Software Design and Architecture

SE-2002

Patterns

Rubab Jaffar
rubab.jaffar@nu.edu.pk

Today's Outline

- Patterns
- Pattern types
  - Design
  - Architecture
  - Coding
- An introduction to design patterns
- Why need design patterns
- Evolution of design patterns
- 3 types of design patterns
  - Creational
  - Structural
  - Behavorial
- Singleton Pattern
- Factory Pattern

Patterns

A pattern can be defined as a sequence of repeating objects, shapes or numbers. We can relate a pattern to any type of event or object. A pattern has a rule that tells us which objects belong to the pattern and which objects do not belong to the pattern.

Design Patterns:

A design pattern provides a scheme for refining the subsystems or components of a software system, or the relation ships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

Architectural Patterns:

An architectural Pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Coding patterns:

A coding pattern is a recurring techniques that provide a structured approach to solving complex problems. Think of them as the building blocks of algorithms, helping you to break down problems into more manageable parts.

Definitions

- A *pattern* is a recurring solution to a standard problem, in a context.
- Christopher Alexander, a professor of architecture…
  - *Why would what a prof of architecture says be relevant to software?*
  - "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Patterns in Engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems.
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - *Should software engineers make use of patterns? Why?*
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture and design

What is a Design Pattern?

- Design pattern is a general reusable solution to a commonly occurring problem in software design.

  A design pattern is not 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.
- Helps the designer in getting to the right design faster

The "gang of four (GoF)

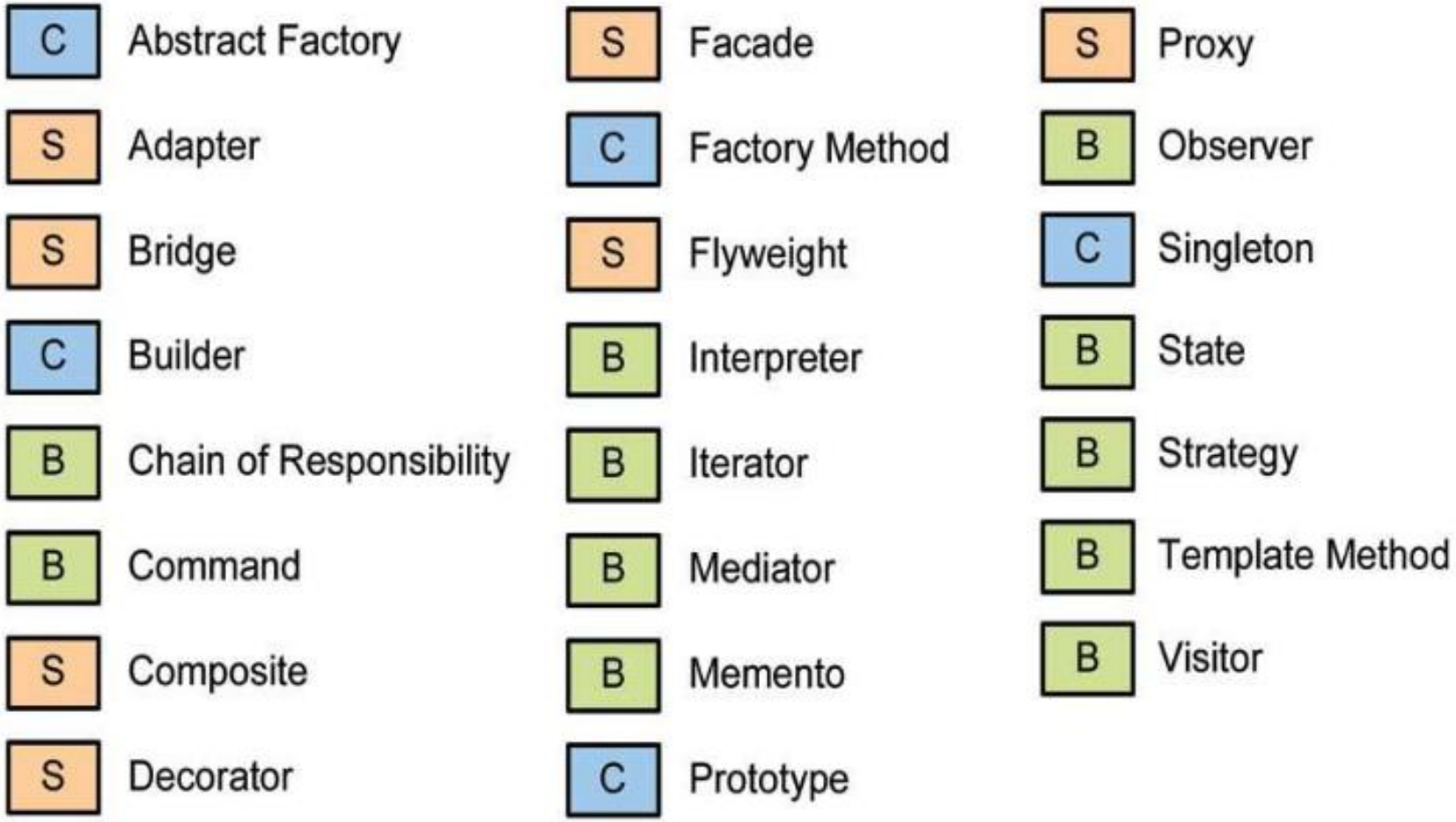
- Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995)
  - *Design Patterns* book catalogs 23 different patterns as solutions to different classes of problems, in C++ & Smalltalk
  - The problems and solutions are broadly applicable, used by many people over many years
    - Why is it useful to learn about this pattern?

      Patterns suggest opportunities for reuse in analysis, design and programming
  - GOF presents each pattern in a structured format

Evolution of Design Patterns an GoF

- The four authors of the book "Design Patterns: Elements of Reusable Object-Oriented Software" are referred to as the "Gang of Four".
- The book consists of two parts:
  - First part contains the pros & cons of OOP whereas
  - The second part consists of 23 design patterns.

THE 23 GANG OF FOUR DESIGN PATTERNS



Types of Design Patterns

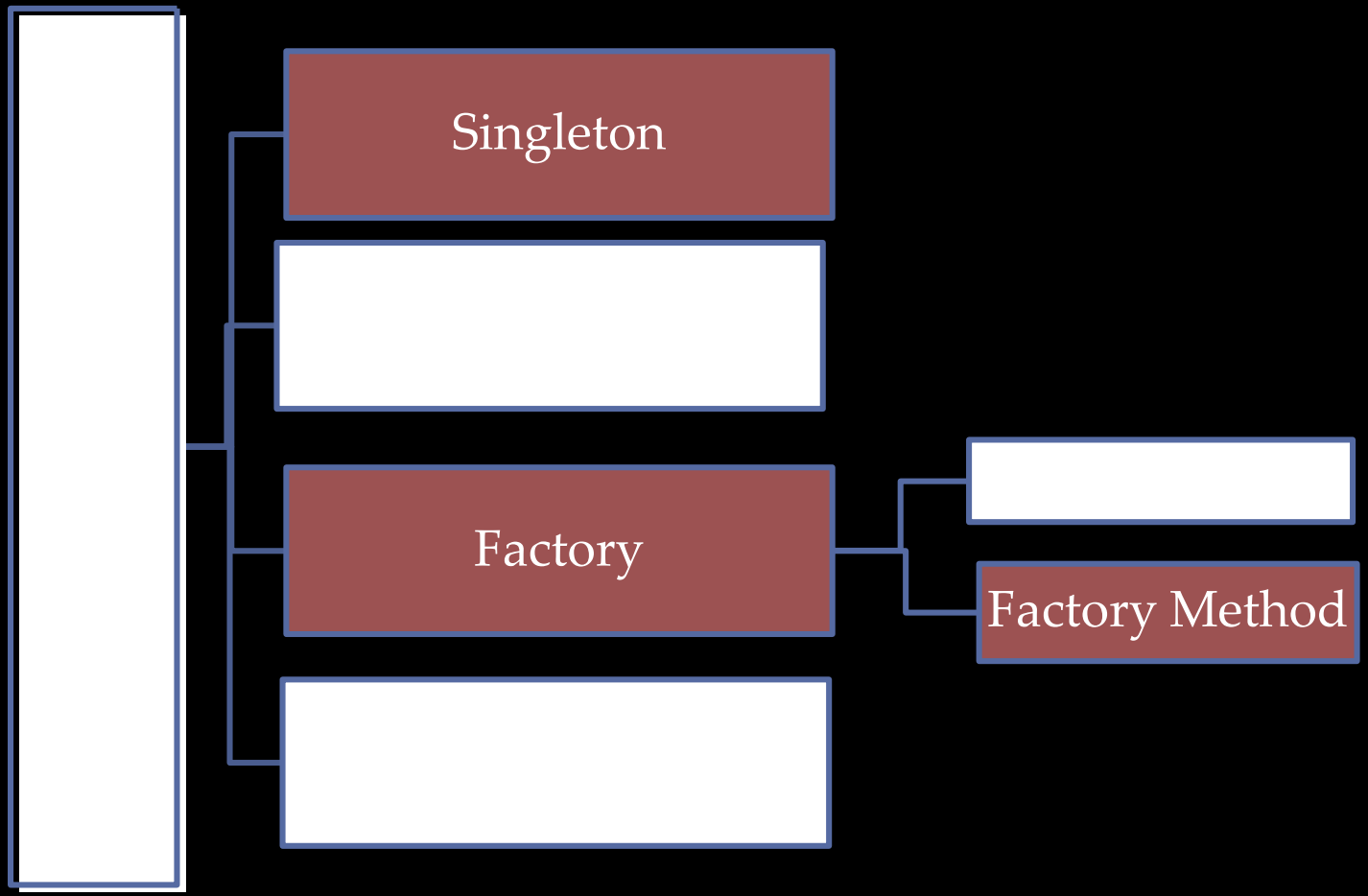
- There are three basic kinds of design patterns:
  - **Structural**
  - **Creational**
  - **Behavioral**

Creational Design Pattern

- **Creational** patterns deals with the object creation and initialization while hiding the creation logic
- It gives the program more flexibility in deciding which objects need to be created for a given case.

- E.g. Singleton, Factory, Abstract Factory

Creational Patterns



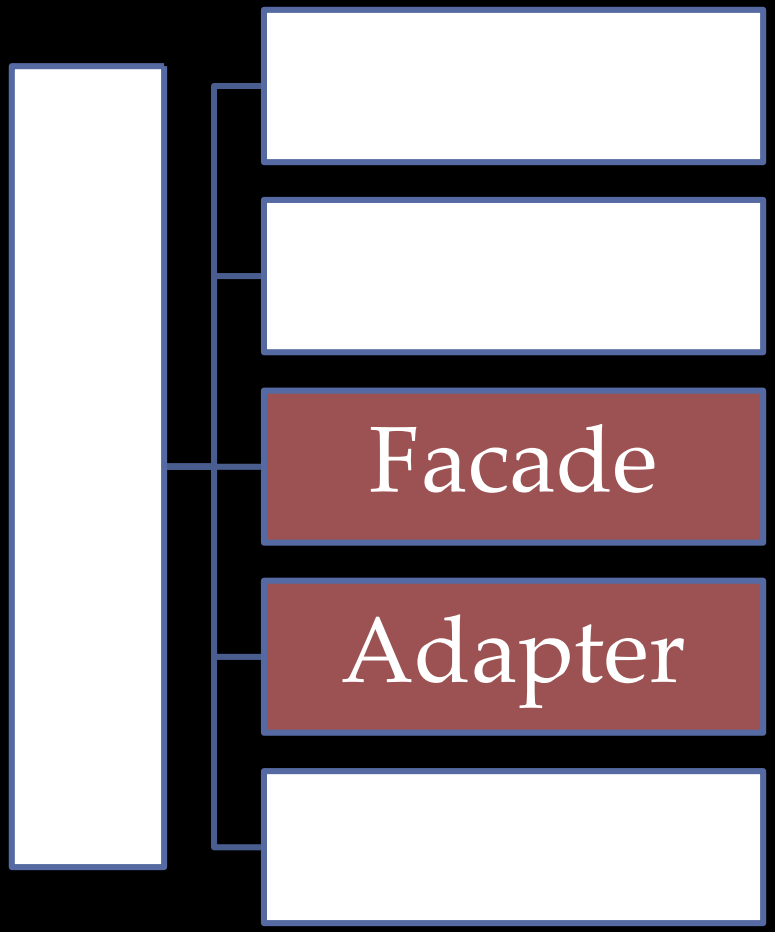
Structural Patterns

- How objects/classes can be combined to form larger structures
- **Structural** patterns generally deal with relationships between entities, making it easier for these entities to work together.

- 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.

- E.g. Adapter, Bridge, façade e.t.c.

Structural Patterns

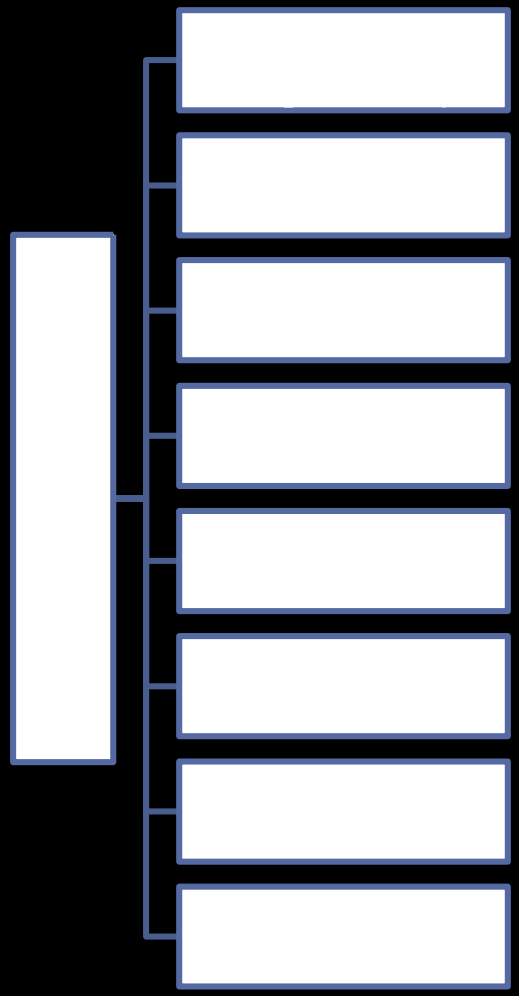


Behavior patterns

- **Behavioral** patterns are used in communications between entities and make it easier and more flexible for these entities to communicate.

- E.g. Chain of Responsibility, command, Interpreter etc..

Behavorial Patterns



Types of design patterns

CREATIONAL

- how objects can be created
  - maintainability
  - control
  - extensibility

STRUCTURAL

- how to form larger structures
  - management of complexity
  - efficiency

BEHAVIOURAL

- how responsibilities can be assigned to objects
  - objects decoupling
  - flexibility
  - better communication

Elements of Design Patterns

The pattern's name	The problem	The solution	The consequences
The name of the pattern is a one or two word description that pattern-literate programmers familiar with patterns can use to communicate with each other. Examples of names include "factory method" "singleton", "mediator", "prototype", and many more. The name of the pattern should recall the problem it solves and the solution.	The problem the pattern solves includes a general intent and a more specific motivation or two. For instance, the intent of the singleton pattern is to prevent more than one instance of a class from being created. A motivating example might be to not allow more than one object to try to access a system's audio hardware at the same time by only allowing a single audio object.	The solution to the problem specifies the elements that make up the pattern such as the specific classes, methods, interfaces, data structures and algorithms. The solution also includes the relationships, responsibilities and collaborators of the different elements. Indeed these inter-relationships and structure are generally more important to the pattern than the individual pieces, which may change without significantly changing the pattern.	Often more than one pattern can solve a problem. Thus the determining factor is often the consequences of the pattern. Some patterns take up more space. Some take up more time. Some patterns are more scalable than others.

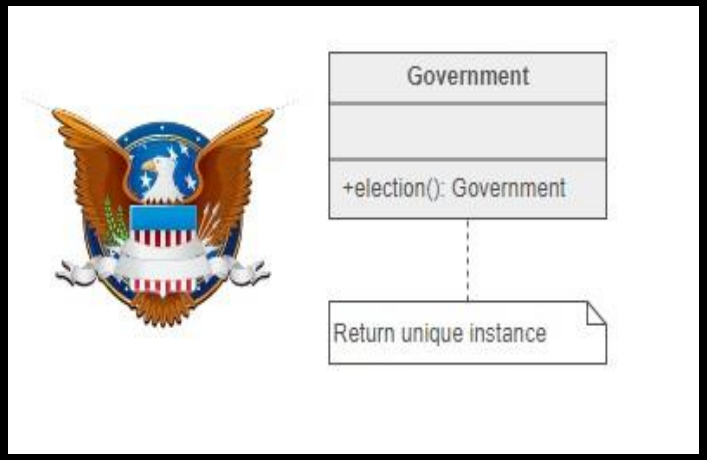
Why need design patterns?

- **Patterns provide object-oriented software developers with:**
  - Reusable solutions to common problems.
  - Names of abstractions above the class and object level.
  - Use the experiences of software developers.
  - A shared library/lingo used by developers.
  - "Design patterns help a designer get a design right faster".

Singleton Pattern
Type: Creational

Singleton Design Pattern

- **Singleton pattern** restricts the instantiation of a class and ensures that only one instance of the class exists.
- Example:
  - President of the US/ Government
  - Java.lang.System
- Encapsulated "just-in-time initialization" or "initialization on first use".



Problem/Solution pa - Singleton

- The Singleton design pattern solves problems like:
  - How can it be ensured that a class has only one instance?
  - How can the sole instance of a class be accessed easily?
  - How can a class control its instantiation?
  - How can the number of instances of a class be restricted?

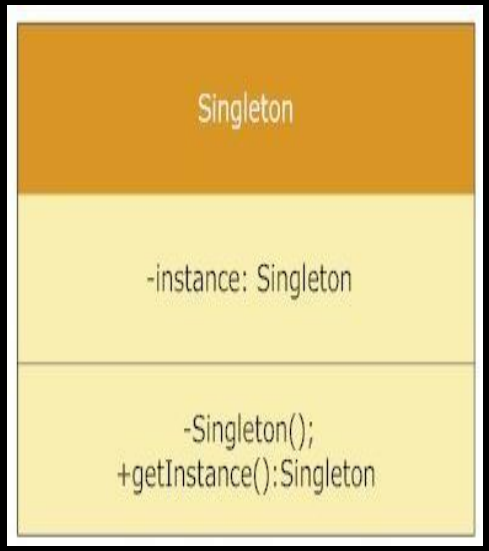
- The Singleton design pattern describes how to solve such problems:
  - Hide the constructor of the class.
  - Define a public static operation (getInstance()) that returns the sole instance of the class.

Implementation Details

•Static member : This contains the instance of the singleton class.

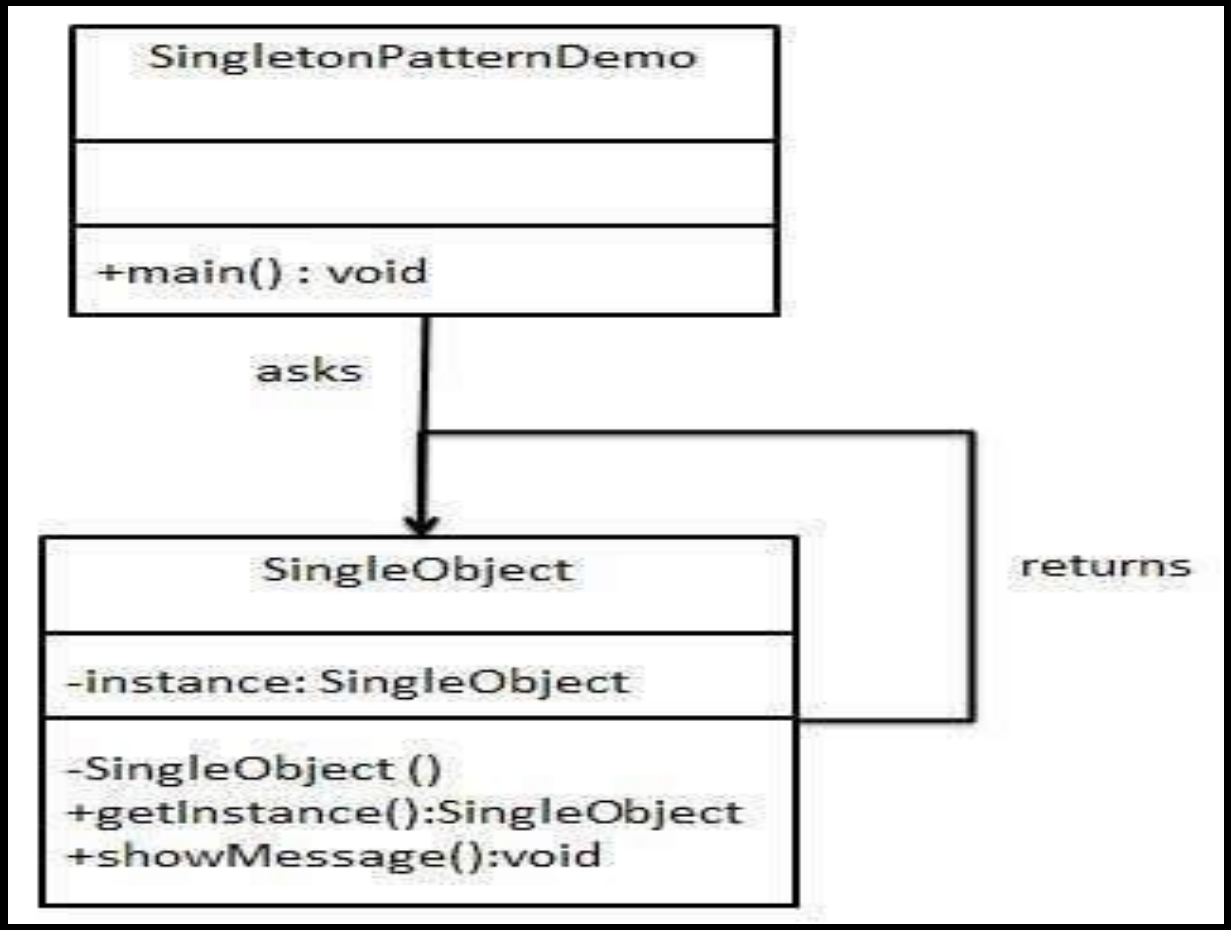
•Private constructor : This will prevent anybody else to instantiate the Singleton class.

•Static public method : This provides the global point of access to the Singleton object and returns the instance to the client calling class.



- **Singleton pattern** is used for logging, drivers objects, caching and thread.

Object Model for Singleton



Method 1: Classic Implementation

```
// Classical Java implementation of singleton
// design pattern
class Singleton
{
        private static Singleton obj;

        // private constructor to force use of
        // getInstance() to create Singleton object
        private Singleton() {}

        public static Singleton getInstance()
        {
                if (obj==null)
                        obj = new Singleton();
                return obj;
        }
}
```

The main problem with above method is that it is not thread safe. Consider the following execution sequence.

Method 2: Make getInstance() synchronized

```
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton
{
        private static Singleton obj;

        private Singleton() {}

        // Only one thread can execute this at a time
        public static synchronized Singleton getInstance()
        {
                if (obj==null)
                        obj = new Singleton();
                return obj;
        }
}
```

Method 3: Eager Instantiation

```
// Static initializer based Java implementation of
// singleton design pattern
class Singleton
{
        private static Singleton obj = new Singleton();

        private Singleton() {}

        public static Singleton getInstance()
        {
                return obj;
        }
}
```

Method 4 (Best): Use "Double Checked Locking"

```
// Double Checked Locking based Java implementation of
// singleton design pattern
class Singleton
{
    private volatile static Singleton obj;

    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
        {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj==null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

Java Code Example for Singleton Pattern

```java
public class SingletonExample {

    // Static member holds only one instance of the
    // SingletonExample class
    private static SingletonExample singletonInstance;

    // SingletonExample prevents any other class from instantiating
    private SingletonExample() {
    }

    // Providing Global point of access
    public static SingletonExample getSingletonInstance() {
        if (null == singletonInstance) {
            singletonInstance = new SingletonExample();
        }
        return singletonInstance;
    }

    public void printSingleton(){
        System.out.println("Inside print Singleton");
    }
}
```

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

AKA

- Virtual Constructor
  - The main intent of the virtual constructor idiom in C++ is to create a copy of an object or a new object without knowing its concrete type and this is exactly the Factory Method of initialization.

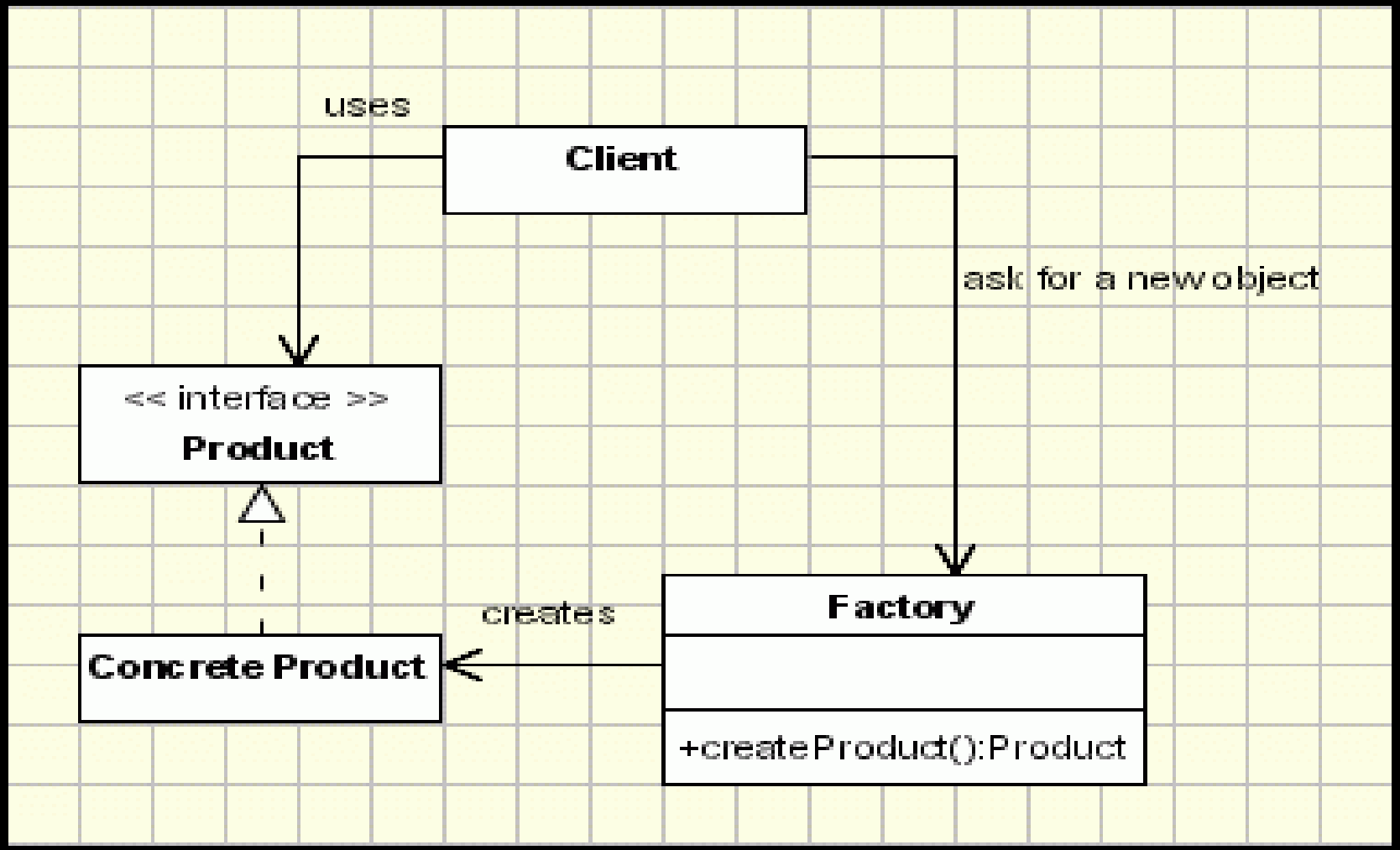
Applicability

- 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

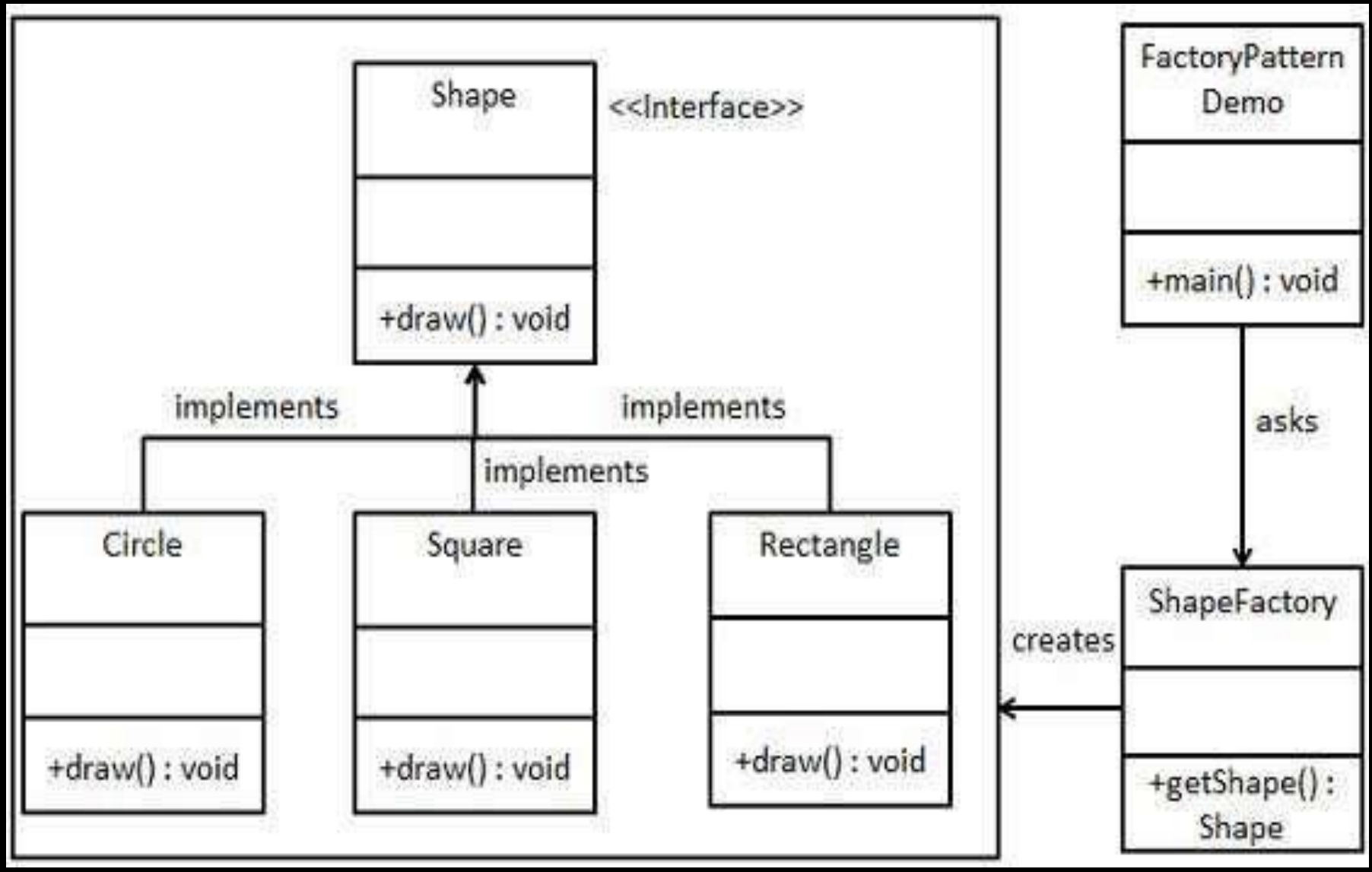


Implementation

- 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.

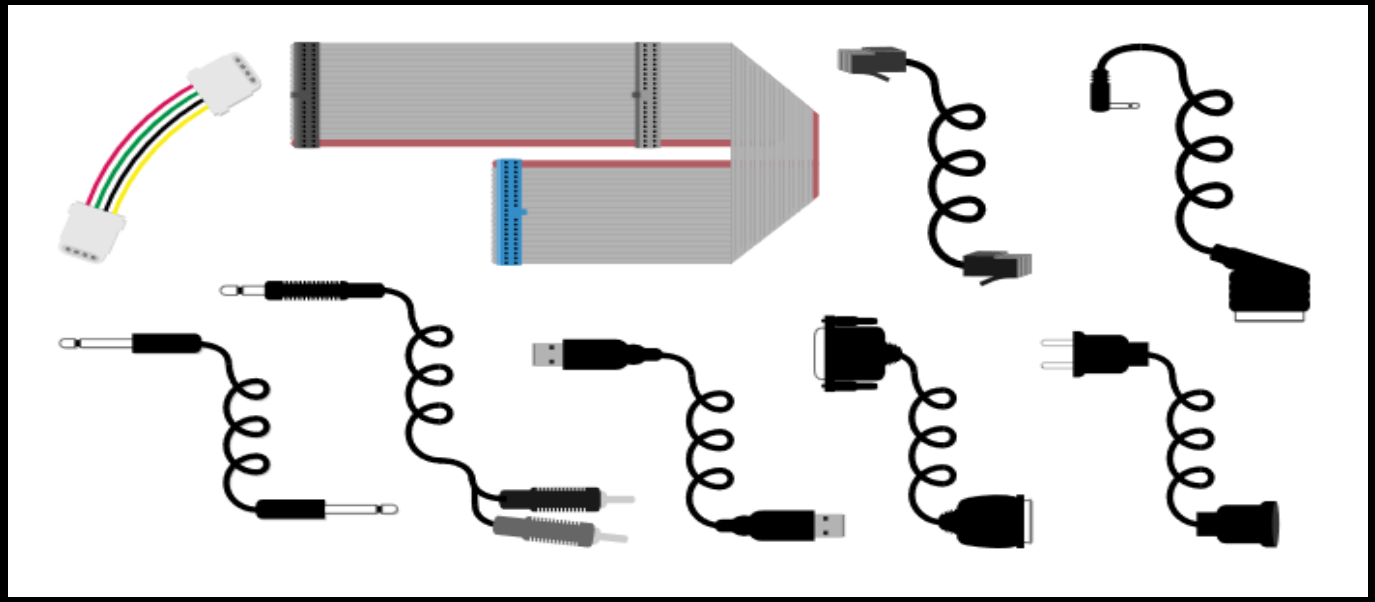
Intent

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.

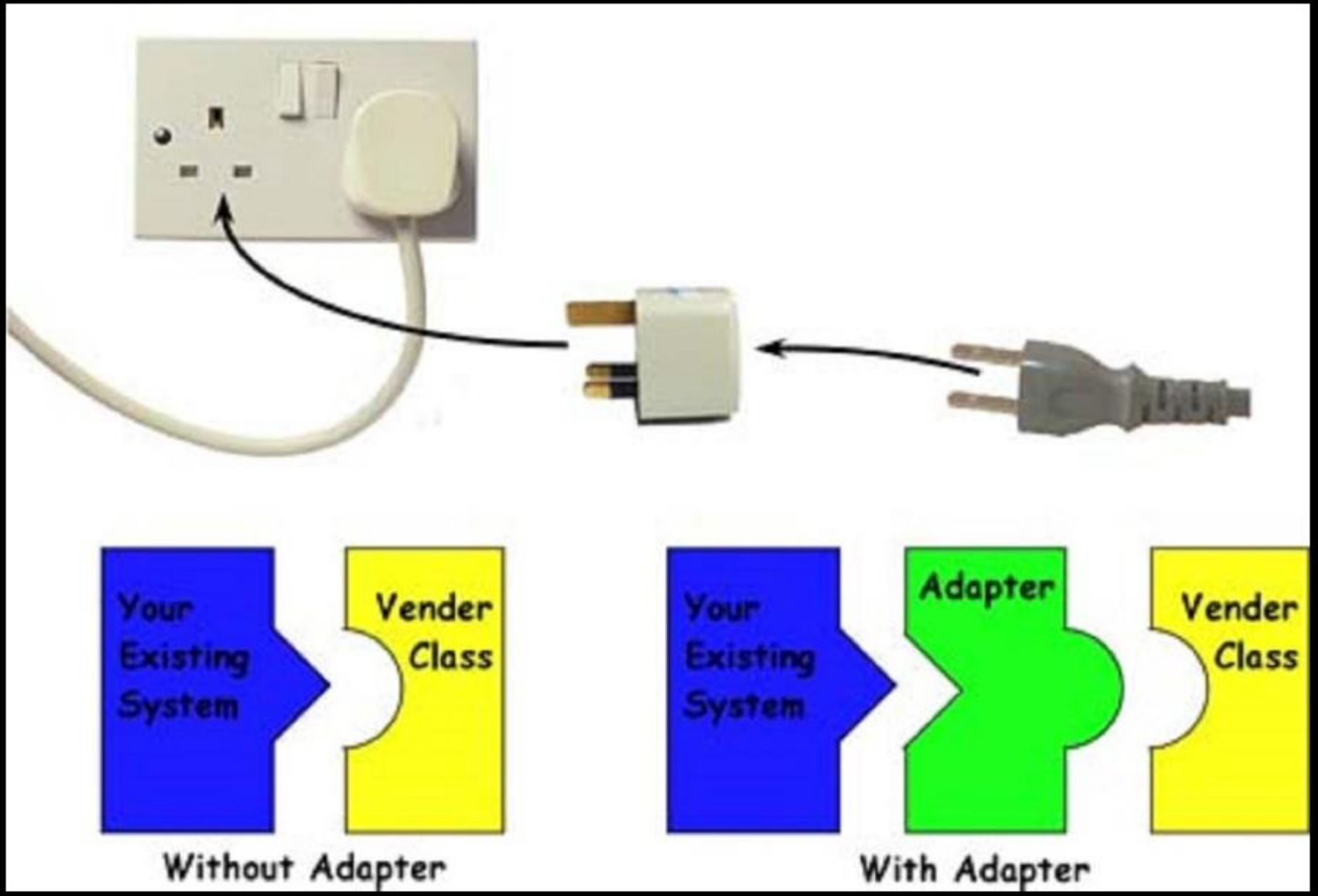


Adapter Motivation

- 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

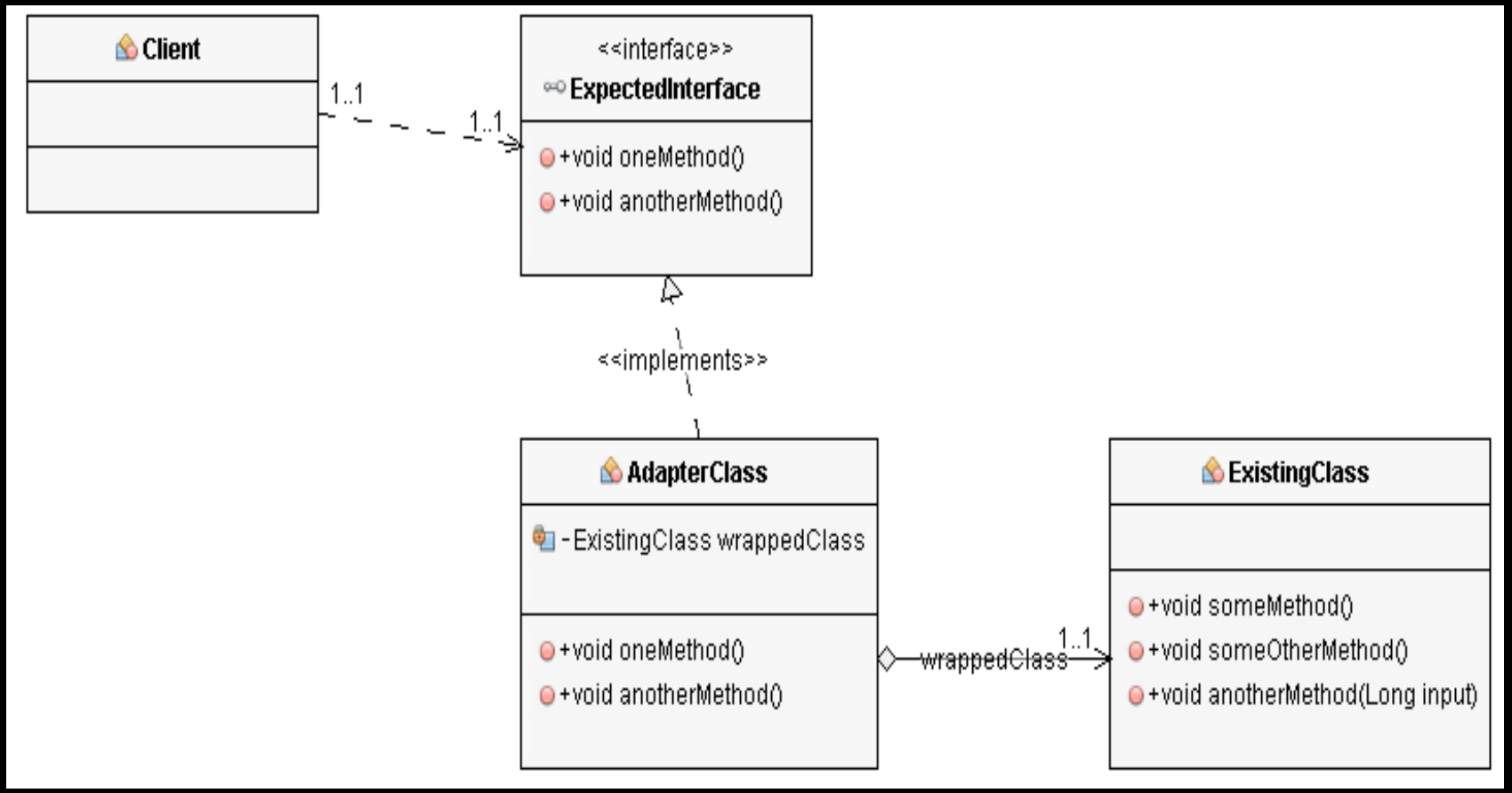
The Adapter implements the interface your class expect.

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

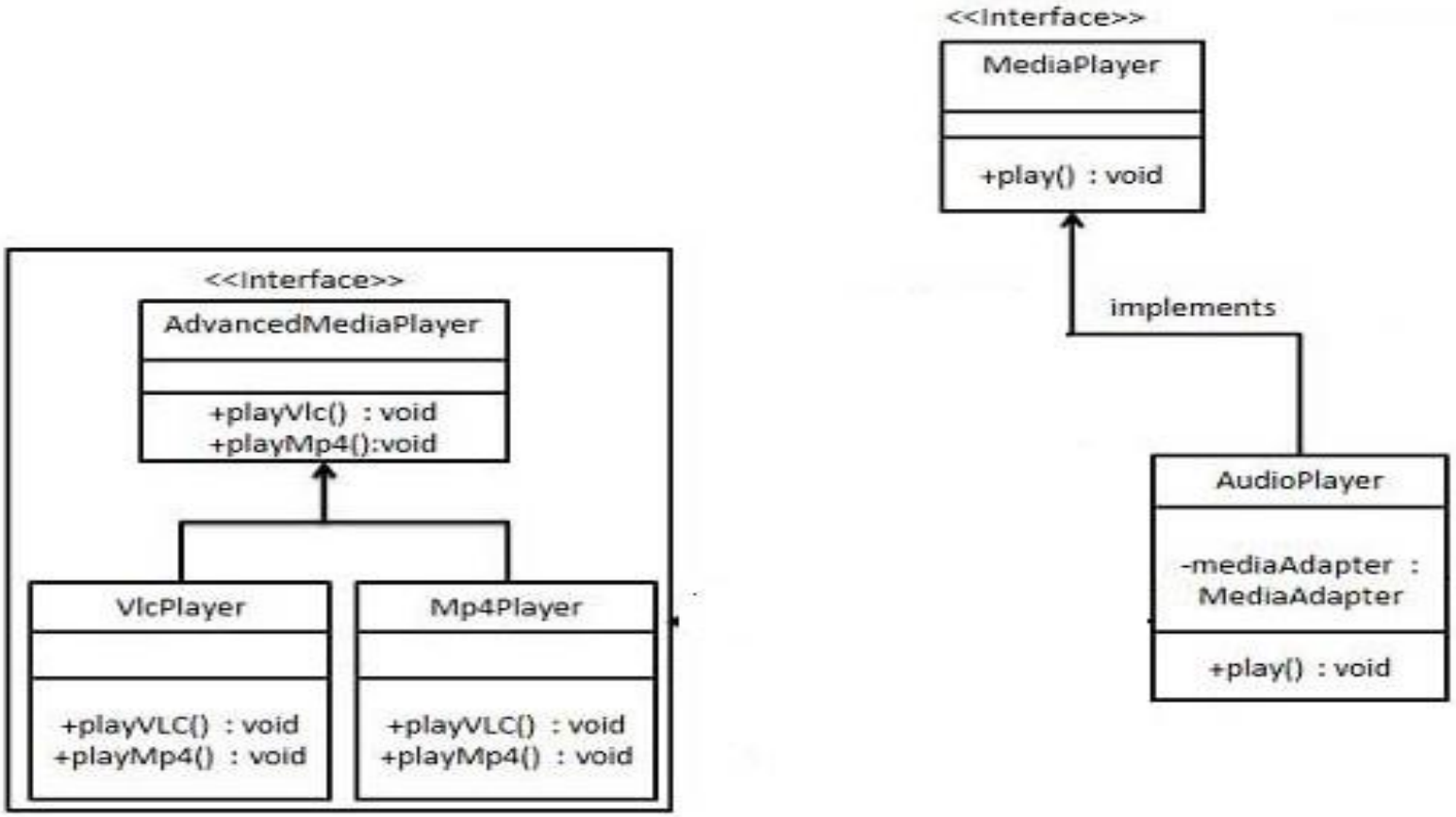
- We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default.

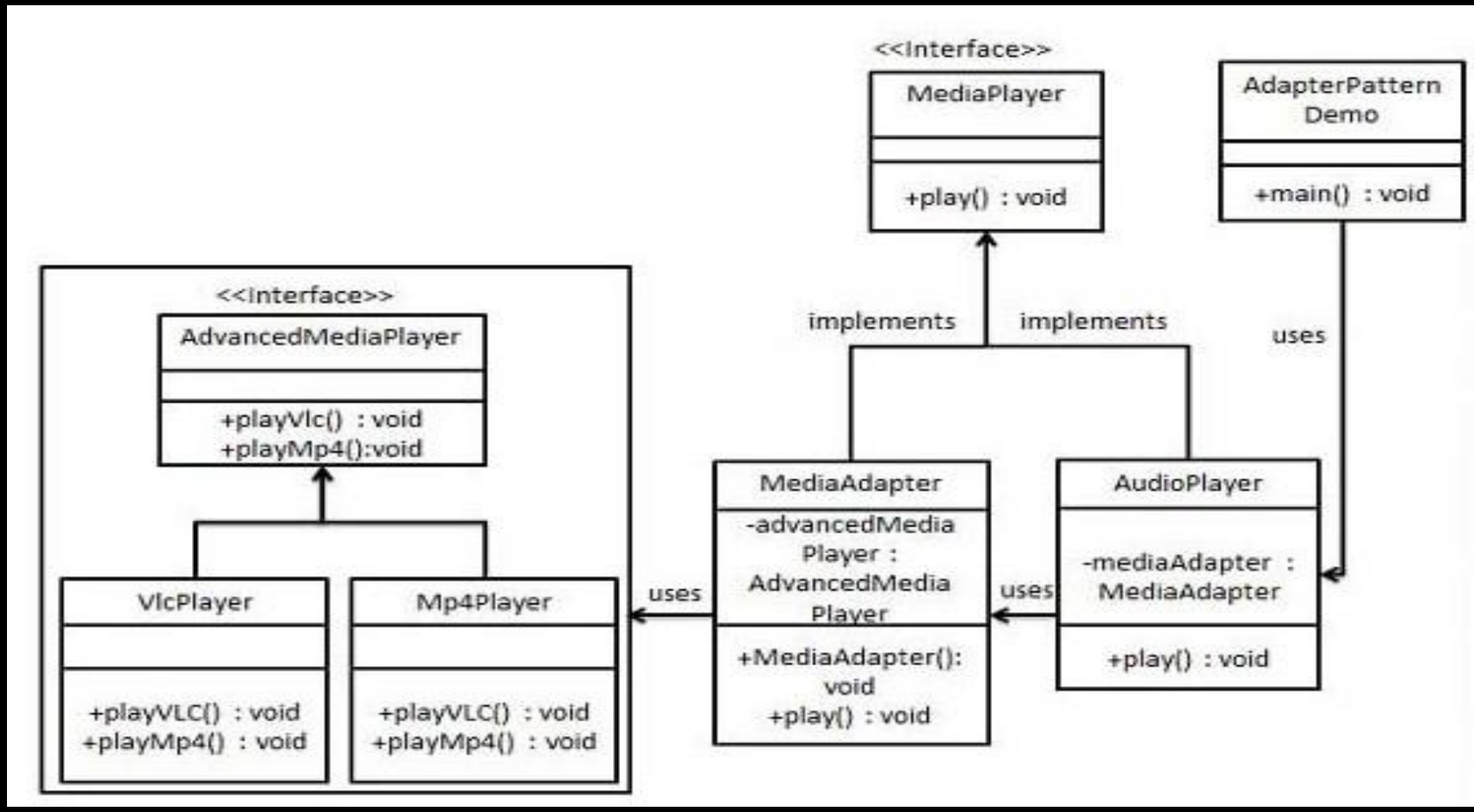
- We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files.

- 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

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.

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.

That is all