NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Object Oriented Analysis & Design (CL-309)

Muhammad Nadeem || Hamza Ahmed
muhammad.nadeem@nu.edu.pk || hamza.ahmed@nu.edu.pk

Lab Session # 11

Objective: To Understand Adapter and Factory design Patter

Adapter Pattern

Introduction:

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plug-in the memory cards into card reader and card reader into the laptop so that memory card can be read via laptop.

Motivation:

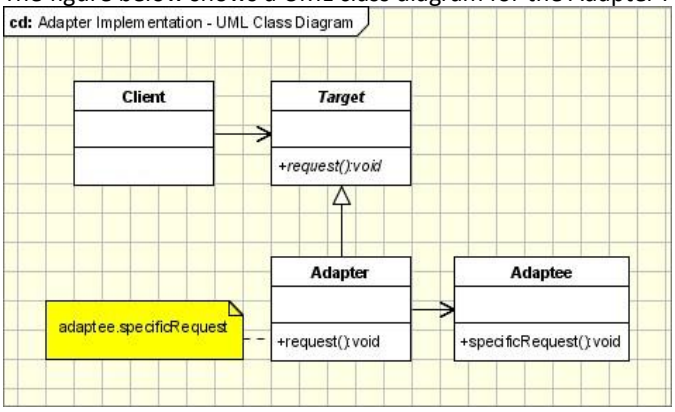
The adapter pattern is adapting between classes and objects. Like any adapter in the real world it is used to be an interface, a bridge between two objects. In real world we have adapters for power supplies, adapters for camera memory cards, and so on. Probably everyone has seen some adapters for memory cards. If you cannot plug in the camera memory in your laptop you can use and adapter. You plug the camera memory in the adapter and the adapter in to laptop slot. That's it, it's really simple.
What about software development? It's the same. Can you imagine an situation when you have some class expecting some type of object and you have an object offering the same features, but exposing a different interface? Of course, you want to use both of them so you don't to implement again one of them, and you don't want to change existing classes, so why not create an adapter.

Intent:

Convert the interface of a class into another interface clients expect.
Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

Implementation:

The figure below shows a UML class diagram for the Adapter Pattern:



The classes/objects participating in adapter pattern:

Target - defines the domain-specific interface that Client uses.

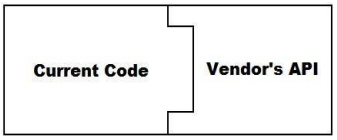
Adapter - adapts the interface Adaptee to the Target interface.
Adaptee - defines an existing interface that needs adapting.
Client - collaborates with objects conforming to the Target interface.

Applicability & Examples:

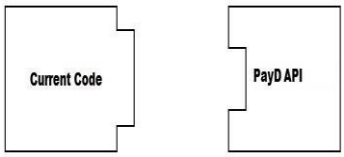
The visitor pattern is used when:
When you have a class(Target) that invokes methods defined in an interface and you have a another class(Adapter) that doesn't implement the interface but implements the operations that should be invoked from the first class through the interface. You can change none of the existing code. The adapter will implement the interface and will be the bridge between the 2 classes. When you write a class (Target) for a generic use relying on some general interfaces and you have some implemented classes, not implementing the interface that needs to be invoked by the Target class. Adapters are encountered everywhere. From real world adapters to software adapters.
Non Software Examples of Adapter Patterns: Power Supply Adapters, card readers and adapters, ...
Software Examples of Adapter Patterns: Wrappers used to adopt 3rd parties libraries and frameworks - most of the applications using third party libraries use adapters as a middle layer between the application and the 3rd party library to decouple the application from the library. If another library has to be used only an adapter for the new library is required without having to change the application code.

Specific problems and implementation:

A software developer, Max, has worked on an e-commerce website. The website allows users to shop and pay online. The site is integrated with a 3rd party payment gateway, through which users can pay their bills using their credit card. Everything was going well, until his manager called him for a change in the project.The manager told him that they are planning to change the payment gateway vendor, and he has to implement that in the code.
The problem that arises here is that the site is attached to the Xpay payment gateway which takes an Xpay type of object. The new vendor, PayD, only allows the PayD type of objects to allow the process. Max doesn't want to change the whole set of 100 of classes which have reference to an object of type XPay. This also raises the risk on the project, which is already running on the production. Neither he can change the 3rd party tool of the payment gateway. The problem has occurred due to the incompatible interfaces between the two different parts of the code. In order to get the process work, Max needs to find a way to make the code compatible with the vendor's provided API.
Current Code with the Xpay's API.



Now, the current code interface is not compatible with the new vendor's interface.



What Max needs here is an Adapter which can sit in between the code and the vendor's API, and can allow the process to flow.

Example for an adapter pattern:

Let's implement X pays API example.

```
package com.javacodegeeks.patterns.adapterpattern.xpay;

public interface Xpay {

        public String getCreditCardNo();
        public String getCustomerName();
        public String getCardExpMonth();
        public String getCardExpYear();
        public Short getCardCVVNo();
        public Double getAmount();

        public void setCreditCardNo(String creditCardNo);
        public void setCustomerName(String customerName);
        public void setCardExpMonth(String cardExpMonth);
        public void setCardExpYear(String cardExpYear);
        public void setCardCVVNo(Short cardCVVNo);
        public void setAmount(Double amount);

}
```

It contains set of setters and getter method used to get the information about the credit card and customer name. This X pay interface is implemented in the code which is used to instantiate an object of this type, and exposes the object to the vendor's API. The following class defines the implementation to the X pay interface.

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayImpl implements Xpay{

	private String creditCardNo;
	private String customerName;
	private String cardExpMonth;
	private String cardExpYear;
	private Short cardCVVNo;
	private Double amount;

	@Override
	public String getCreditCardNo() {
		return creditCardNo;
	}

	@Override
	public String getCustomerName() {
		return customerName;
	}

	@Override
	public String getCardExpMonth() {
		return cardExpMonth;
	}

	@Override
	public String getCardExpYear() {
		return cardExpYear;
	}
```

```
	@Override
	public Short getCardCVVNo() {
		return cardCVVNo;
	}

	@Override
	public Double getAmount() {
		return amount;
	}

	@Override
	public void setCreditCardNo(String creditCardNo) {
		this.creditCardNo = creditCardNo;
	}

	@Override
	public void setCustomerName(String customerName) {
		this.customerName = customerName;
	}

	@Override
	public void setCardExpMonth(String cardExpMonth) {
		this.cardExpMonth = cardExpMonth;
	}

	@Override
	public void setCardExpYear(String cardExpYear) {
		this.cardExpYear = cardExpYear;
	}

	@Override
	public void setCardCVVNo(Short cardCVVNo) {
		this.cardCVVNo = cardCVVNo;
	}

	@Override
	public void setAmount(Double amount) {
		this.amount = amount;
	}

}
```

New vendor's key interface looks like this:

```
package com.javacodegeeks.patterns.adapterpattern.payd;

public interface PayD {

	public String getCustCardNo();
	public String getCardOwnerName();
	public String getCardExpMonthDate();
	public Integer getCVVNo();
	public Double getTotalAmount();

	public void setCustCardNo(String custCardNo);
	public void setCardOwnerName(String cardOwnerName);
	public void setCardExpMonthDate(String cardExpMonthDate);
	public void setCVVNo(Integer cVVNo);
	public void setTotalAmount(Double totalAmount);
}
```

As you can see, this interface has a set of different methods which need to be implemented in the code. But Xpay is created by most part of the code, it's really hard and risky to change the entire set of classes.
We need some way, that's able to fulfill the vendor's requirement in order to process the payment and also make less or no change in the current code. The way is provided by the Adapter pattern.
We will create an adapter which will be of type PayD, and it wraps an Xpay object (the type it supposes to be adapted).

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayToPayDAdapter implements PayD{

	private String custCardNo;
	private String cardOwnerName;
	private String cardExpMonthDate;
	private Integer cVVNo;
	private Double totalAmount;

	private final Xpay xpay;

	public XpayToPayDAdapter(Xpay xpay){
		this.xpay =  xpay;
		setProp();
	}

	@Override
	public String getCustCardNo() {
		return custCardNo;
	}

	@Override
	public String getCardOwnerName() {
		return cardOwnerName;
	}

	@Override
	public String getCardExpMonthDate() {
		return cardExpMonthDate;
	}

	@Override
	public Integer getCVVNo() {
		return cVVNo;
	}

	@Override
	public Double getTotalAmount() {
		return totalAmount;
	}

	@Override
	public void setCustCardNo(String custCardNo) {
		this.custCardNo = custCardNo;
	}

	@Override
	public void setCardOwnerName(String cardOwnerName) {
		this.cardOwnerName = cardOwnerName;
	}
```

```
        @Override
        public void setCardExpMonthDate(String cardExpMonthDate) {
                this.cardExpMonthDate = cardExpMonthDate;
        }

        @Override
        public void setCVVNo(Integer cVVNo) {
                this.cVVNo = cVVNo;
        }

        @Override
        public void setTotalAmount(Double totalAmount) {
                this.totalAmount = totalAmount;
        }

        private void setProp(){
                setCardOwnerName(this.xpay.getCustomerName());
                setCustCardNo(this.xpay.getCreditCardNo());
                setCardExpMonthDate(this.xpay.getCardExpMonth()+"/"+this.xpay.
                    getCardExpYear());
                setCVVNo(this.xpay.getCardCVVNo().intValue());
                setTotalAmount(this.xpay.getAmount());
        }

}
```

In the above code, we have created an Adapter(XpayToPayDAdapter). The adapter implements the PayD interface, as it is required to mimic like a PayD type of object. The adapter uses object composition to hold the object, it's supposed to be adapting, an Xpay type of object. The object is passed into the adapter through its constructor.Now, please note that we have two incompatible types of interfaces, which we need to fit together using an adapter in order to make the code work. These two interfaces have a different set of methods. But the sole purpose of these interfaces is very much similar, i.e. to provide the customer and credit card info to their specific vendors.The setProp() method of the above class is used to set the xpay's properties into the payD's object. We set the methods which are similar in work in both the interfaces. However, there is only single method in PayD interface to set the month and the year of the credit card, as opposed to two methods in the Xpay interface.We joined the result of the two methods of the Xpay object (this.xpay.getCardExpMonth()+"/"+this.xpay.getCardExpYear()) and sets it into the setCardExpMonthDate() method.Let us test the above code and see whether it can solve the Max's problem.

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class RunAdapterExample {

        public static void main(String[] args) {

                // Object for Xpay
                Xpay xpay = new XpayImpl();
                xpay.setCreditCardNo("4789565874102365");
                xpay.setCustomerName("Max Warner");
                xpay.setCardExpMonth("09");
                xpay.setCardExpYear("25");
                xpay.setCardCVVNo((short)235);
                xpay.setAmount(2565.23);

                PayD payD = new XpayToPayDAdapter(xpay);
                testPayD(payD);
```

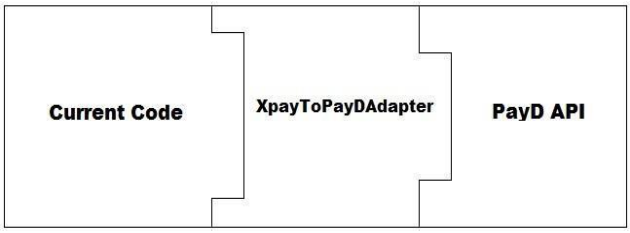
```
        }

        private static void testPayD(PayD payD){

                System.out.println(payD.getCardOwnerName());
                System.out.println(payD.getCustCardNo());
                System.out.println(payD.getCardExpMonthDate());
                System.out.println(payD.getCVVNo());
                System.out.println(payD.getTotalAmount());
        }
}
```

In the above class, first we have created an Xpay object and set its properties. Then, we created an adapter and pass it that xpay object in its constructor, and assigned it to the PayD interface. The testPayD() static method takes a PayD type as an argument which run and print its methods in order to test. As far as, the type passed into the testPayD() method is of type PayD the method will execute the object without any problem. Above, we passed an adapter to it, which looks like a type of PayD, but internally it wraps an Xpay type of object.

So, in the Max's project all we need to implement the vendor's API in the code and pass this adapter to the vendor's method to make the payment work. We do not need to change anything in the existing code.



Factory Pattern

Motivation:

The Factory Design Pattern is probably the most used design pattern in modern programming languages like Java and C#. It comes in different variants and implementations. The essence of the Factory method Pattern is to "Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses."

In simple words, if we have a super class and n sub-classes, and based on data provided, we have to return the object of one of the sub-classes, we use a factory pattern.

The idea behind the Factory Method pattern is that it allows for the case where a client doesn't know what concrete classes it will be required to create at runtime, but just wants to get a class that will do the job while AbstractFactory pattern is best utilized when your system has to create multiple families of products or you want to provide a library of products without exposing the implementation details.

Intent:

1) Creates objects without exposing the instantiation logic to the client.

2) Refers to the newly created object through a common interface.

The Factory Method pattern uses an abstract class called a factory to instantiate new objects.

Factory methods encapsulate the creation of objects. Creating objects without exposing the instantiation logic to client referring to the newly created objects through a common interface.

So basically the factory pattern is used wherever the sub classes are given the privileged of instantiating a method that can create an object.

Factory Method is used to create one product only but Abstract Factory is about creating families of related or dependent products. Factory Method pattern exposes a method to the client for creating the object whereas in case of Abstract Factory they expose a family of related objects which may consist of these Factory methods. Factory Method pattern hides the construction of single object whereas Abstract factory method hides the construction of a family of related objects. Abstract factories are usually implemented using (a set of) factory methods.AbstractFactory pattern uses composition to delegate responsibility of creating object to another class while Factory design pattern uses inheritance and relies on derived class or sub class to create object.

Factory Example:

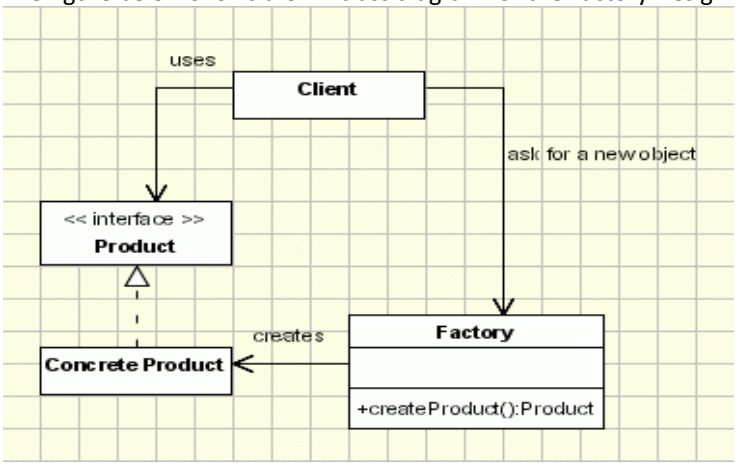
Imagine you are constructing a house and you approach a carpenter for a door. You give the measurement for the door and your requirements, and he will construct a door for you. In this case, the carpenter is a factory of doors. Your specifications are inputs for the factory, and the door is the output or product from the factory.

Factory Pattern:

1. Creates object through inheritance.
2. Produce only one product.
3. Implements code in the abstract creator that makes use of the concrete type that sub-class produces.

Implementation:

The figure below shows a UML class diagram for the Factory Design Pattern:



The implementation is really simple the client needs a product, but instead of creating it directly using the new operator, it asks the factory object for a new product, providing the information about the type of object it needs. The factory instantiates a new concrete product and then returns to the client the newly created product (casted to abstract product class).The client uses the products as abstract products without being aware about their concrete implementation.

Specific problems and implementation:

Example for Factory class:

Step 1:

Factory Design Pattern Super Class

Super class in factory design pattern can be an interface, abstract class or a normal java class. For our factory design pattern example, we have abstract super class with overridden toString() method for testing purpose.

```
Package com.journaldev.design.model;

Public abstract class Computer {

        Public abstract String getRAM ();

        Public abstract String getHDD ();

        Public abstract String getCPU ();

        @Override

        Public String toString (){

                return "RAM= "+this.getRAM ()+", HDD="+this.getHDD()+",
CPU="+this.getCPU();

        }

}
```

Step 2:

Factory Design Pattern Sub Classes

Let's say we have two sub-classes PC and Server with below implementation.

```
Package com.journaldev.design.model;

Public class PC extends Computer {

        Private String ram;
```

```
        Private String hdd;
        Private String cpu;
        Public PC(String ram, String hdd, String cpu)
{       this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
        }
        @Override
        Public String getRAM() {
                return this.ram;
        }
        @Override
        Public String getHDD () {
                return this.hdd;
        }
        @Override
        Public String getCPU() {
                return this.cpu;
        }

}
```

```
Package com.journaldev.design.model;
Public class Server extends Computer {
        Private String ram;
        Private String hdd;
        Private String cpu;
        Public Server (String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
        }
        @Override
        Public String getRAM () {
        return this.ram;
        }
```

```
        @Override
        Public String getHDD() {
        return this.hdd;
        }
        @Override
        public String getCPU() {
        return this.cpu;
        }
}
```

Notice that both the classes are extending Computer super class.

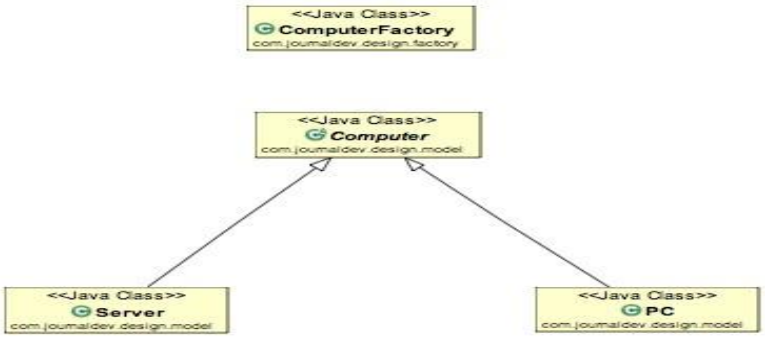
Step 3:

Factory Class

Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation.

```
Package com.journaldev.design.factory;
import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;
import com.journaldev.design.model.Server;
Public class ComputerFactory {
        Public static Computer getComputer (String type, String ram, String hdd,
String cpu){
                If ("PC".equalsIgnoreCase (type)) return new PC(ram, hdd, cpu);
                else if("Server".equalsIgnoreCase (type)) return new Server(ram, hdd,
cpu);
                return null;
        }
}
```

Some important points about Factory Design Pattern method are;
1. We can keep Factory class Singleton or we can keep the method that returns the subclass as static.
2. Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.



Step 4:

Here is a simple test client program that uses above factory design pattern implementation.

```
Package com.journaldev.design.test;


import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;


Public class TestFactory {


        Public static void main (String [] args) {
                Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4
GHz");
                Computer server = ComputerFactory.getComputer ("server","16 GB","1
TB","2.9 GHz");
                System.out.println ("Factory PC Config::"+pc);
                System.out.println ("Factory Server Config::"+server);
        }


}
```

Output of above program is:

```
Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz
Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

Factory Design Pattern Advantages:

1. Factory design pattern provides approach to code for interface rather than implementation

2. Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
3. Factory pattern provides abstraction between implementation and client classes through inheritance.