NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

D

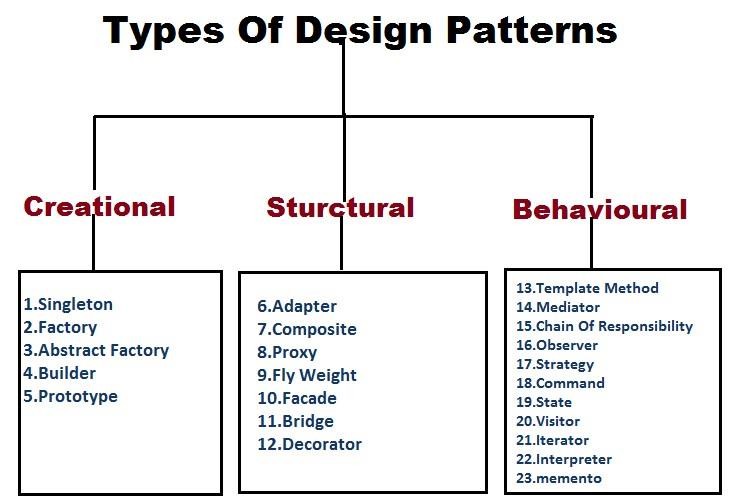
Asfc

SL-2002 – Software Design & Architecture Lab

Lab Instructor: Rubab Manzar “rubab.jaffar[@nu.edu.pk”](mailto:a@nu.edu.pk)

Lab 12

* Singleton pattern
* Faced Pattern
* Factory pattern
* Adapter pattern
* Example
* Exercise



Design patterns in java are best practices which are used to resolve some known issues. Design patterns can be divided into 3 different types. Here we have listed down some of the widely used design patterns in Java.

***Singleton Design Pattern***

***Intent***

• Ensure a class has only one instance, and provide a global point of access to it.

• Encapsulated "just-in-time initialization" or "initialization on first use".

***Problem***

• Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Initialization Types:

1. Lazy Initialization

2. Early Initialization

**Example-01: Early Initialization**

This is the simplest method of creating a singleton class. In this, object of class is created when it is loaded to the memory by JVM. It is done by assigning the reference an instance directly.

It can be used when program will always use instance of this class, or the cost of creating the instance is not too large in terms of resources and time.

**public class** SingletonExample {

**private static** SingletonExample *instance* = **new** SingletonExample();

**private** SingletonExample(){}

**public static** SingletonExample getInstance() {

**return** *instance*;

}

**public static void** main(String[] args) {

// **TODO** Auto-generated method stub

SingletonExample instance = SingletonExample.*getInstance*();

System.***out***.println(instance);

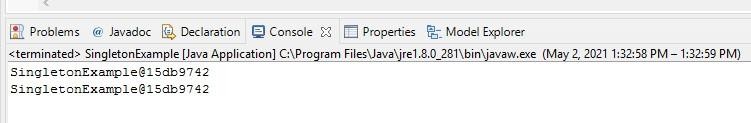
SingletonExample instance1 = SingletonExample.*getInstance*();

System.***out***.println(instance1);

}

}

Output:



**Example-02: Lazy Initialization**

In this method, object is created only if it is needed. This may prevent resource wastage. An implementation of getInstance() method is required which return the instance. There is a null check that if object is not created then create, otherwise return previously created. To make sure that class cannot be instantiated in any other way, constructor is made final. As object is created with in a method, it ensures that object will not be created until and unless it is required.

Instance is kept private so that no one can access it directly.

It can be used in a single threaded environment because multiple threads can break singleton property because they can access get instance method simultaneously and create multiple objects.

//Java Code to create singleton class

// With Lazy initialization

**public cl**ass GFG

{

// private instance, so that it can be

// accessed by only by getInstance() method

**private static** GFG *instance*;

**private** GFG()

{

// private constructor

}

//method to return instance of class

**public static** GFG getInstance()

{

**if** (*instance* == **null**)

{

// if instance is null, initialize

*instance* = **new** GFG();

}

**return** *instance*;

}

}

**Example-03**

Following code create two separate instance for same methods. This is not technically right to run a same method with two different object. To reduce a memory overhead we have to work with one instance to call same method at different time. Using Singleton.

The Singleton's purpose is to control object creation, limiting the number to one but allowing the flexibility to create more objects if the situation changes. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields.

**public class** Singleton {

//

**private static int** *counter*=0;

**private** Singleton(){

*counter*++;

System.***out***.println("counter value is "+*counter*);

}

**public static void** getInstance(String Message) {

System.***out***.println(Message);

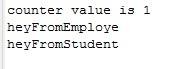
}

**public static void** main(String[] args) {

// **TODO** Auto-generated method stub

Singleton instance = **new** Singleton ();

instance.*getInstance*("heyFromEmploye");



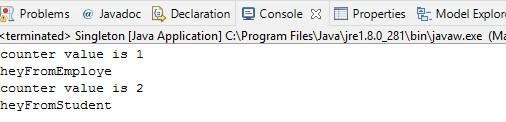
Singleton instance1= **new** Singleton ();

instance1.*getInstance*("heyFromStudent");

}

}

Here you can see two separate counter value occurred because at the backhand instance are created more then one time. And this is not right to create more then one instance.



Following code are singleton based solution, that restrict to create more than one object. You can see counter value is one due to one instance using lazy Initialization.

**public class** Singleton {

//

**private static int** *counter*=0;

**private static** Singleton *instance*=**null** ;

**public static** Singleton getInstance() {

**if**(*instance* == **null**) {

*instance* = **new** Singleton(); }

**return** *instance*; }

**private** Singleton(){

*counter*++;

System.***out***.println("counter value is "+*counter*);

}

**public static void** getInstance(String Message) { System.***out***.println(Message);

}

**public static void** main(String[] args) {

// **TODO** Auto-generated method stub

Singleton instance = Singleton.*getInstance*();

instance.*getInstance*("heyFromEmploye");

Singleton instance1 = Singleton.*getInstance*();

instance1.*getInstance*("heyFromStudent");

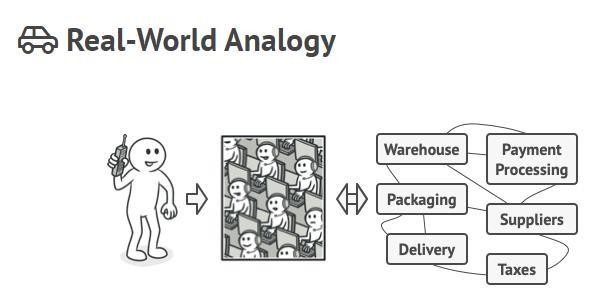
}}

Result

***Facade Design Pattern***

◦ Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

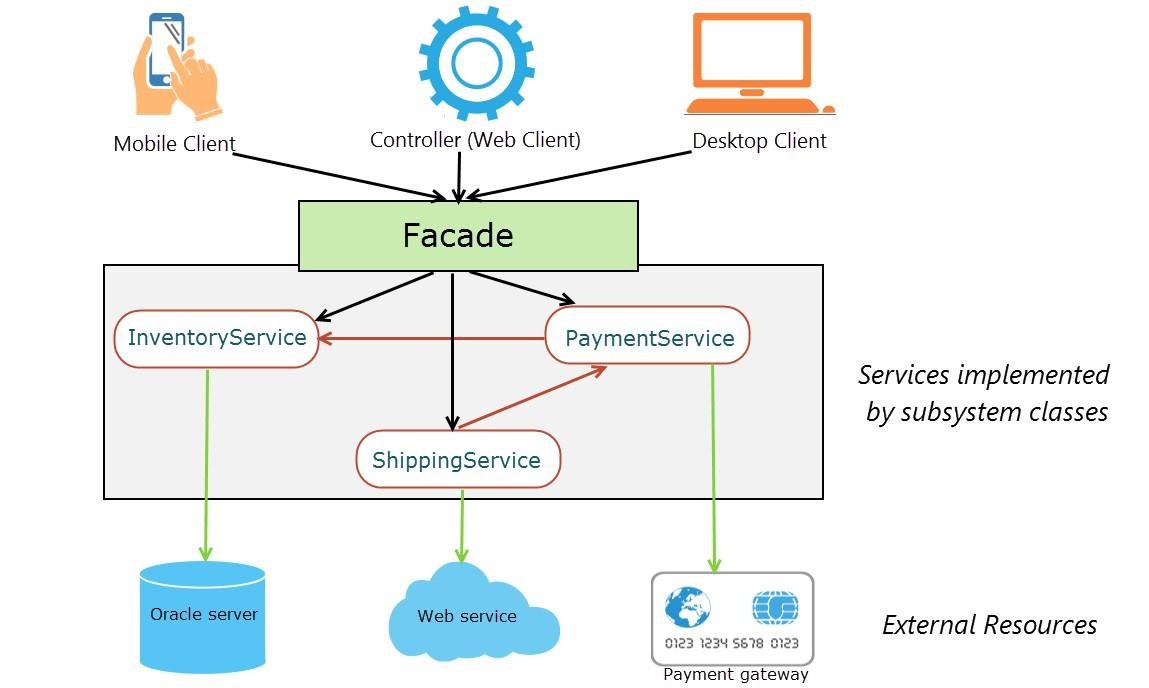
◦ Wrap a complicated subsystem with a simpler interface.



**Real world facade examples**

▪ To understand the facade, let’s take a very simple example of a desktop computer machine. When we have to start a computer, all we have to do is press the start button. We really do not care what all things go inside the computer hardware and software. It is an example of Facade pattern.

▪ In Java programming, we must have connected to a database to fetch some data. We simply call the method dataSource.getConnection() to get the connection but internally a lot of things happen such as loading the driver, creating connection or fetching connection from pool, update stats and then return the connection reference to caller method. It is another example of Facade pattern in the programming world.



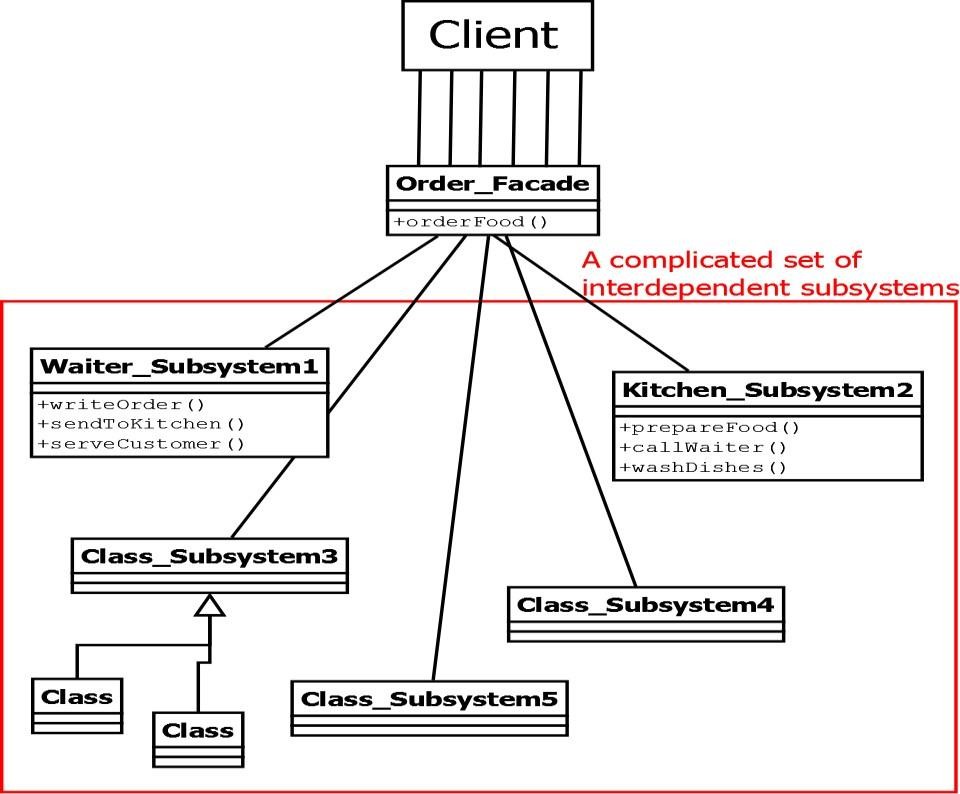
***Example-01-Food Order***

1. **Client:** The client in this example is the customer of a restaurant that wants to order food.

2. **Facade:** Its job is to be able to provide to the client more simplified access towards numerous interdependent subsystems that are considered complicated. In this example, a client’s food-order would require a series of carefully sequenced method calls of two different subsystems (Waiter and Kitchen).

3. **Subsystems:** The subsystems are hidden from the client. They might also be not accessible to the client. The client cannot fiddle with any of the subsystems where a simple code-change may prove to be fatal or even break other unknown parts of the system itself. In this scenario, the waiter and the kitchen have to do a series of tasks. A subsystem’ s task is sometimes dependent on the another’s task.

4. For example, the kitchen cannot prepare the food if the waiter doesn’t bring the order to the kitchen. The waiter cannot serve the customer if the food is not cooked.



***Code***

***Waiter\_Subsystem.java***

**public class** Waiter\_Subsystem1 {

**public void** writeOrder() {

System.out.println(" Waiter writes client's order\n");

}

**public void** sendToKitchen(){

System.out.println(" Send order to kitchen\n");

}

**public void** serveCustomer(){

System.out.println(" Yeeei customer is served!!!\n");

}}

***Kitchen\_Subsystem2.java***

**public class** Kitchen\_Subsystem2 {

**public void** prepareFood(){ System.***out***.println(" Cook food\n");} **public void** callWaiter() { System.***out***.println(" Call Waiter\n");} **public void** washDishes() { System.***out***.println(" Wash the dishes\n");}

}

***Order\_Facade.java***

**public class** Order\_Facade {

Waiter\_Subsystem1 waiter=**new** Waiter\_Subsystem1(); Kitchen\_Subsystem2 kitchen=**new** Kitchen\_Subsystem2();

**public**

**void** orderFood()

{

subsystems:\n");

System.***out***.println("A series of interdependent calls on various

waiter.writeOrder(); waiter.sendToKitchen(); kitchen.prepareFood(); kitchen.callWaiter(); waiter.serveCustomer(); kitchen.washDishes();

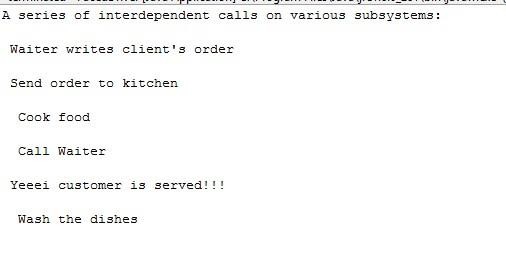
}

}

***FacedDriver.java***

|  |  |  |
| --- | --- | --- |
| **public class** | FacedDriver | { |
| **public static void** main(String[] args) {  // **TODO** Auto-generated method stub  Order\_Facade facade = **new** Order\_Facade();  facade.orderFood();  }} | | |

***Result***



***When Should this pattern be used?***

The facade pattern is appropriate when you have a **complex system** that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

**Pattern Name**

Factory Method Design Pattern

In Factory pattern, we create object without exposing the creation logic to the client and refer to

newly created object using a common interface.

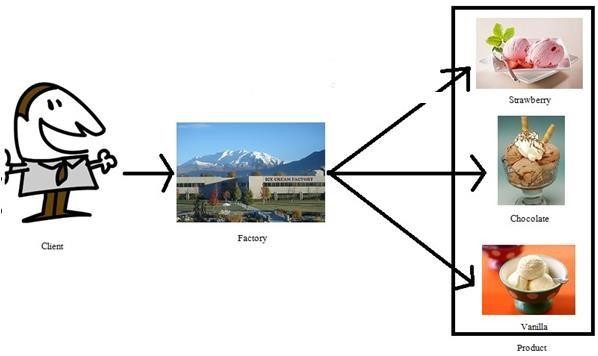
**Intent**

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.

Factory Method is similar to Abstract Factory but without the emphasis on families.

**Real time example**



**Problem**

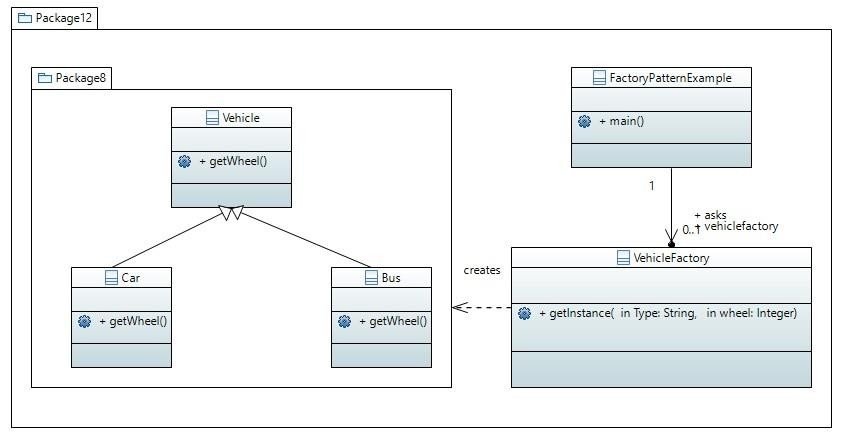
Managing helper classes and allow to define their own domain objects and provide for their instantiation.

**Applicability**

Concreate class can’t anticipate the class object it must create. Concreate class wants its subclass to specify the class object.

**Example:**

In this example Vehicle is the interface containing function getWheel to get the number of wheels for both Car and Bus. Car and Bus classes are implementing it. Vehicle Factory is the factory class using getInstance() to pass the type of vehicle and FactoryPatternExample is the client class.



**FactoryPatternExample.java**

**abstract class** Vehicle {

**public abstract int** getWheel();

**public** String toString() {

**return** "Wheel: " + **this**.getWheel();

}

}

**class** Car **extends** Vehicle {

**int** wheel;

Car(**int** wheel) {

**this**.wheel = wheel;

}

[@Override](mailto:@Override)

**public int** getWheel() {

**return this**.wheel;

}

}

**class** Bike **extends** Vehicle {

**int** wheel;

Bike(**int** wheel) {

**this**.wheel = wheel;

}

[@Override](mailto:@Override)

**public int** getWheel() {

**return this**.wheel;

}

}

**class** VehicleFactory {

**public static** Vehicle getInstance(String type, **int** wheel) {

**if**(type == "car") {

**return new** Car(wheel);

} **else if**(type == "bike") {

**return new** Bike(wheel);

}

**return null**;

}

}

**public class** FactoryPatternExample {

**public static void** main(String[] args) {

Vehicle car = VehicleFactory.*getInstance*("car", 4); System.***out***.println(car);

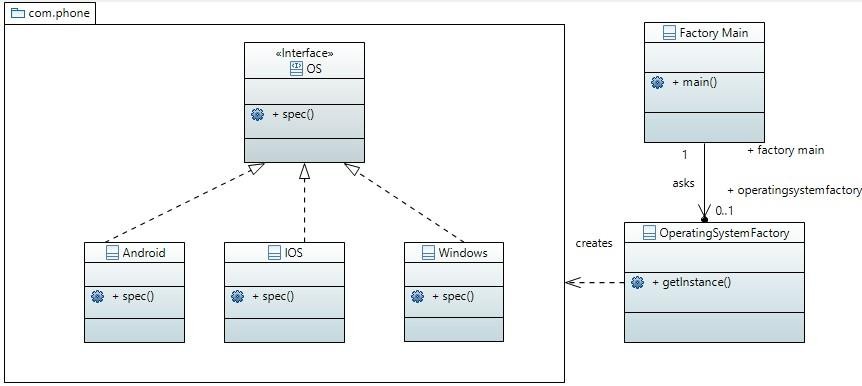
Vehicle bike = VehicleFactory.*getInstance*("bike", 2); System.***out***.println(bike);

}

}

**Example**

All the mobile phones have an Operating System (OS), every phone has it’s own OS like Samsung phones have Android OS, Iphone uses IOS and Nokia uses Windows. User does not know about the technical details operating system working behind the phone. User To implement it we can create an OS interface with the implementation of Android, IOS and Windows. Now how to know what OS are you using, it can be done based on the requirements



**OS.java**

**package** com.phone;

**public interface** OS {

**void** spec();

}

**Android.java**

**package** com.phone;

**public class** Android **implements** OS {

[@Override](mailto:@Override)

**public void** spec() {

System.***out***.println("Most powerful OS");

}

}

**IOS.java**

**package** com.phone;

**public class** IOS **implements** OS{

[@Override](mailto:@Override)

**public void** spec() {

System.***out***.println("Most secure OS");

}

}

**Windows.java**

**package** com.phone;

**public class** Windows **implements** OS {

[@Override](mailto:@Override)

**public void** spec() {

System.***out***.println("Rarely used OS");

}

}

**OperatingSystemFactory.java**

**import** com.phone.Android; **import** com.phone.IOS; **import** com.phone.OS; **import** com.phone.Windows;

**public class** OperatingSystemFactory {

**public** OS getInstance(String str)

{

**if** (str.equals("Open")) {

**return new** Android();

}

**else if** (str.equals("Closed")) {

**return new** IOS();

}

**else**

}

}

**return new** Windows();

**FactoryMain.java**

**import**  com.phone.Android; **import** com.phone.OS; **import**  com.phone.Windows;

**public class** FactoryMain {

**public static void** main(String[] args) {

OperatingSystemFactory osf= **new** OperatingSystemFactory(); OS obj = osf.getInstance("Open");

obj.spec();

}

}

**Pattern Name**

Adapter Design Pattern

Adapter pattern helps in making two incompatible interfaces to work together. It is basically a bridge between two incompatible interfaces. We need to note the fact that here the interfaces

may be incompatible but the inner functionality suits the requirement.

**Intent**

The adapter design pattern allows two incompatible classes to interact with each other by converting interface of one class into an interface expected by the client.

Adapter pattern works as a bridge between two incompatible interfaces.

**Example**

Real-life example: Memory Card reader acts like an adapter between a laptop and a memory card.



Adapter

**Problem**

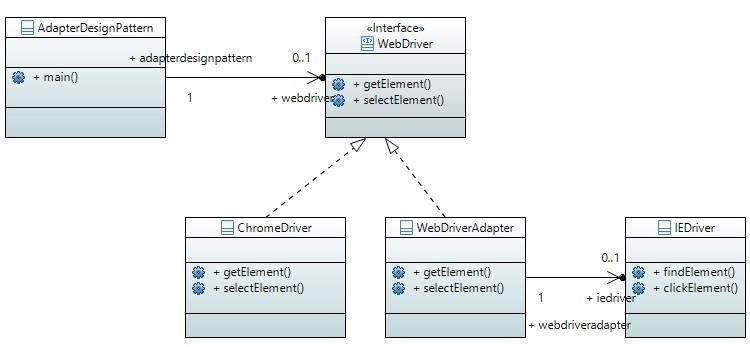
Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

**Applicability**

When we want two incompatible classes to interact with each other

**Example**

Consider the example of web driver executer, there is a web driver interface having getElement and selectElement methods, this inteface assumes that all Web drivers will support this getElement and selectElement. We have ChromeDriver that supports getElement and setElement, but let’s suppose we have another object which say that I can do that thing but my interface is different, and we have IEDriver (Internet Explore) that can perform the same function but it has findElement instead of getElement and clickElement instead of selectElement so interface is different, for that we can use Web Driver adapter, it will adapt the IE driver and it will execute the functions of IE driver, IE driver is adaptee here.



**WebDriver.java**

**interface** WebDriver {

**public void** getElement();

**public void** selectElement();}

**ChromeDriver.java**

**class** ChromeDriver **implements** WebDriver {

[@Override](mailto:@Override)

**public void** getElement() {

System.***out***.println("Get element from ChromeDriver");

}

[@Override](mailto:@Override)

**public void** selectElement() {

System.***out***.println("Select element from ChromeDriver");

}

}

**IEDriver.java**

**class** IEDriver {

**public void** findElement() {

System.***out***.println("Find element from IEDriver");

}

**public void** clickElement() { System.***out***.println("Click element from IEDriver");

}

}

**WebDriverAdapter.java**

**class** WebDriverAdapter **implements** WebDriver { IEDriver ieDriver;

**public** WebDriverAdapter(IEDriver ieDriver) {

**this**.ieDriver = ieDriver;

}

[@Override](mailto:@Override)

**public void** getElement() {

ieDriver.findElement();

}

[@Override](mailto:@Override)

**public void** selectElement() {

ieDriver.clickElement();

}

}

**AdapterDesignPattern.java**

**public class** AdapterDesignPattern {

**public static void** main(String[] args) { ChromeDriver a = **new** ChromeDriver(); a.getElement();

a.selectElement();

IEDriver e = **new** IEDriver(); e.findElement(); e.clickElement();

WebDriver wID = **new** WebDriverAdapter(e);

wID.getElement();

wID.selectElement();

}

}