Facade Design Pattern

With java

Introduction to Facade Design Pattern

- Facade design pattern is another pattern from structural design pattern category. This supports loose coupling by hiding the complex details via a simple interface.
- This pattern introduces a component called facade, which is a simplified interface component.
- This facade simplifies the responsibilities of the client by standing between the complex sub-systems.
- The complex subsystems can be a third party library, a legacy code, or lower level heavy code collections with a number of components with different modules.
- Usually, the sub-systems contain several complex processes, which require separate attention to executing.

Note to Remember

- Does the ‘facade’ component in the pattern as same as the Java interface? No, it’s not the same concept.
- Facade doesn’t need exactly be an interface. It is just another layer to abstract the inner services and hides the complexity.
- Facade can be an abstract or concrete class. This facade sometimes known as a conceptual component, since we can to implement the additional layer in any form.

GoF definition for facade:

‘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.’

This pattern removes a complexity of a code by introducing an additional layer over a set of complex components. It creates a simplified interface that performs many other actions behind the scenes. This higher-level unified interface covers the underlying structural complexity.

Simple Real Life Examples - ATM Machine

- Let’s assume a situation where a customer needs to withdraw money from a bank.
- When you insert the card at ATM the internal system performs several tasks and checks conditions before approving the withdrawal.
- Internal bank system checks whether the card is active and valid, then whether the account is valid, whether the entered security code is valid and match with the account holder and whether there is enough balance to release the money.
- The ATM will release money only after validating all the above tasks. Thus, there are many different processes behind the scene, in withdrawing money from an ATM.

Simple Real Life Examples - ATM Machine

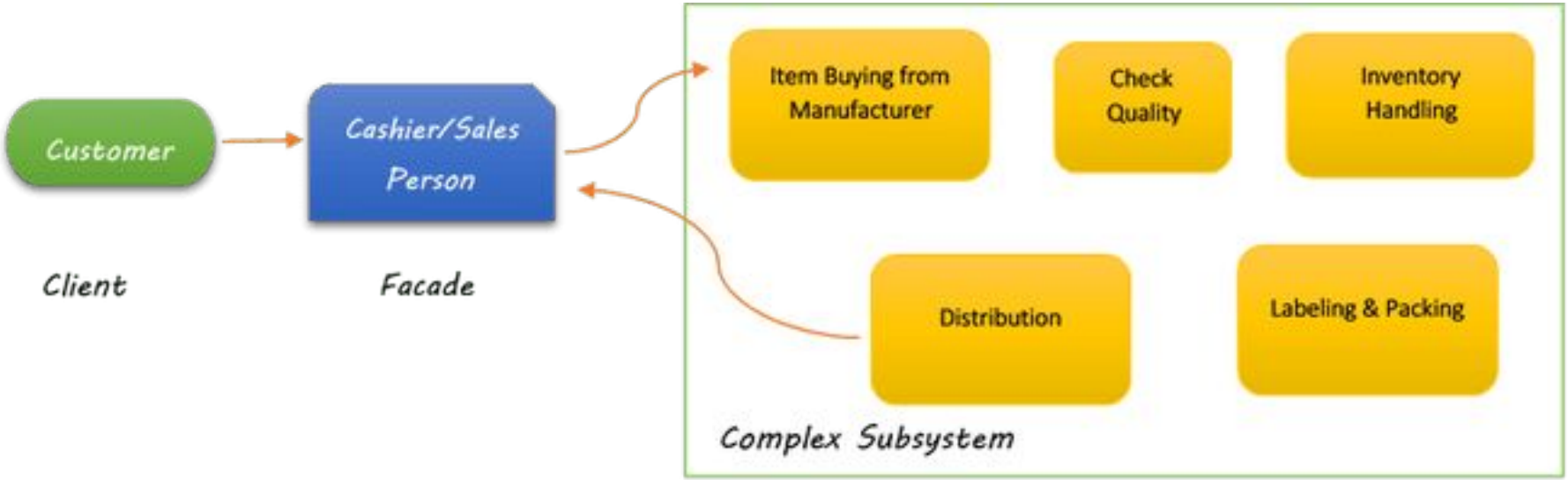


Figure 1 - Facade Pattern Real Life Example

What is Façade?

- The ‘Facade’ is a component, which represents an additional layer.
- It provides the main solution to the problem of dealing with complex sub-systems.
- This is a more simplified interface, which simply acts as the point of entry to the complicated subsystem.
- In a more standard way, façade is a wrapper component that encapsulates a sub-system in order to hide the complexity of the subsystem.
- The ‘Wrapper’ contains set of members who access the subsystem on behalf of the façade user.
- It hides the inner processes and client only needs to communicate with façade in order to fulfil this requirement. In further complex systems, a façade class can be used to wrap all the interfaces and classes for a subsystem.

How Does Façade Pattern work?

Façade pattern mainly consists of two sections.

- The external interface is known as the facade
- The internal complex subsystem

When a client needs to get some service from the system, the external interface is the component, which directly interacts with the client.

The client can communicate with the internal complex system only by sending requests to the façade.

Façade will forward the requests to the suitable subsystem component.

Façade will translate the client request as suited for the subsystem receiving interfaces. Subsystem objects will perform the actual work.

Hence, the client does not know the behaviours or the interactions of the internal subsystem.

Façade pattern ‘information hiding’ design principle to prevent all client classes from seeing the internal class functionalities and object behaviours.

However, the client can deal with the inner system when requires. The system will still expose the lower-level components for the clients who have any interest.

Façade just provides an extra layer to cover the complex subsystem.

When to use Façade Pattern

- system gets loaded and complex to handle by simple client calls
- there is a need for abstraction to hide the internal complex procedures
- classes of a system are tightly –coupled
- there are many dependencies between system implementation and the client
- there is heavy learning curve to understand the legacy system
- there is a need for layering to the each subsystem

Advantage of Facade Pattern

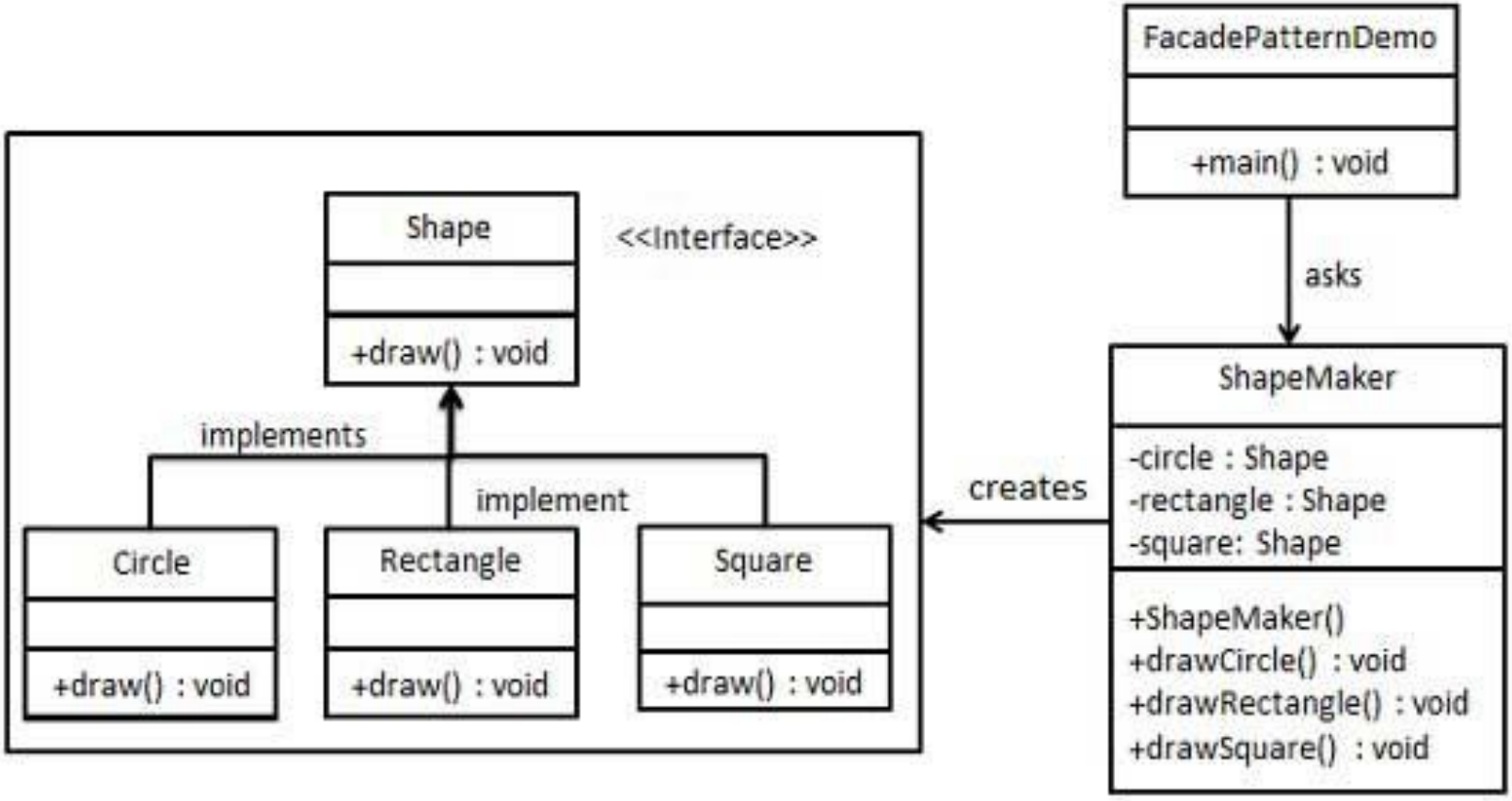
- It shields the client from dealing with the complicated inner system
- Facilitate loose-coupling to reduce the complexity
- Can use to shield the legacy and poorly designed code which is unable to refactor
- Ability to introduce more the one façade to one system to layer the subsystems
- Reduce the learning curve for new developers

Drawbacks of Façade Design Pattern

- If used without a real need, your system may get further complicated by unnecessary codes
- You have to keep a keen eye on modifications to the system. If any unmonitored modifications to the subsystem might break the façade execution

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.


FacadePatternDemo
+main() : void
asks
ShapeMaker
-circle : Shape
-rectangle : Shape
-square: Shape
+ShapeMaker()
+drawCircle() : void
+drawRectangle() : void
+drawSquare() : void
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
+draw() : void
Rectangle
+draw() : void
Square
+draw() : void

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()
```