Software Design and Analysis

CS-3004

Design Patterns

Lecture # 37, 38, 39

Rubab Jaffar
rubab.jaffar@nu.edu.pk

Today's Outline

- Factory Pattern
  - Intent
  - Example
  - Motivation
  - UML Class Diagram
- Adapter Pattern
  - Intent
  - Example
  - Motivation
  - UML Class Diagram
  - Implementation
- Facade Pattern
  - Intent
  - Example
  - Motivation
  - UML Class Diagram
  - Implementation

Factory Pattern
Type: Creational

Intent

- "Define an interface for creating an object, but let subclasses decide which class to instantiate"
- It lets a class defer instantiation to subclasses at run time.
- It refers to the newly created object through a common interface.
- The factory method pattern is a design pattern that define an interface for creating an object from one among a set of classes based on some logic.
- Factory method pattern is also known as virtual constructor pattern.
- Factory method pattern is the most widely used pattern in the software engineering world

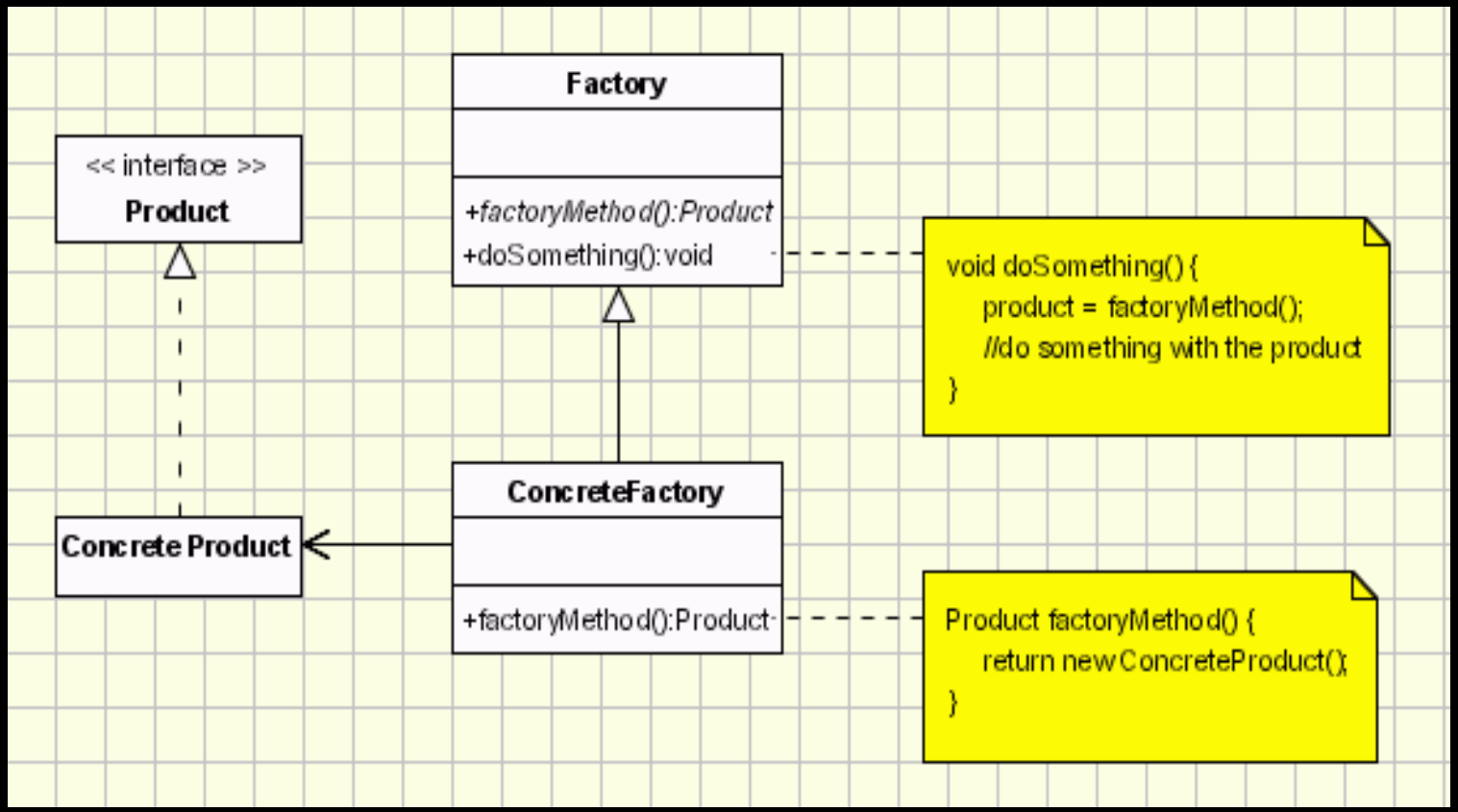
Applicability

- Use the Factory Method pattern when
  - a class can´t anticipate the class of objects it must create.
  - At times, application only knows about the super class (may be abstract class), but doesn't know which sub class (concrete implementation) to be instantiated at compile time.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Participants

- Product
  - Defines the interface of objects the factory method creates
- ConcreteProduct
  - Implements the product interface
- Creator
  - Declares the factory method which returns object of type product
  - May contain a default implementation of the factory method
  - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.
- ConcreteCreator
  - Overrides factory method to return instance of ConcreteProduct

UML class diagram of Factory Design Pattern

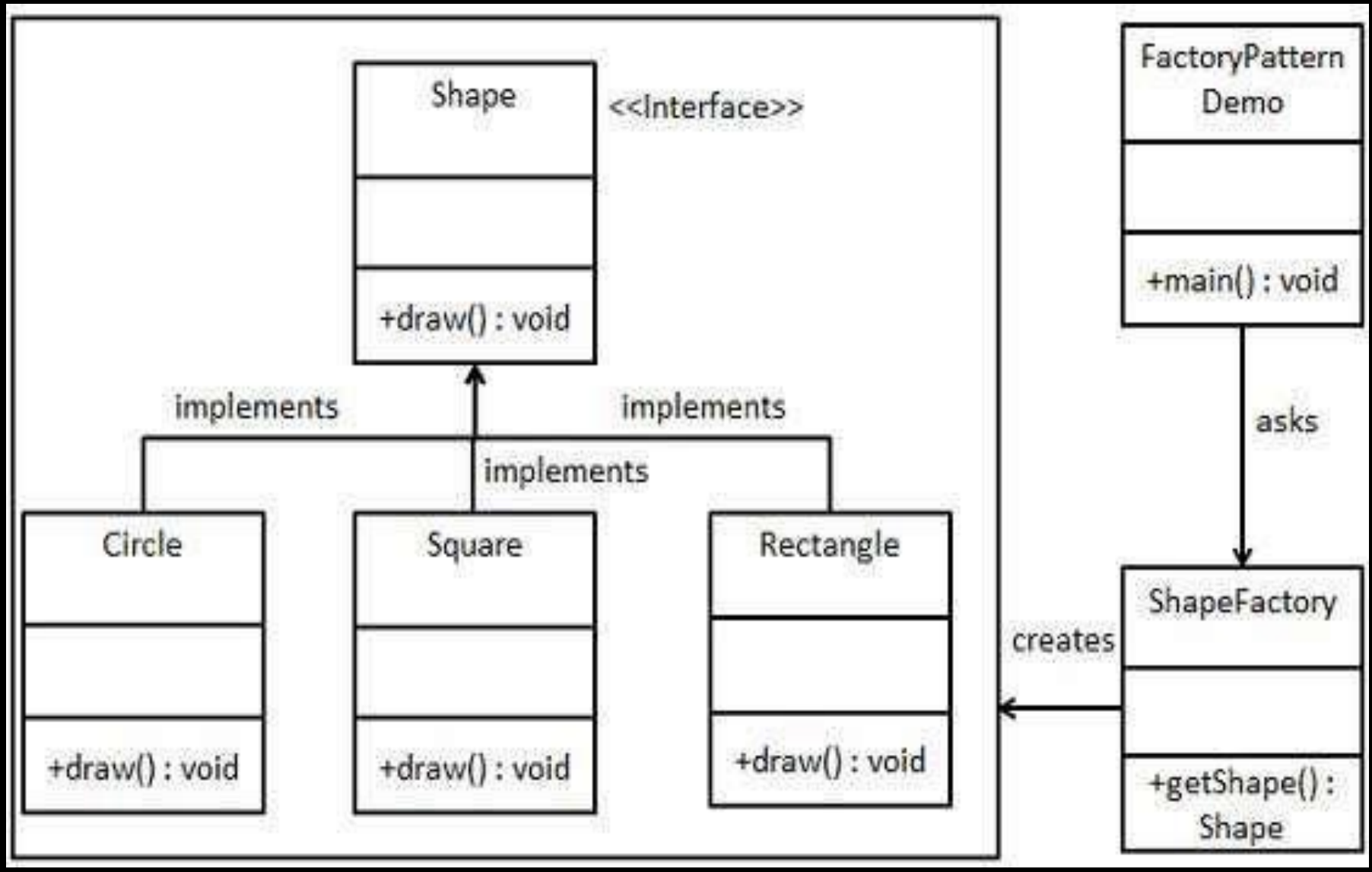


Implementation

- 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.

Example

- We're going to create a *Shape interface and concrete* classes implementing the *Shape interface.*
- *A factory* class *ShapeFactory is defined as a next step.*
- *FactoryPatternDemo, our demo class will* use *ShapeFactory to get a Shape object. It will pass* information *(CIRCLE / RECTANGLE / SQUARE)*to *ShapeFactory to get the type of object it needs.*


Shape
<<Interface>>
+draw() : void
FactoryPattern Demo
+main() : void
asks
implements
implements
implements
Circle
+draw() : void
Square
+draw() : void
Rectangle
+draw() : void
creates
ShapeFactory
+getShape() : Shape

Step 1

- Create an interface. (*Shape.java)*

```
public interface Shape {
void draw();
}
```

Step 2

- Create concrete classes implementing the same interface.

```
public class Rectangle implements Shape {
@Override
public void draw() {
System.out.println("Inside Rectangle::draw() method.");
} }

public class Square implements Shape {
@Override
public void draw() {
System.out.println("Inside Square::draw() method.");
} }

public class Circle implements Shape {
@Override
public void draw() {
System.out.println("Inside Circle::draw() method.");
} }
```

Step 3

- Create a Factory to generate object of concrete class based on given information. *(ShapeFactory.java)*

```
public class ShapeFactory {
//use getShape method to get object of type shape
public Shape getShape(String shapeType){
if(shapeType == null){
return null;
}
if(shapeType.equalsIgnoreCase("CIRCLE")){
return new Circle();
} else if(shapeType.equalsIgnoreCase("RECTANGLE")){
return new Rectangle();
} else if(shapeType.equalsIgnoreCase("SQUARE")){
return new Square();
}
return null;
} }
```

Step 4

- Use Factory to get object of concrete class by passing information such as type. *(FactoryPatternDemo.java)*

```
public class FactoryPatternDemo {
  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();
    Shape shape1 = shapeFactory.getShape("CIRCLE");
    shape1.draw();
    Shape shape2 = shapeFactory.getShape("RECTANGLE");
    shape2.draw();
    Shape shape3 = shapeFactory.getShape("SQUARE");
    shape3.draw();
  }
}
```

Step 5

Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.

Adapter Pattern
Type: Structural

Structural Design Patterns

They deal with how classes and objects deal with to form large structures.

- Structural Design patterns use inheritance to compose interfaces or implementations.
- Structural Design Patterns basically ease the design by identifying the relationships between entities.

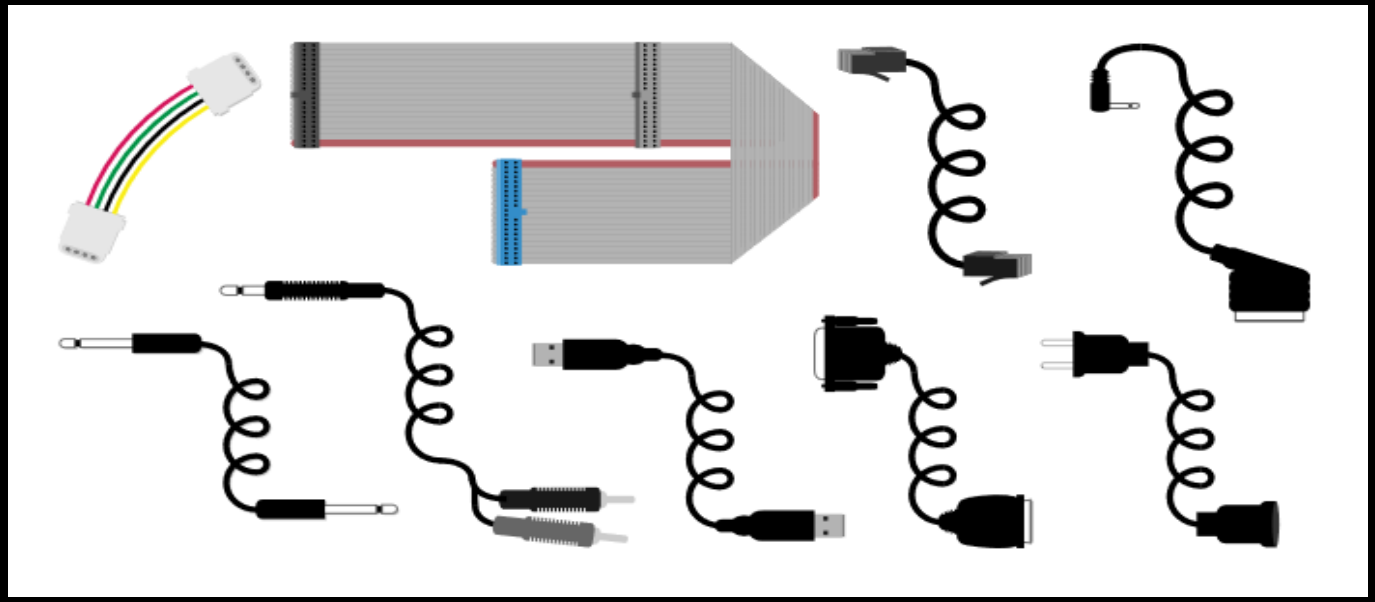
Intent

The Adapter design pattern is useful in situations wher an existing class provides a needed service but there is a mismatch between the interface offered and the interface clients expect.

- The Adapter pattern shows how to convert the interface of the existing class into the interface clients expect.
- Wrap an existing class with a new interface.
- Also known as Wrapper Pattern.

Problem

- An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

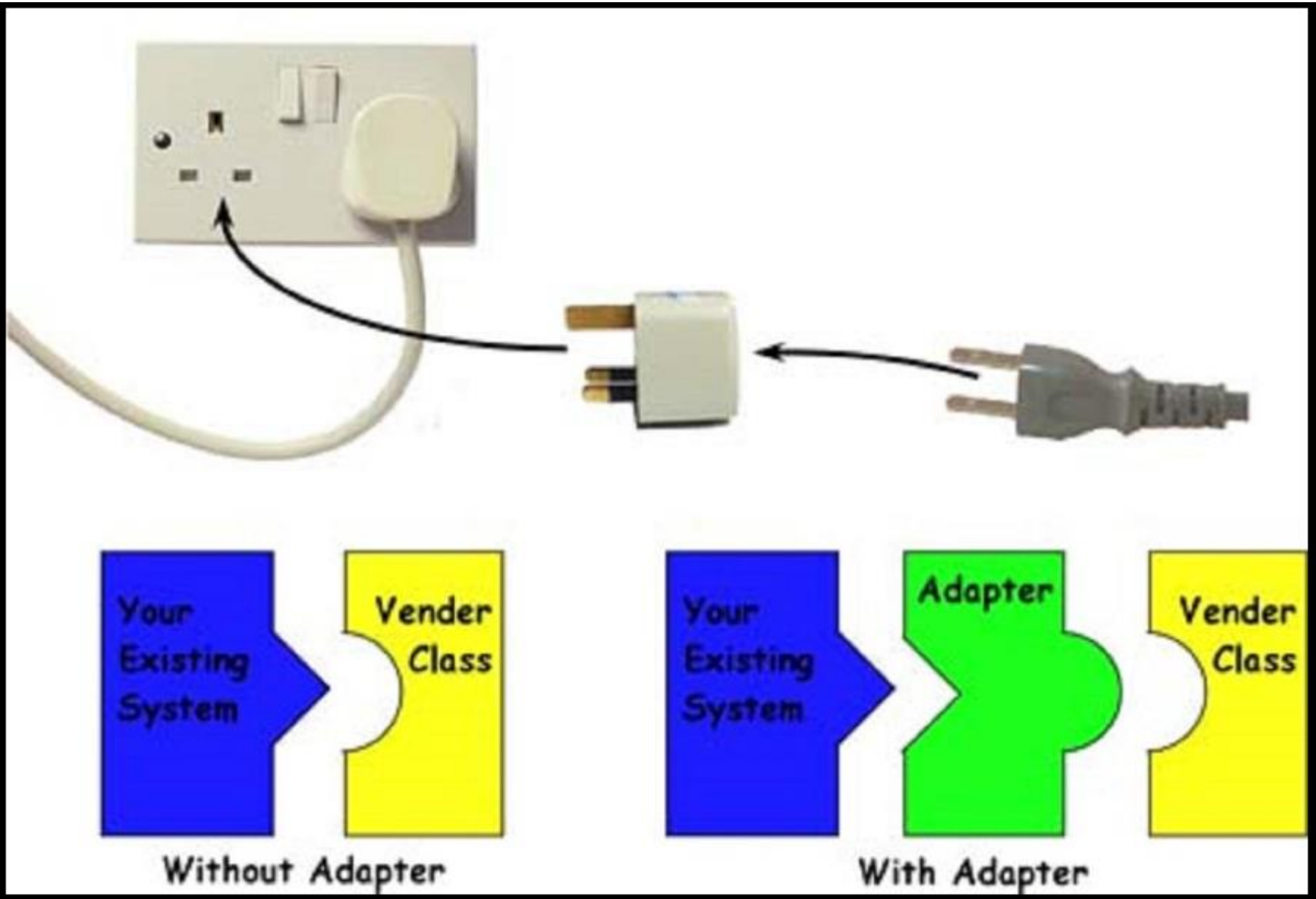


Adapter Motivation

- Situation:
  - You have some code you want to use for a program
  - You can't incorporate the code directly (e.g. you just have the .class file, say as part of a library)
  - The code does not have the interface you want
    - Different method names
    - More or fewer methods than you need
- To use this code, you must *adapt* it to your situation

Adapter Motivation

- Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application.
- We can not change the library interface, since we may not have its source code.
- Even if we did have the source code, we probably should not change the library for each domain-specific application.



The interface doesn't match the one you've written for your code against. This is not going to work

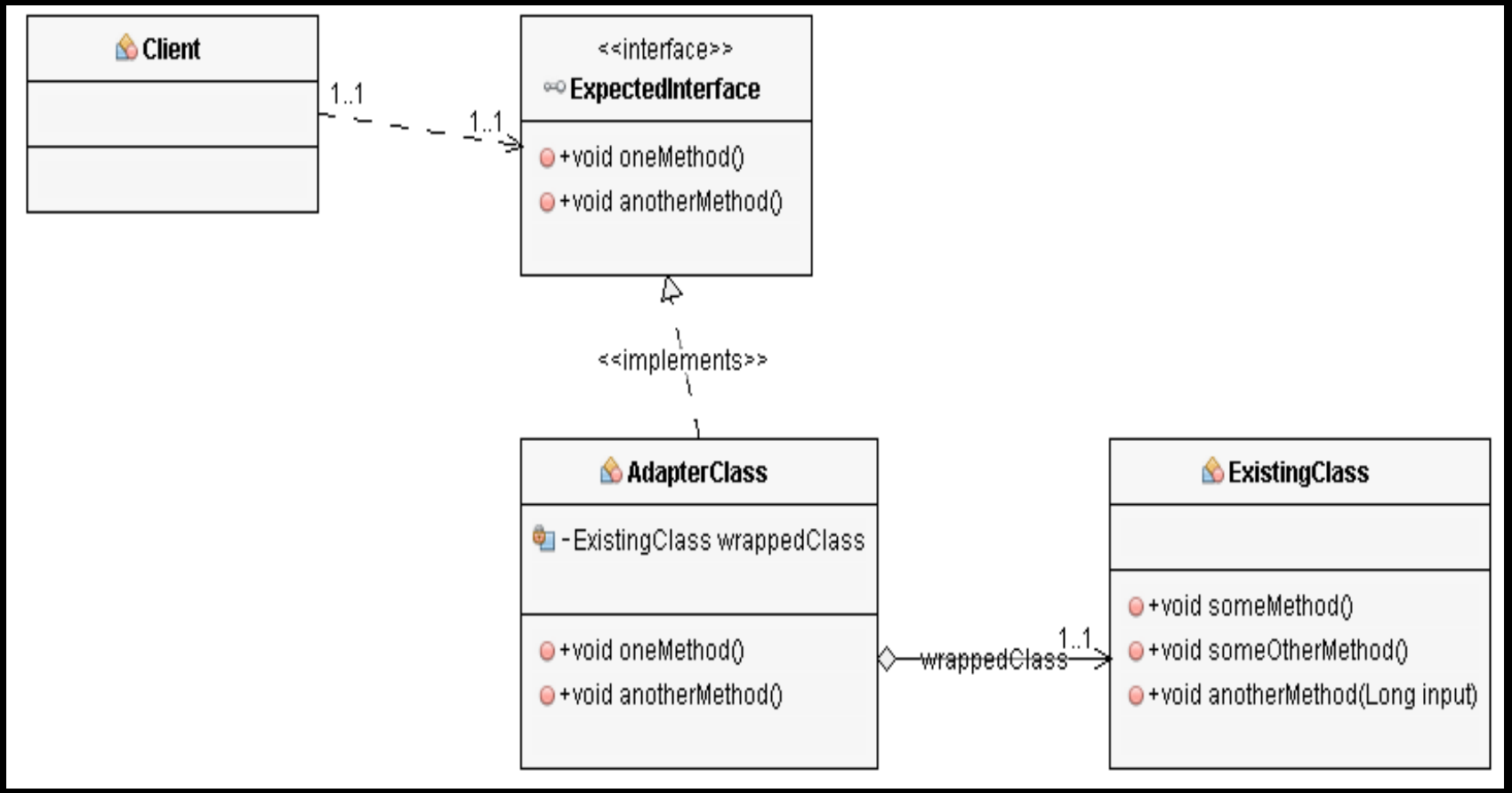
The Adapter implements the interface your class expect.

Adapter talks the vendor interface to service your request.

Goal of Adapter Pattern

- Keeping the client code intact we need to write a new class which will make use of services offered by the class.
- Convert the services offered by class to the client in such a way that functionality will not be changed and the class will become reusable as shown in next slide.

Object Model Adapter Pattern



Example

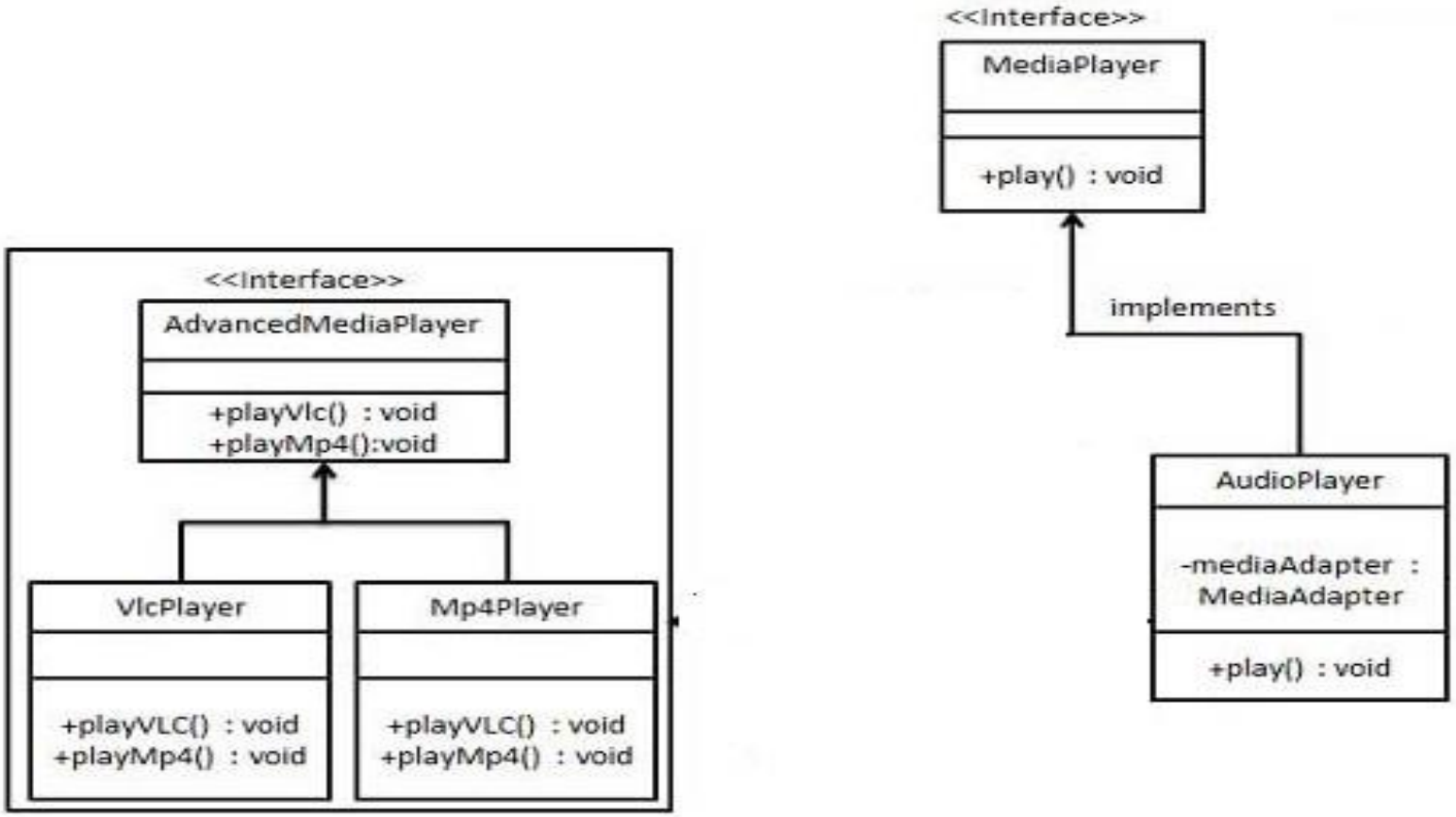
- We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default.

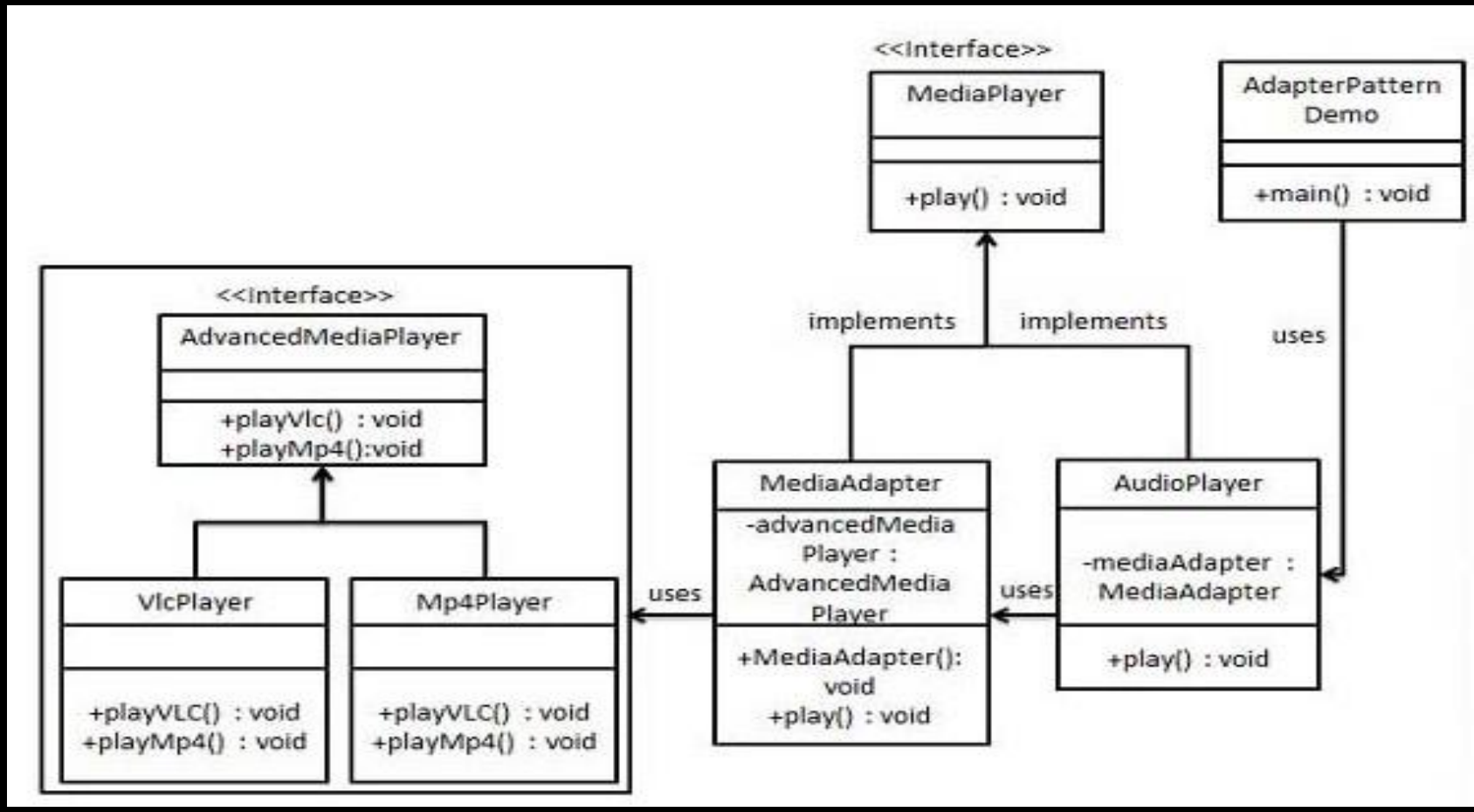
- We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files.

- We want to make AudioPlayer to play other formats as well. To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format.

- AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will use AudioPlayer class to play various formats.

formats.


<<Interface>>
MediaPlayer
+play() : void
implements
AudioPlayer
-mediaAdapter : MediaAdapter
+play() : void
<<Interface>>
AdvancedMediaPlayer
+playVlc() : void
+playMp4():void
VlcPlayer
+playVLC() : void
+playMp4() : void
Mp4Player
+playVLC() : void
+playMp4() : void


<<Interface>>
MediaPlayer
+play() : void
AdapterPattern Demo
+main() : void
<<Interface>>
AdvancedMediaPlayer
+playVlc() : void
+playMp4():void
implements
implements
uses
MediaAdapter
-advancedMedia Player : AdvancedMedia Player
+MediaAdapter(): void
+play() : void
AudioPlayer
-mediaAdapter : MediaAdapter
+play() : void
uses
uses
VlcPlayer
+playVLC() : void
+playMp4() : void
Mp4Player
+playVLC() : void
+playMp4() : void

Step 1

- Create interfaces for Media Player and Advanced Media Player.
- *MediaPlayer.java*

```
public interface MediaPlayer {
public  void  play(String  audioType,  String
  fileName); }
```

- *AdvancedMediaPlayer.java*

```
public interface AdvancedMediaPlayer {
public void playVlc(String fileName);
public void playMp4(String fileName); }
```

Step 2

- Create concrete classes implementing the *AdvancedMediaPlayer* interface.
- *VlcPlayer.java*

```
public class VlcPlayer implements
  AdvancedMediaPlayer{
@Override
public void playVlc(String fileName) {
System.out.println("Playing vlc file. Name: "+
  fileName);
}
@Override
public void playMp4(String fileName) {
//do nothing
        }
}
```

Step 2

- Create concrete classes implementing the *AdvancedMediaPlayer interface.*
- *Mp4Player.java*

```
public class Mp4Player implements
  AdvancedMediaPlayer{
@Override
public void playVlc(String fileName) {
//do nothing
}
@Override
public void playMp4(String fileName) {
System.out.println("Playing mp4 file. Name: "+
  fileName);
}
}
```

Step 3

Create adapter class implementing the *MediaPlayer interface. (MediaAdapter.java)*

```
public class MediaAdapter implements MediaPlayer {
AdvancedMediaPlayer advancedMusicPlayer;
public MediaAdapter(String audioType){
if(audioType.equalsIgnoreCase("vlc") ){
advancedMusicPlayer = new VlcPlayer();
} else if (audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer = new Mp4Player();
} }
@Override
public void play(String audioType, String fileName) {
if(audioType.equalsIgnoreCase("vlc")){
advancedMusicPlayer.playVlc(fileName);
} else if(audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer.playMp4(fileName);
   }
            }
}
```

Step 4

- Create concrete class implementing the *MediaPlayer interface.*
- *AudioPlayer.java*

```
public class AudioPlayer implements MediaPlayer {
MediaAdapter mediaAdapter;
@Override
public void play(String audioType, String fileName) {
//inbuilt support to play mp3 music files
if(audioType.equalsIgnoreCase("mp3")){
System.out.println("Playing mp3 file. Name: " + fileName);
}
//mediaAdapter is providing support to play other file formats
else if(audioType.equalsIgnoreCase("vlc") ||
audioType.equalsIgnoreCase("mp4")){
mediaAdapter = new MediaAdapter(audioType);
mediaAdapter.play(audioType, fileName);
} else{
System.out.println("Invalid media. " + audioType + " format not
supported");
   }
  }}
```

Step 5

- Use the AudioPlayer to play different types of audio formats.
- *AdapterPatternDemo.java*

```
public class AdapterPatternDemo {
public static void main(String[] args) {
AudioPlayer audioPlayer = new AudioPlayer();
audioPlayer.play("mp3", "beyond the
  horizon.mp3");
audioPlayer.play("mp4", "alone.mp4");
audioPlayer.play("vlc", "far far away.vlc");
audioPlayer.play("avi", "mind me.avi");
  }
}
```

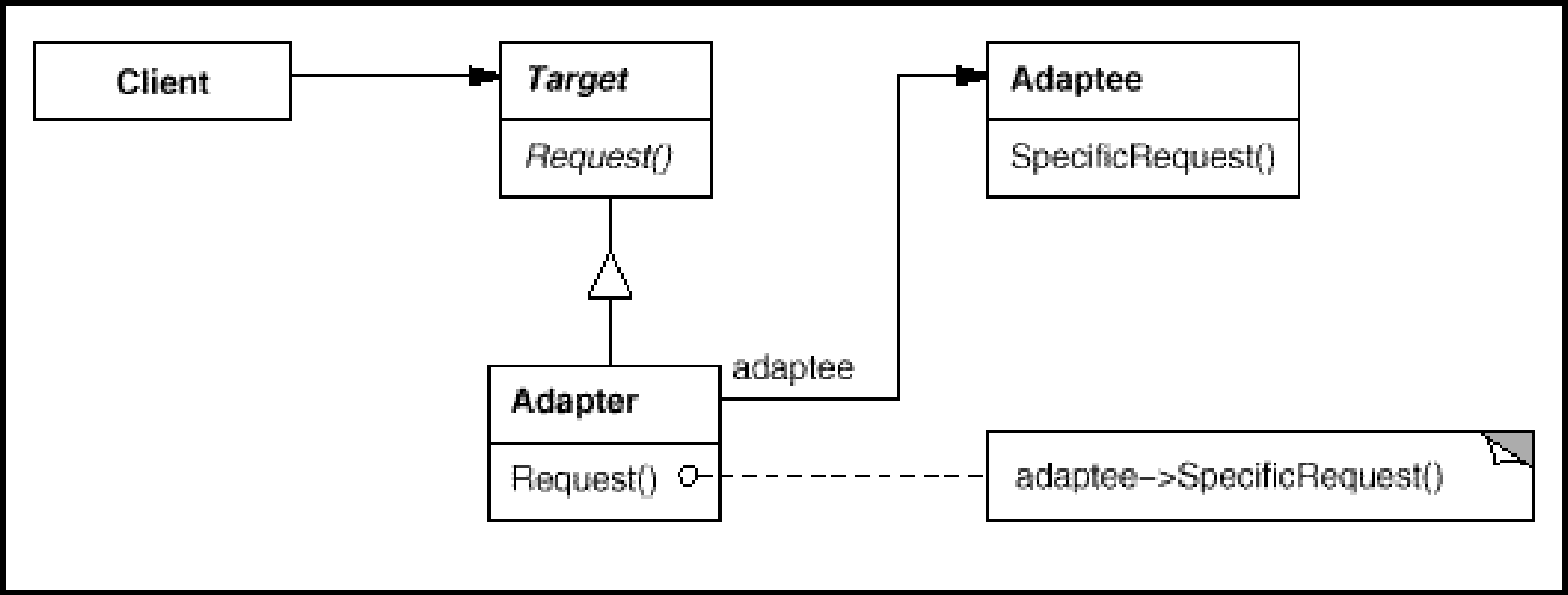
Step 6

- Verify the output.

  Playing mp3 file. Name: beyond the horizon.mp3
  Playing mp4 file. Name: alone.mp4
  Playing vlc file. Name: far far away.vlc
  Invalid media. avi format not supported

UML Class Diagram for Adapter pattern

- Clients needs a target that implements one interface



Adapter *Example*

Client Code:

```
Adaptee a = new Adaptee();   Target t = new Adapter(a);
 public void test() {  t.request();  }
```

Target Code:

```
class Target {
     public void request() {}
}
```

Adaptee Code:

```
class Adaptee {
   public void specificRequest() {
     System.out.println("Adaptee:
SpecificRequest");
}
```

Adapter Code:

```
class Adapter extends Target {
  private Adaptee adaptee;
  public Adapter(Adaptee a) { adaptee = a;}
  public void request() { adaptee.specificRequest();}
}
```

Flow of Events in Adapter Pattern

- Client call operations on Adaptor instance, which in return call adaptee opertions that carry out the request.
- To use an adapter:
  - The client makes a request to the adapter by calling a method on it using the target interface.
  - The adapter translates that request on the adaptee using the adaptee interface.
  - Client receive the results of the call and is unaware of adapter's presence.

Components of Adapter Class

1. Adaptee: Defines an existing interface that needs adapting; it represents the component with which the client wants to interact with the.
2. Target: Defines the domain-specific interface that the client uses; it basically represents the interface of the adapter that helps the client interact with the adaptee.
3. Adapter: Adapts the interface Adaptee to the Target interface; in other words, it implements the Target interface, defined above and connects the adaptee, with the client, using the target interface implementation
4. Client: The main client that wants to get the operation, is done from the Adaptee.

Consequences

- Class Adapter
  - Adapts Adaptee to Target by committing to a concrete Adapter class.
  - Lets Adapter override some of the Adaptee's behavior by subclassing.
  - Introduces only one object and no additional pointer indirection is needed to get the adaptee.
- Object Adapter
  - Lets a single adapter work with a group of adaptees such as a base class and all its sub classes.
  - The adapter can add functioanlity to all adaptees at once.
  - Makes it harder to override Adaptee behavior as the Adapter may not know with what Adaptee it is working with.

Class Adapter Vs Object Adapter

- **Class Adapter**
  - uses *inheritance* and can only wrap a **class**. It cannot wrap an interface since by definition it must derive from some base class.

- **Object Adapter**
  - uses *composition* and can wrap classes or interfaces, or both. It can do this since it contains, as a private, encapsulated member, the class or interface **object instance** it wraps.

Limitations of Adapter Design Pattern

- Due to adapter class the changes are encapsulated within it and client is decoupled from the changes in the class.

When to use Adapter Pattern

- When you have a third-party code that cannot interact with the client code. For example, you might want to use a third-party logger service, but your code is having incompatibility issues, you can use this pattern.
- When you want to use an existing code with extended functionality but not without changing it, as it is being used in other components, you can extend it using the adapter pattern.
- Again you can use an object adapter for a code, which is using sealed class components, or needs multiple inheritance. For a requirement where you need to use single inheritance, you can choose a class adapter.

Facade Pattern
Type: Structural

Intent

- A facade or façade is generally one exterior side of a building, usually, but not always, the front. The word comes from the French language, literally meaning “frontage” or “face”.

- 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.

- Wrap a complicated subsystem with a simpler interface.

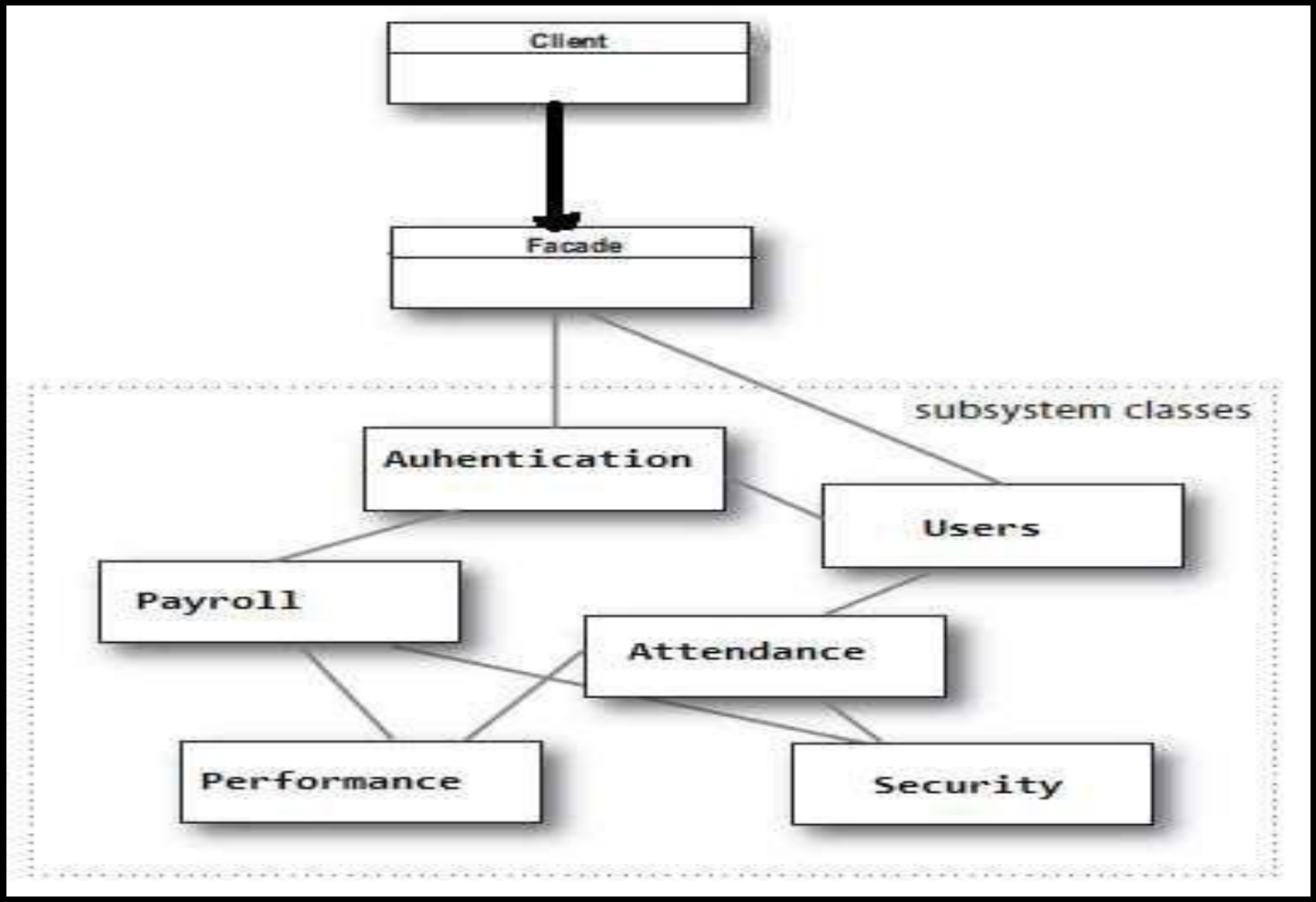
Problem/Solution pa - Facade

- What problems can the Facade design pattern solve?
  - To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem.
  - The dependencies on a subsystem should be minimized.
  - Clients that access a complex subsystem directly refer to (depend on) many different objects having different interfaces (tight coupling), which makes the clients hard to implement, change, test, and reuse.

Problem/Solution pa – Façade (Continued)

- What solution does the Facade design pattern describe?
  - Define a Facade object that implements a simple interface in terms of (by delegating to) the interfaces in the subsystem and may perform additional functionality before/after forwarding a request.
  - This enables to work through a Facade object to minimize the dependencies on a subsystem.

Another Façade Example

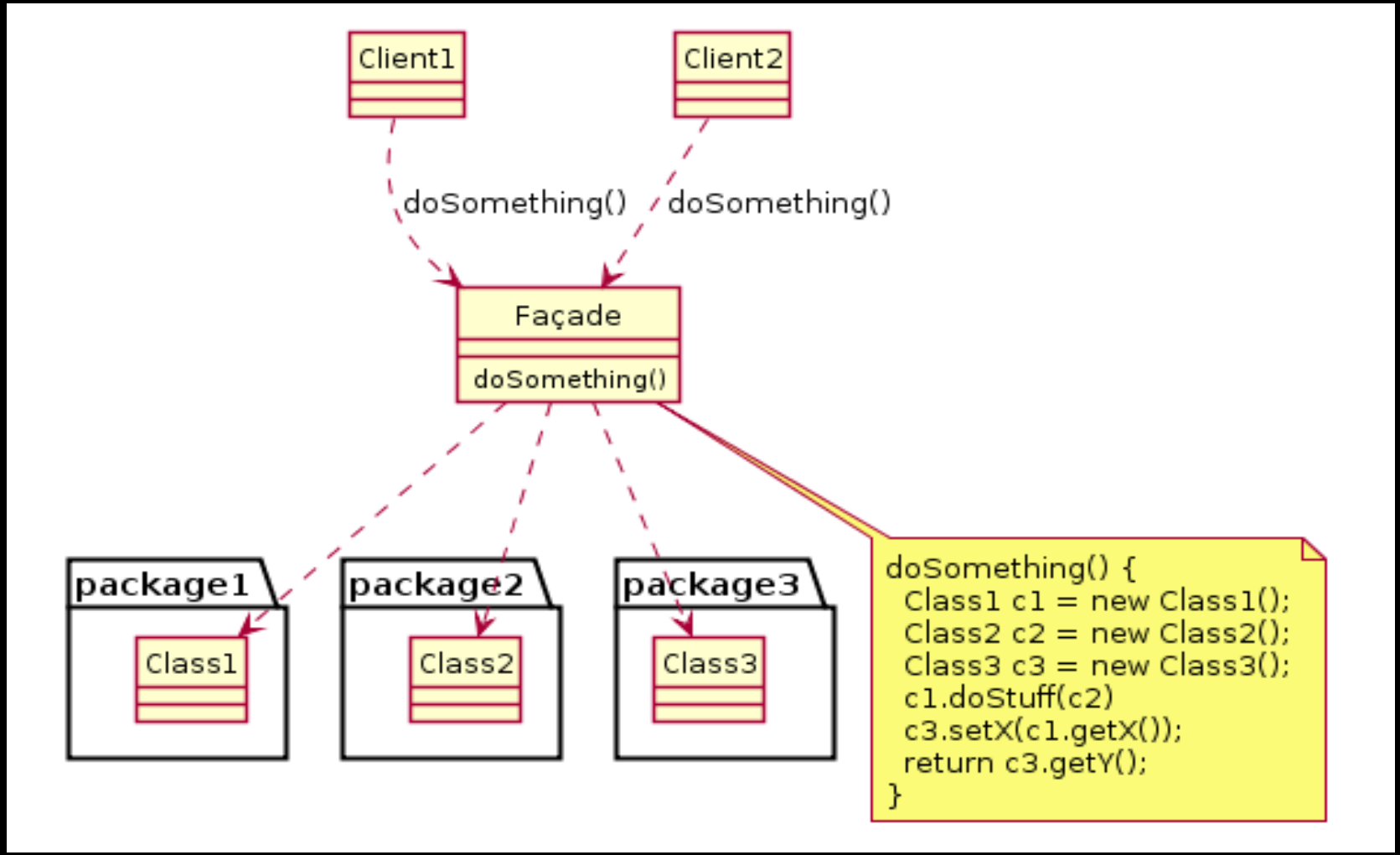


Participants

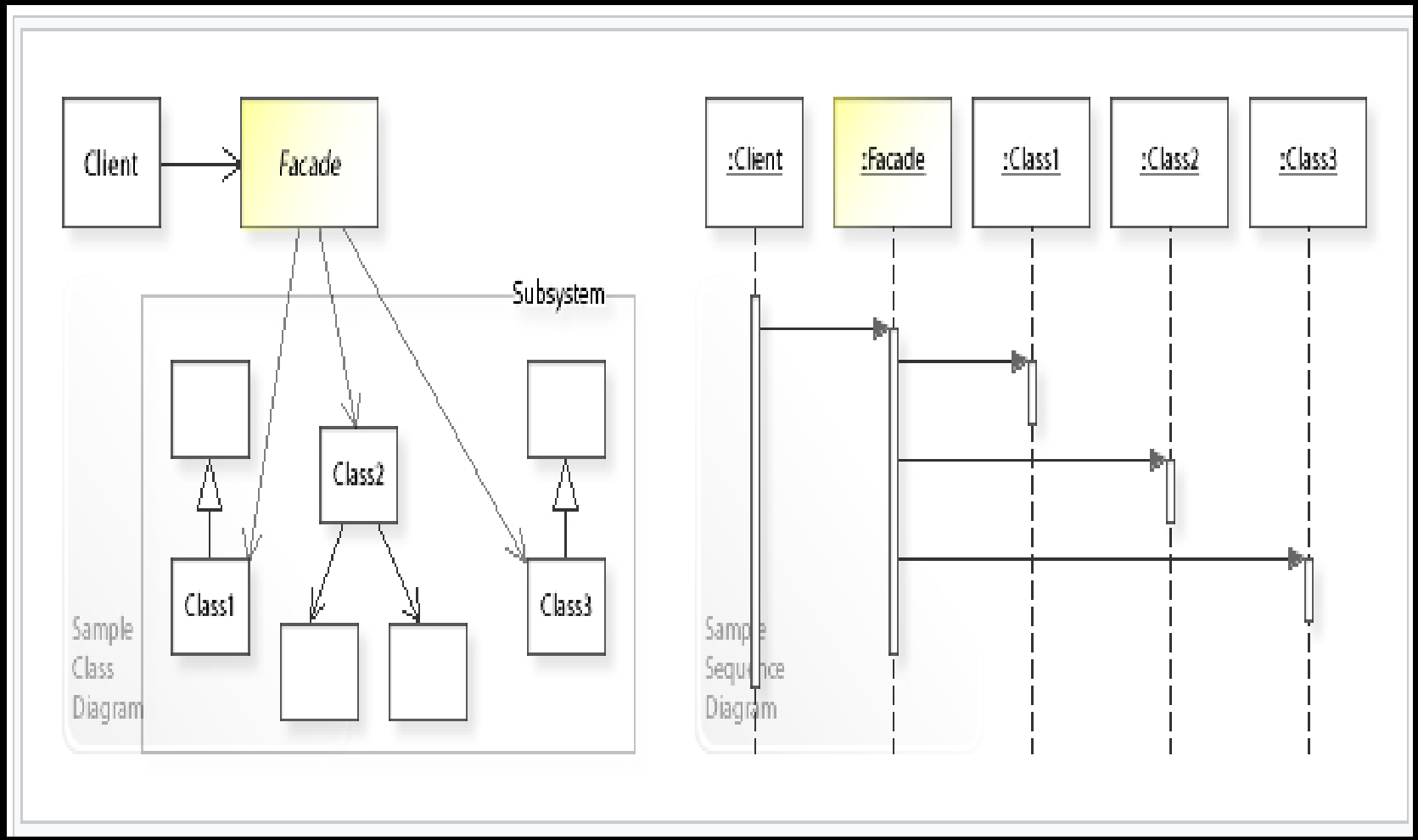
- **Facade**:
- Knows which subsystem classes are responsible for a request. Delegates client requests to appropriate subsystem objects.

- **Subsystem classes**:
- Implements subsystem functionality.
- Handle work assigned by Facade object.
- Have no knowledge of the facade; that is, they keep no references to it.

Exampl Façade



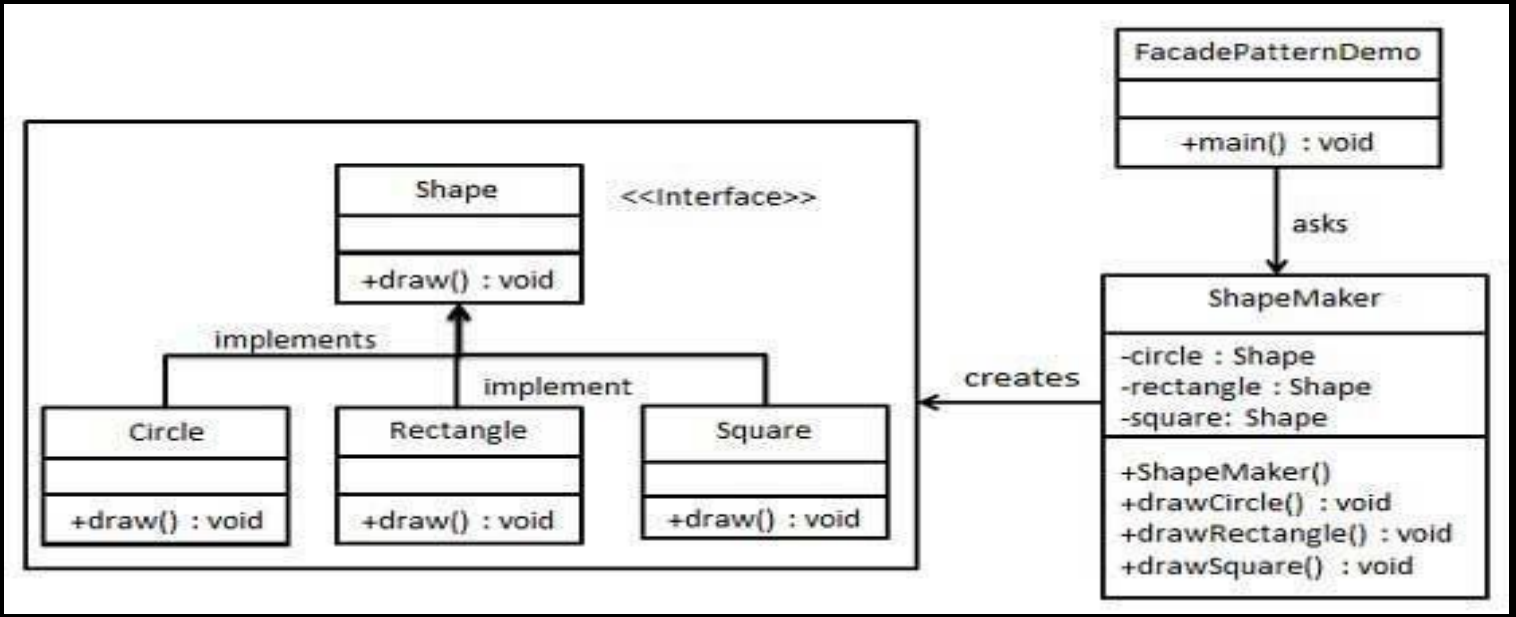
Exampl Facade



Code Example for Facade Pattern

Façade Example

- 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()
```

Façade Design 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.

That is all