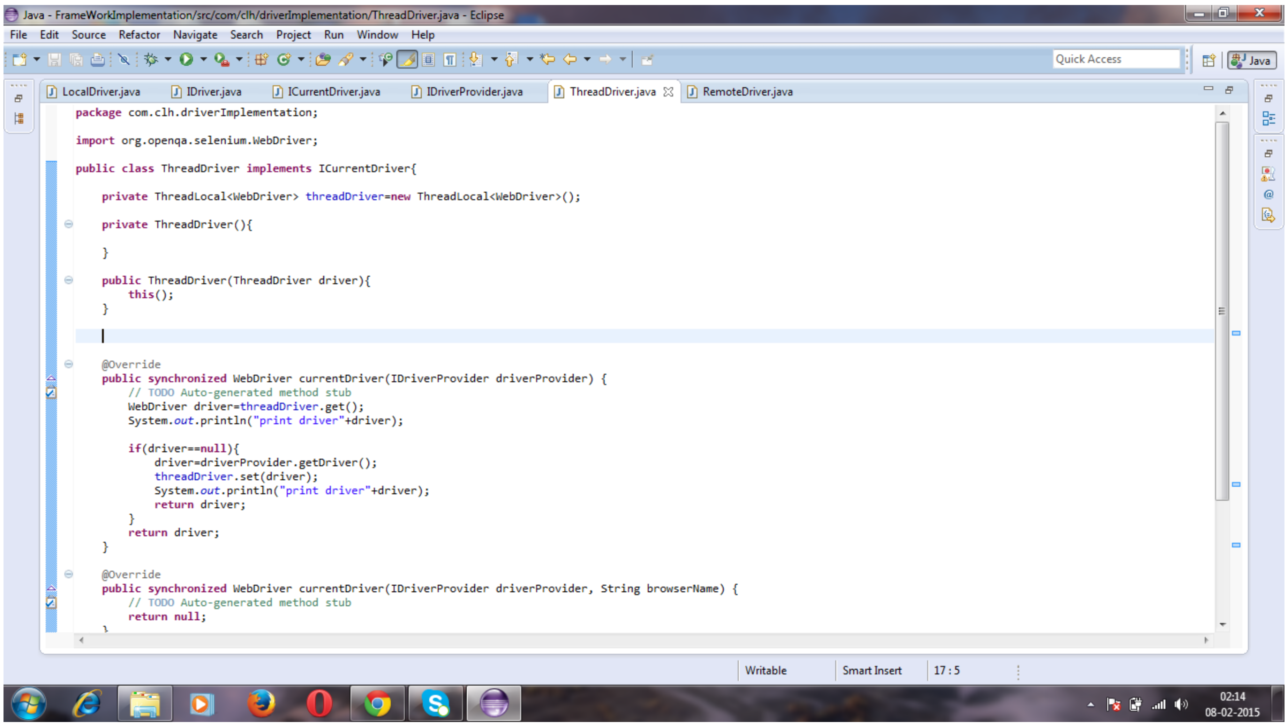
```java
package com.clh.driverImplementation;

import org.openqa.selenium.WebDriver;

public interface ICurrentDriver {
    /**
     * Will return the current driver
     */
    public WebDriver currentDriver(IDriverProvider driverProvider);


    public WebDriver currentDriver(IDriverProvider driverProvider,final String browserName);

    /**
     * Will clear the driver session
     */

    public WebDriver clearDriver();

}
```



```java
package com.clh.driverImplementation;

import org.openqa.selenium.WebDriver;

public class LocalDriver  implements ICurrentDriver{

        private WebDriver driver;


        private LocalDriver(){
            driver=null;
        }

        LocalDriver(LocalDriver localDriver){
            this();
        }

        @Override
        public WebDriver currentDriver(IDriverProvider driverProvider) {
            // TODO Auto-generated method stub
            if(driver==null)
            {
                driver=driverProvider.getDriver();
                return driver;
            }
            return driver;

        }

        /*
         * For testNg Parameterization
         */
        @Override
        public WebDriver currentDriver(IDriverProvider driverProvider,
                String browserName) {
            if (driver==null)
            {
                 driver =driverProvider.getDriver(browserName);
```

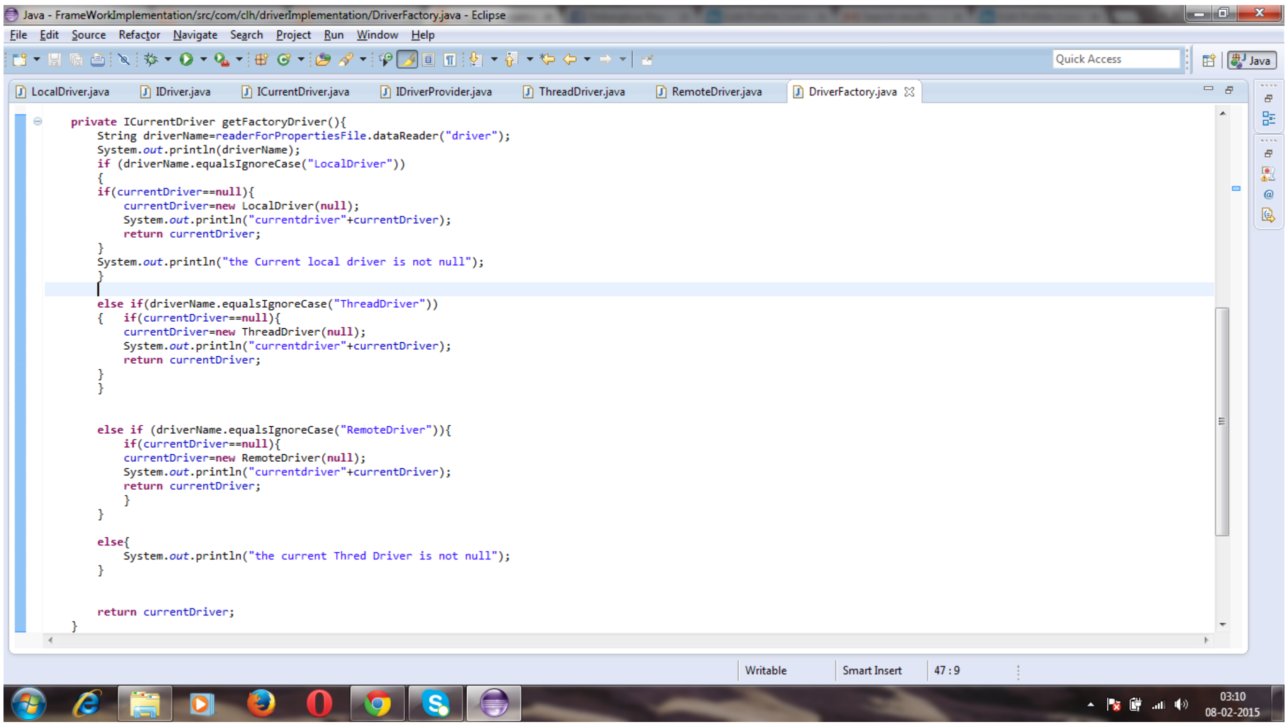


```java
package com.clh.driverImplementation;
import org.openqa.selenium.WebDriver;

public class RemoteDriver implements ICurrentDriver {

    DesiredCapabilities capabilities = new DesiredCapabilities();

    RemoteWebDriver remoteDriver;
    WebDriver driver;

    private RemoteDriver() {
        driver=null;
    }

    RemoteDriver(RemoteDriver remoteDriver)
    {
        this();
    }
    @Override
    public WebDriver currentDriver(IDriverProvider driverProvider) {
        if(driver==null)
        {
            driver=driverProvider.getDriver();
            System.out.println("print driver"+driver);
            return driver;
        }
        return driver;
    }

    @Override
    public WebDriver currentDriver(IDriverProvider driverProvider, String browserName) {
        if (driver==null)
        {    driver =driverProvider.getDriver(browserName);
             System.out.println(driver);
             return driver;

        }
        else if (driver!=null){
```



```java
package com.clh.driverImplementation;

import org.openqa.selenium.WebDriver;

public class ThreadDriver implements ICurrentDriver{

    private ThreadLocal<WebDriver> threadDriver=new ThreadLocal<WebDriver>();

    private ThreadDriver(){

    }

    public ThreadDriver(ThreadDriver driver){
        this();
    }



    @Override
    public synchronized WebDriver currentDriver(IDriverProvider driverProvider) {
        // TODO Auto-generated method stub
        WebDriver driver=threadDriver.get();
        System.out.println("print driver"+driver);

        if(driver==null){
            driver=driverProvider.getDriver();
            threadDriver.set(driver);
            System.out.println("print driver"+driver);
            return driver;
        }
        return driver;
    }

    @Override
    public synchronized WebDriver currentDriver(IDriverProvider driverProvider, String browserName) {
        // TODO Auto-generated method stub
        return null;
    }
```

We can Design another Class Factory which will provide us the Object for the required Driver classes at runtime depending on the need.So we don't need to create and use separate Objects for each of the Driver classes,we will use only Factory class object and can access any of the Driver classes members and fields.



```java
    private ICurrentDriver getFactoryDriver(){
        String driverName=readerForPropertiesFile.dataReader("driver");
        System.out.println(driverName);
        if (driverName.equalsIgnoreCase("LocalDriver"))
        {
        if(currentDriver==null){
            currentDriver=new LocalDriver(null);
            System.out.println("currentdriver"+currentDriver);
            return currentDriver;
        }
        System.out.println("the Current local driver is not null");
        }

        else if(driverName.equalsIgnoreCase("ThreadDriver"))
        {   if(currentDriver==null){
            currentDriver=new ThreadDriver(null);
            System.out.println("currentdriver"+currentDriver);
            return currentDriver;
        }
        }


        else if (driverName.equalsIgnoreCase("RemoteDriver")){
            if(currentDriver==null){
            currentDriver=new RemoteDriver(null);
            System.out.println("currentdriver"+currentDriver);
            return currentDriver;
            }
        }

        else{
            System.out.println("the current Thred Driver is not null");
        }


        return currentDriver;
    }
```

By this way we can use Design Patterns as per the requirement, we can use this concept in case of Handling Data in Selenium Framework.