**Design Pattern**

**Introduction:**

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software **design**. A **design pattern** isn't a finished **design** that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

**Facade:**

In our Railway Reservation system, we have used the Facade pattern to simplify and modularize CRUD operations to the database.

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities. This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

**Problem Context:**

Database operations generally are straight foward, but complex operations. We have chosen to make use of a class called DatabaseFacade, that provides a simplified interface to the various database accessor classes and databases operations.

**Solution:**

Instead of separate database accessor classes, we use a single Façade.

**Structure of Façade Pattern:**

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

We have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc.

So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports.

But when the complexity increases or the interface behaviour names are confusing, client application will find it difficult to manage it.

So we can apply Facade design pattern here and provide a wrapper interface on top of the existing interface to help client application.

Facade Design Pattern – Set of Interfaces

We can have two helper interfaces, namely MySqlHelper and OracleHelper.

package com.journaldev.design.facade;

import java.sql.Connection;

public class MySqlHelper{

publicstaticConnection getMySqlDBConnection(){

//get MySql DB connection using connection parameters

returnnull;

}

publicvoid generateMySqlPDFReport(String tableName,Connection con){

//get data from table and generate pdf report

}

publicvoid generateMySqlHTMLReport(String tableName,Connection con){

//get data from table and generate pdf report

}

}

This is important for report generation in our project.

### Facade Design Pattern Interface

We can create a Facade pattern interface like below. Notice the use of Java Enum for type safety.

package com.journaldev.design.facade;

import java.sql.Connection;

publicclassHelperFacade{

publicstaticvoid generateReport(DBTypes dbType,ReportTypes reportType,String tableName){

Connection con =null;

switch(dbType){

case MYSQL:

con =MySqlHelper.getMySqlDBConnection();

MySqlHelper mySqlHelper =newMySqlHelper();

switch(reportType){

case HTML:

mySqlHelper.generateMySqlHTMLReport(tableName, con);

break;

case PDF:

mySqlHelper.generateMySqlPDFReport(tableName, con);

break;

}

break;

case ORACLE:

con =OracleHelper.getOracleDBConnection();

OracleHelper oracleHelper =newOracleHelper();

switch(reportType){

case HTML:

oracleHelper.generateOracleHTMLReport(tableName, con);

break;

case PDF:

oracleHelper.generateOraclePDFReport(tableName, con);

break;

}

break;

}

}

publicstaticenumDBTypes{

MYSQL,ORACLE;

}

publicstaticenumReportTypes{

HTML,PDF;

}

}

### This covers both MySql and Oracle database.

### Facade Design Pattern Client Program

Now let’s see client code without using Facade pattern and using Facade pattern interface.

package com.journaldev.design.test;

import java.sql.Connection;

import com.journaldev.design.facade.HelperFacade;

import com.journaldev.design.facade.MySqlHelper;

import com.journaldev.design.facade.OracleHelper;

publicclassFacadePatternTest{

publicstaticvoid main(String[] args){

String tableName="Employee";

//generating MySql HTML report and Oracle PDF report without using Facade

Connection con =MySqlHelper.getMySqlDBConnection();

MySqlHelper mySqlHelper =newMySqlHelper();

mySqlHelper.generateMySqlHTMLReport(tableName, con);

Connection con1 =OracleHelper.getOracleDBConnection();

OracleHelper oracleHelper =newOracleHelper();

oracleHelper.generateOraclePDFReport(tableName, con1);

//generating MySql HTML report and Oracle PDF report using Facade

HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL,HelperFacade.ReportTypes.HTML, tableName);

HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE,HelperFacade.ReportTypes.PDF, tableName);

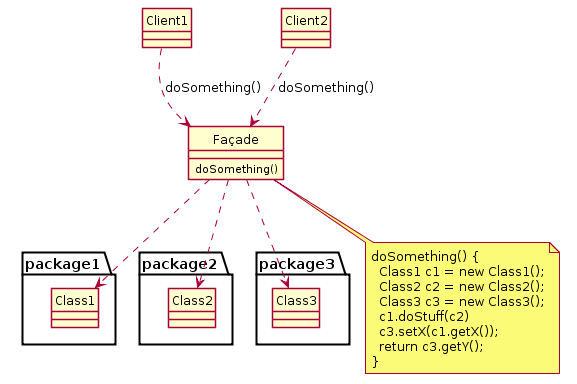
}

}

As you can see that using Facade pattern interface is a lot easier and cleaner way to avoid having a lot of logic at client side. JDBC Driver Manager class to get the database connection is a wonderful example of facade design pattern.

### Facade Design Pattern Important Points

* Facade design pattern is more like a helper for client applications, it doesn’t hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
* Facade design pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
* Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.
* Facade design pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
* We can use [Factory pattern](http://www.journaldev.com/1392/factory-design-pattern-in-java) with Facade to provide better interface to client systems.

****

**Basic structure of façade pattern being used in the project.**

**POSTLAB QUESTONS**

**Q1)What are the different usages of design patterns ?**

## **Usage of Design Pattern**

Design Patterns have two main usages in software development.

### Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

### Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

**Q2)What are the different categories of design patterns ?**

## **Types of Design Patterns**

As per the design pattern referencebook **Design Patterns - Elements of Reusable Object-Oriented Software** , there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioural patterns. We'll also discuss another category of design pattern: J2EE design patterns.

|  |  |
| --- | --- |
| **S.N.** | **Pattern & Description** |
| 1 | **Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case. |
| 2 | **Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. |
| 3 | **Behavioural Patterns** These design patterns are specifically concerned with communication between objects. |
| 4 | **J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center. |

**Q3) Explain FAÇADE pattern. Show Implementation of Façade pattern with example.**

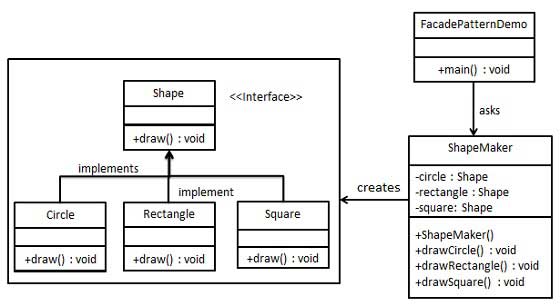
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

## **Implementation**

We are going to create a “Shape” interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

*ShapeMaker* class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



## **Step 1**

Create an interface.

Shape.java

publicinterfaceShape{

void draw();

}

## **Step 2**

Create concrete classes implementing the same interface.

*Rectangle.java*

publicclassRectangleimplementsShape{

@Override

publicvoid draw(){

System.out.println("Rectangle::draw()");

}

}

*Square.java*

publicclassSquareimplementsShape{

@Override

publicvoid draw(){

System.out.println("Square::draw()");

}

}

*Circle.java*

publicclassCircleimplementsShape{

@Override

publicvoid draw(){

System.out.println("Circle::draw()");

}

}

## **Step 3**

Create a facade class.

*ShapeMaker.java*

public class ShapeMaker{

private Shape circle;

private Shape rectangle;

private Shape square;

public ShapeMaker(){

circle =newCircle();

rectangle =newRectangle();

square =newSquare();

}

publicvoid drawCircle(){

circle.draw();

}

publicvoid drawRectangle(){

rectangle.draw();

}

publicvoid drawSquare(){

square.draw();

}

}

## **Step 4**

Use the facade to draw various types of shapes.

*FacadePatternDemo.java*

Public clas sFacadePatternDemo{

public static void main(String[] args){

ShapeMaker shapeMaker =new ShapeMaker();

shapeMaker.drawCircle();

shapeMaker.drawRectangle();

shapeMaker.drawSquare();

}

}

## **Step 5**

Verify the output.

Circle::draw()

Rectangle::draw()

Square::draw()

**Q4) What are the uses and advantages of Façade pattern in the project?**

A Facade Pattern says that just **"**just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client**".**

In other words, Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

Practically, **every Abstract Factory** is a type of **Facade.**

#### **Advantage of Facade Pattern**

* It shields the clients from the complexities of the sub-system components.
* It promotes loose coupling between subsystems and its clients.

#### **Usage of Facade Pattern:**

It is used:

* When you want to provide simple interface to a complex sub-system.
* When several dependencies exist between clients and the implementation classes of an abstraction.

**Q5) When should Design Patterns be used? What are the advantages of using design patterns?**

We must use the design patterns **during the analysis and requirement phase of SDLC** (Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

## **Advantage of design pattern:**

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They capture the software engineering experiences.
4. They provide transparency to the design of an application.
5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6. Design patterns don’t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.