**Design Patterns - Facade Pattern**

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.

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 **Façade.**

**Example:**

 A customer in a restaurant orders food from the menu, which is probably described in half a line. The order goes to the kitchen and the food comes back after a while. **Simple!** The customer doesn’t want to know who will cut the meat for how long will it be cooked and who is going to wash the dishes afterward. The customer just wants to eat a tasty meal that meets the expectations. Therefore, the menu serves as the facade to make it easier for the customer by avoiding the complexities coming from the kitchen or even the tasks that the waiter is assigned through this process.

#### 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.

### Example of Facade Pattern

Let's understand the example of facade design pattern by the above UML diagram.

#### UML for Facade Pattern:

****

## 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*

public interface Shape {

 void draw();

}

## Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

 @Override

 public void draw() {

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

 }

}

*Square.java*

public class Square implements Shape {

 @Override

 public void draw() {

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

 }

}

*Circle.java*

public class Circle implements Shape {

 @Override

 public void 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 = new Circle();

 rectangle = new Rectangle();

 square = new Square();

 }

 public void drawCircle(){

 circle.draw();

 }

 public void drawRectangle(){

 rectangle.draw();

 }

 public void drawSquare(){

 square.draw();

 }

}

## Step 4

Use the facade to draw various types of shapes.

*FacadePatternDemo.java*

public class FacadePatternDemo {

 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()