NATIONAL UNIVERSITY OF COMPUTER AND EMERGINGSCIENCES

CS3005 – Software Design & Architecture Lab

LAB Instructors: Sobia Iftikhar “Sobia.iftikhar@nu.edu.pk”

Lab 13

Objectives

- Factory pattern
- Adapter pattern
- Template pattern
- Examples
- Exercise

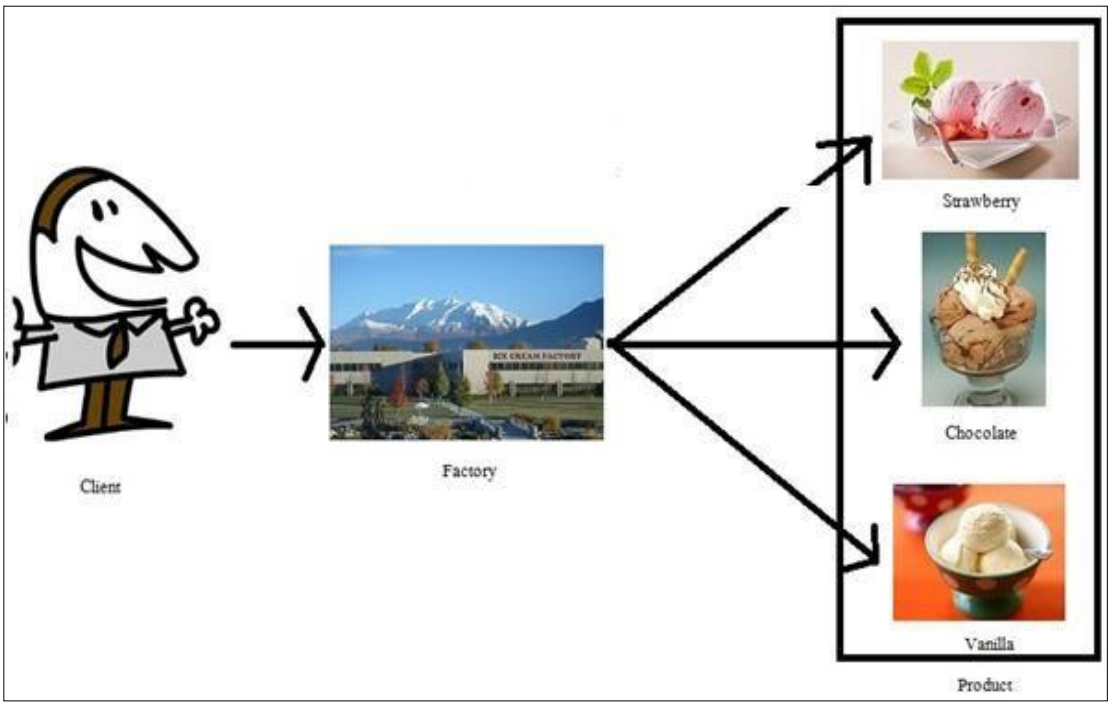
Pattern Name

- 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

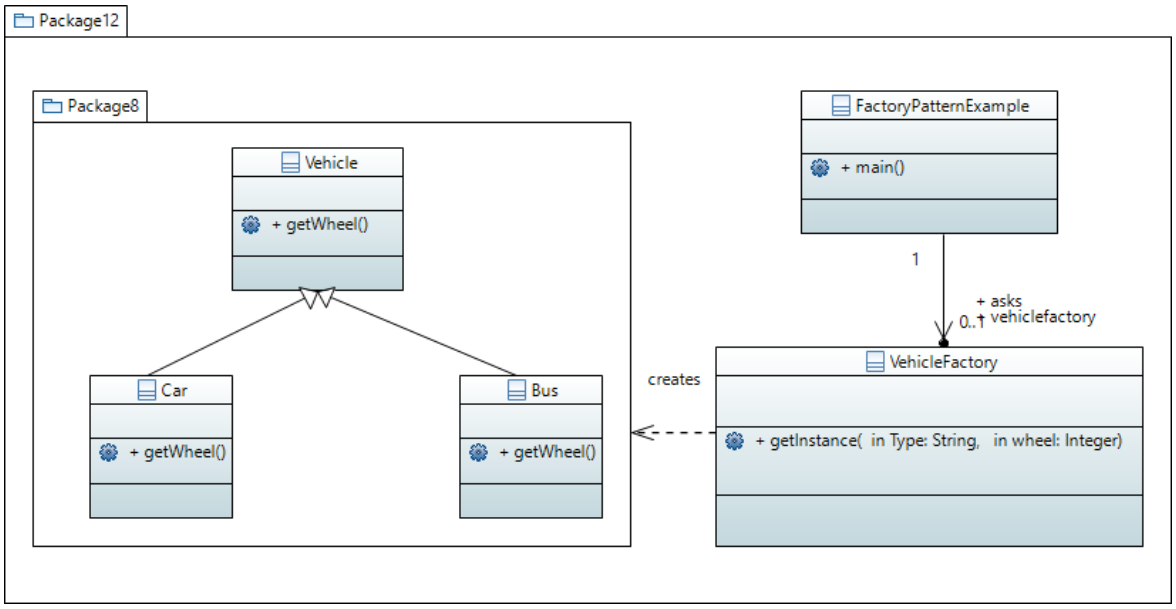
- Managing helper classes and allow to define their own domain objects and provide for their instantiation.

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.

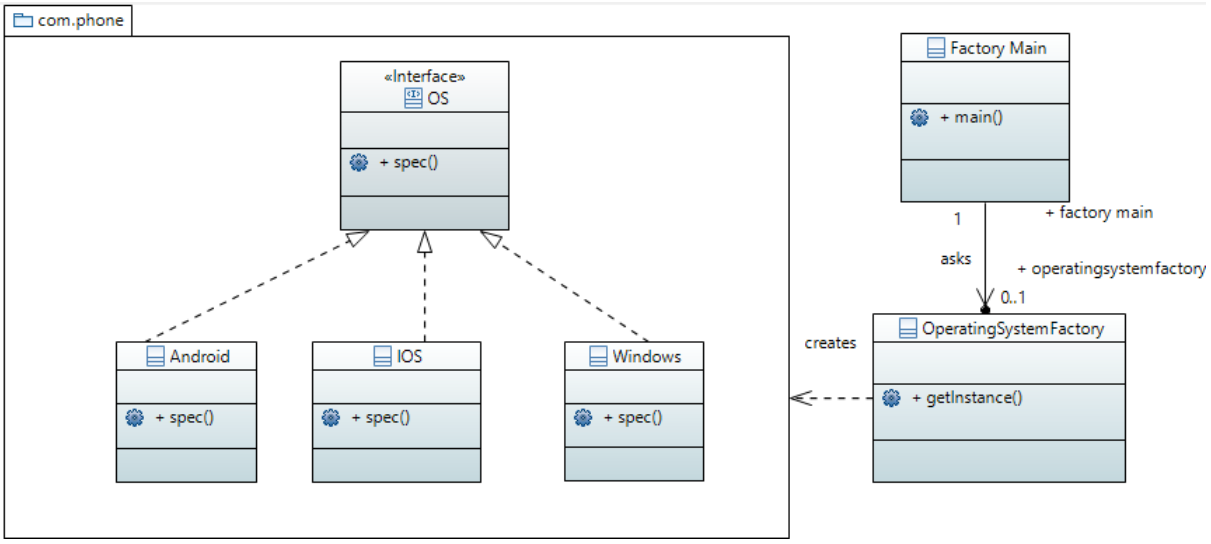


FactoryPatternExample.java

```
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
   public int getWheel() {
      return this.wheel;
   }
}

class Bike extends Vehicle {
   int wheel;

   Bike(int wheel) {
      this.wheel = wheel;
   }

   @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
	public void spec() {

		System.out.println("Most powerful OS");
	}
}
```

IOS.java

```
package com.phone;

public class IOS implements OS{

	@Override
	public void spec() {
		System.out.println("Most secure OS");

	}

}
```

Windows.java

```
package com.phone;

public class Windows implements OS {

	@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.

Problem

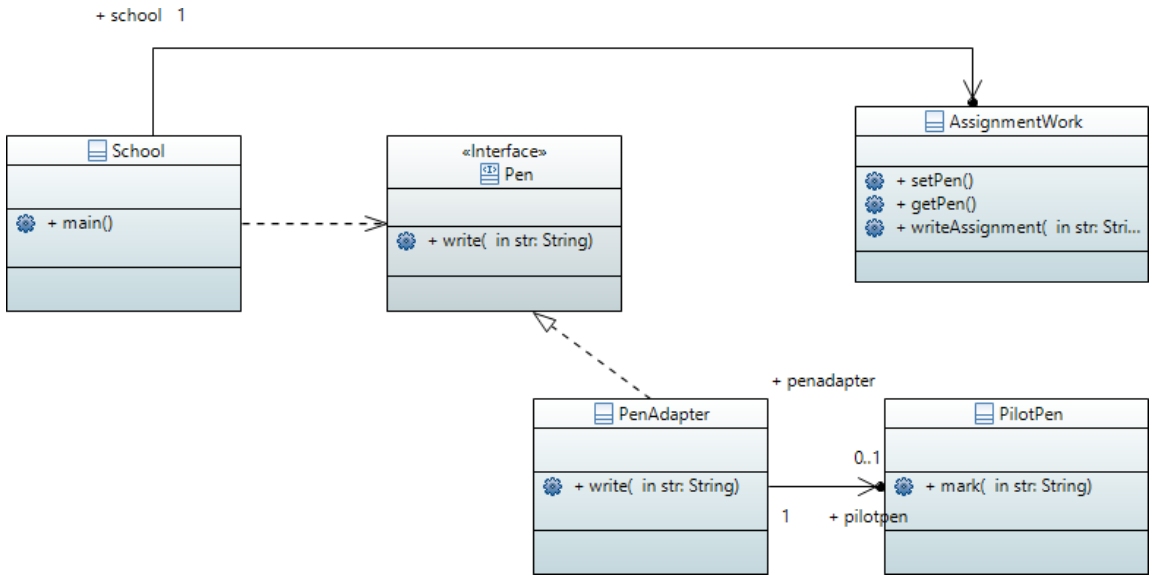
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

Applicability

- When we want two incompatible classes to interact with each other

Example:

In every school we get an assignment to work on, and to write that assignment we require a pen. So let's say writing assignment is a function and to perform that function we need a pen.
In this example we have School class containing main method and we have AssignmentWork class for writing assignment so to write assignment we require Pen and Pen is the interface here having write method. Here we do not have the implementation of Pen but we have the implementation of PilotPen. To use the pen we have to combine it with PilotPen. This can be done by the PenAdapter, which will take PilotPen object and it will provide pen object.



Pen.java

```
public interface Pen {
	void write(String str);
}
```

PilotPen.java

```
	class PilotPen{
		public void mark(String str){
			System.out.println(str);
		}
	}
```

PenAdapter.java

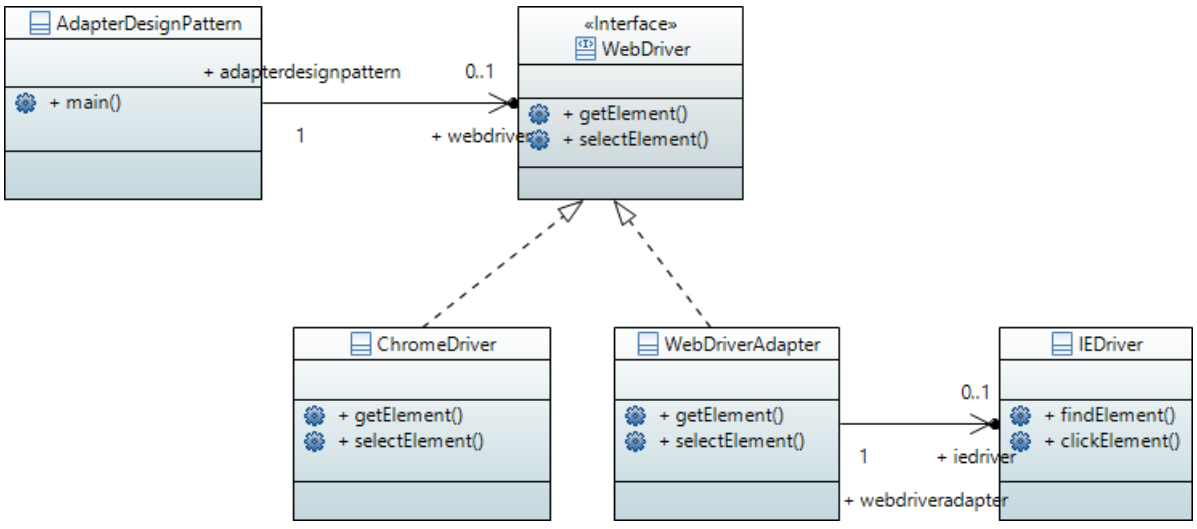
```
class PenAdapter implements Pen{
	PilotPen pp = new PilotPen();

	@Override
	public void write(String str){
		pp.mark(str);
	}
}
```

AssignmentWork.java

```
class AssignmentWork{
        private Pen pen;

        public void setPen(Pen pen){
                this.pen = pen;
        }

        public Pen getPen(){
                return pen;
        }

        public void writeAssignment(String str){
                pen.write(str);
        }
}
```

Example 2

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 lets 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 adap 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
  public void getElement() {
    System.out.println("Get element from ChromeDriver");
  }

  @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
  public void getElement() {
    ieDriver.findElement();

  }

  @Override
  public void selectElement() {
    ieDriver.clickElement();
  }

}
```

Main.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();

  }

}
```