
# Design Patterns

## Lecture 13 – Part 02

Jayathma Chathurangani
ejc@ucsc.cmb.ac.lk

# OUTLINE

1. Introduction
2. Creational Patterns
3. Structural Patterns
4. Behavioural Patterns

# TOPIC 03

## Structural Patterns

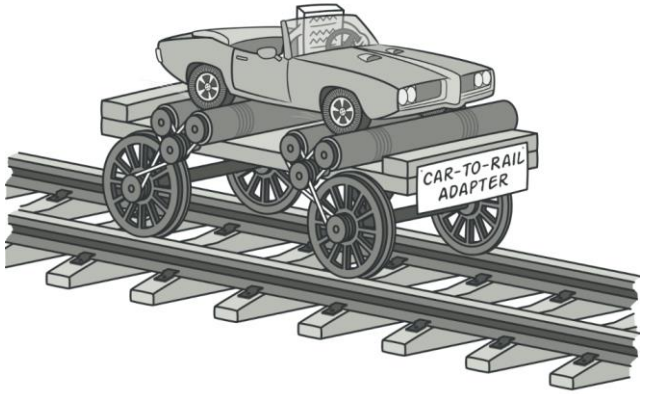
*Structural patterns provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.*

# 3.1 Adaptor Pattern

**Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.**

### ❖ Introduction

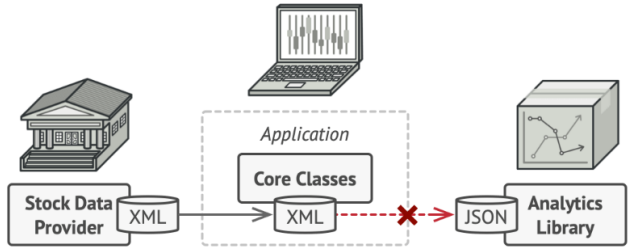
- The adapter design pattern is one of the structural design patterns and it's used so that two unrelated interfaces can work together.
- The object that joins these unrelated interfaces is called an Adapter.
- As a real-life example, we can think of a mobile charger as an adapter because the mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between the mobile charging socket and the wall socket.



# 3.1 Adaptor Pattern (Continued)

## ❖ Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.
- You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.
- You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible..

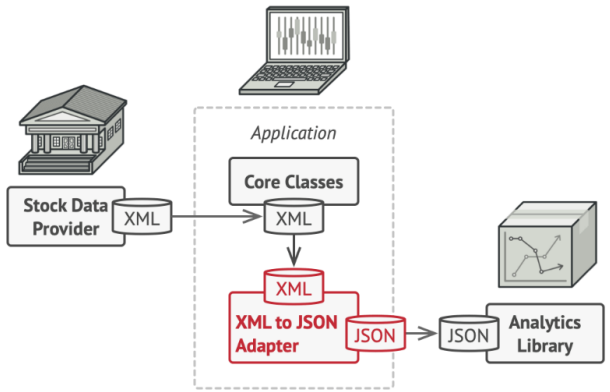



*You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.*

# 3.1 Adaptor Pattern (Continued)

## ❖ Solution

- You can create an adapter. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter.
- Let's get back to our stock market app. To solve the dilemma of incompatible formats, you can create XML-to-JSON adapters for every class of the analytics library that your code works with directly. Then you adjust your code to communicate with the library only via these adapters. When an adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.

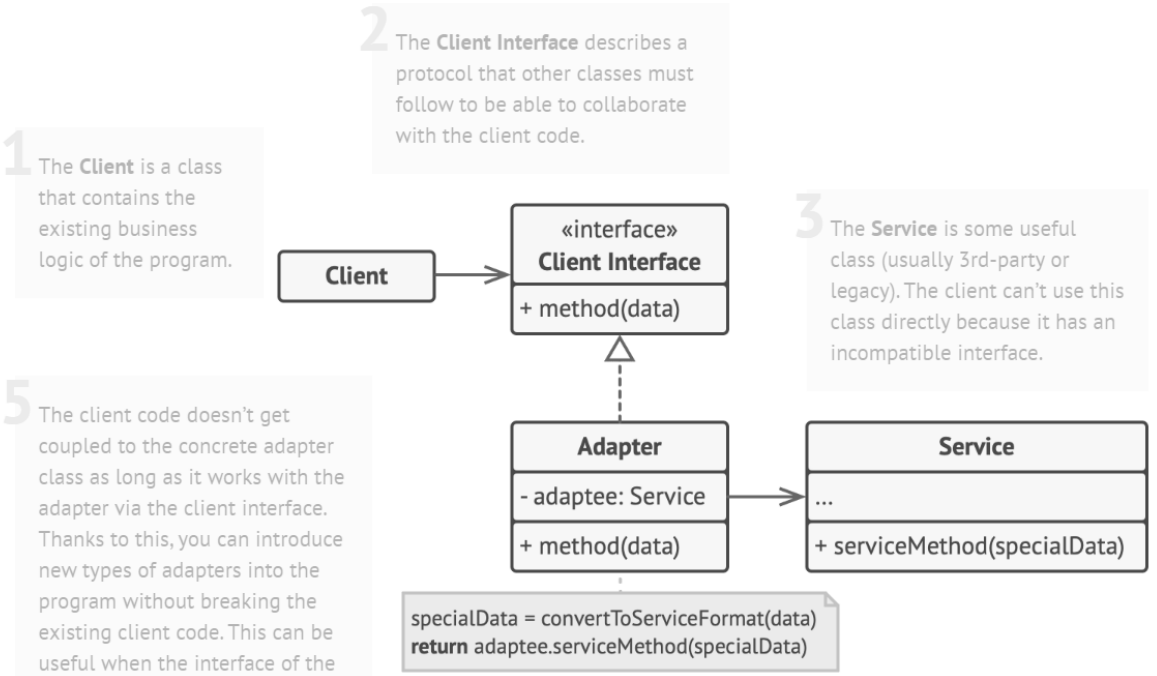


# 3.1 Adaptor Pattern (Continued)

## ❖ Structure

**Object adapter**

This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.

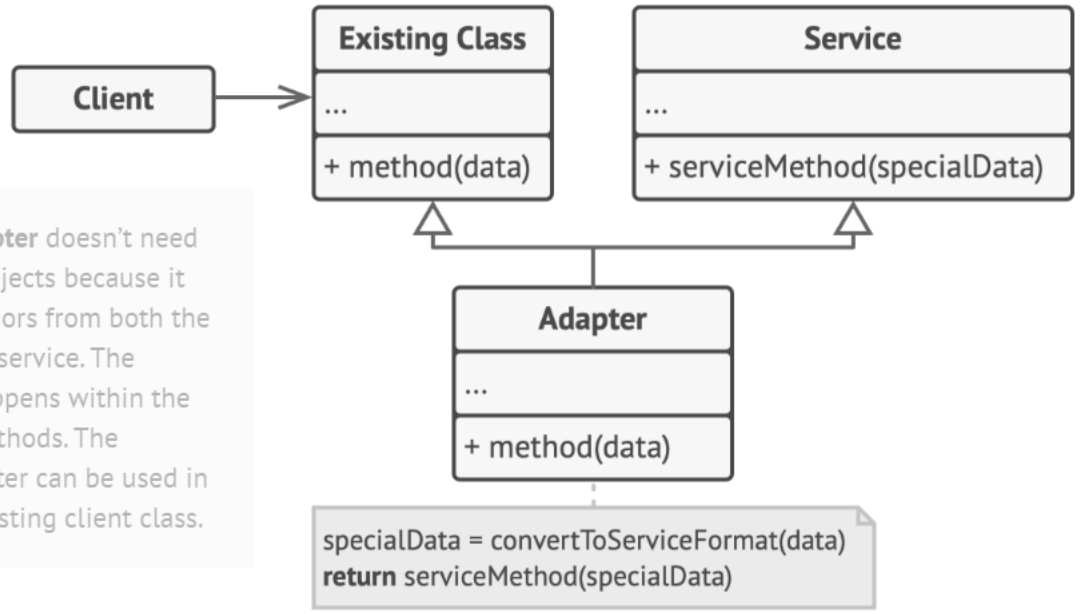


# 3.1 Adaptor Pattern (Continued)

## ❖ Structure - Continued

### Class adapter

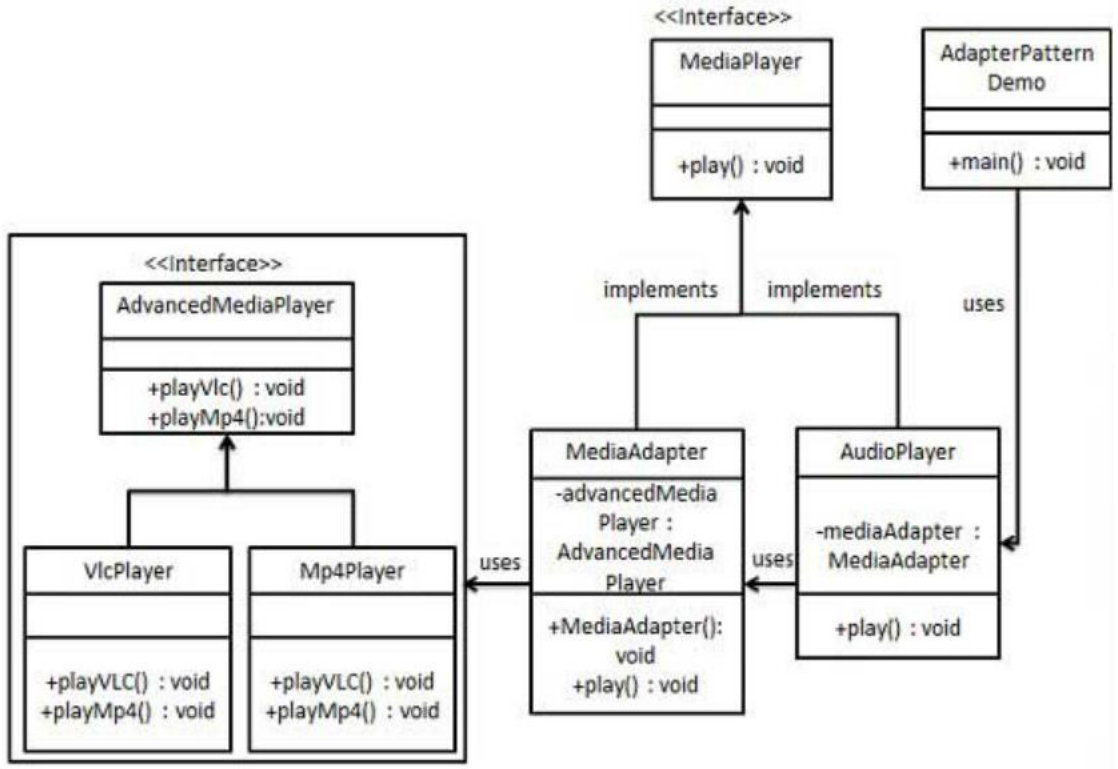
This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time. Note that this approach can only be implemented in programming languages that support multiple inheritance, such as C++.




1 The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

# 3.1 Adaptor Pattern (Continued)

## ❖ Implementation




➢ **Implementation –**

- We have a MediaPlayer interface and a concrete class AudioPlayer implementing theMediaPlayer 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.

# 3.1 Adaptor Pattern (Continued)

## ❖ Implementation - Continued

### 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
   }
}
```

*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);
      }
   }
}
```

# 3.1 Adaptor Pattern (Continued)

## ❖ Implementation - Continued

### 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");
   }
}
```

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
```

# 3.1 Adaptor Pattern (Continued)

## ❖ Applicability

**Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.**

The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.

---

**Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.**

You could extend each subclass and put the missing functionality into new child classes. However, you'll need to duplicate the code across all of these new classes, which **smells really bad**.

# 3.1 Adaptor Pattern (Continued)

## ❖ Pros and Cons

✓ *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.

✓ *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

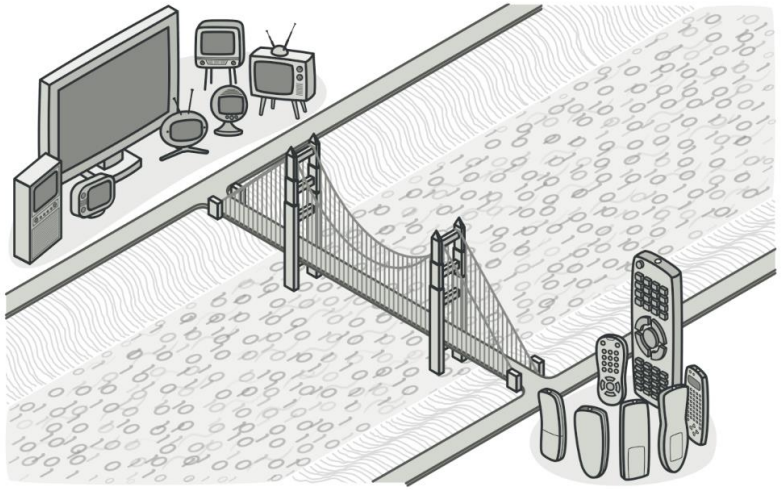
✗ The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

# 3.2 Bridge Pattern

**Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.**

## ❖ Introduction

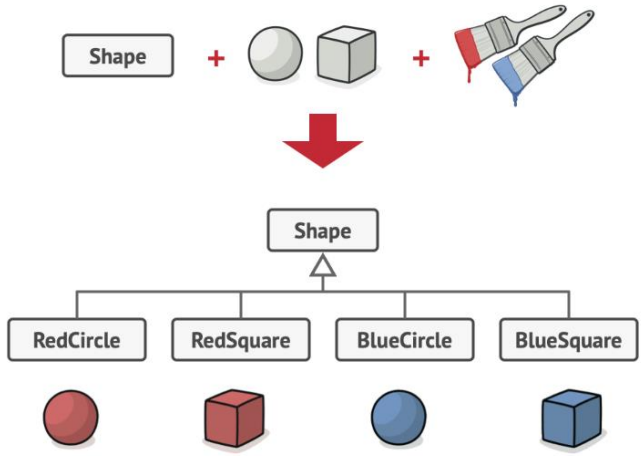
- When we have interface hierarchies in both interfaces as well as implementations, then the bridge design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client programs.
- The implementation of bridge design pattern follows the notion to prefer Composition over inheritance.
- Decouple an abstraction from its implementation so that the two can vary independently



# 3.2 Bridge Pattern (Continued)

### ❖ Problem

- Abstraction? Implementation? Sound scary? Stay calm and let's consider a simple example.
- Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.
- Adding new shape types and colors to the hierarchy will grow it exponentially. For example, to add a triangle shape you'd need to introduce two subclasses, one for each color. And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

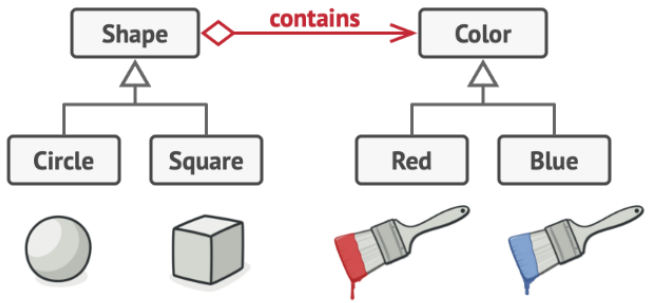



*Number of class combinations grows in geometric progression.*

# 3.2 Bridge Pattern (Continued)

### ❖ Solution

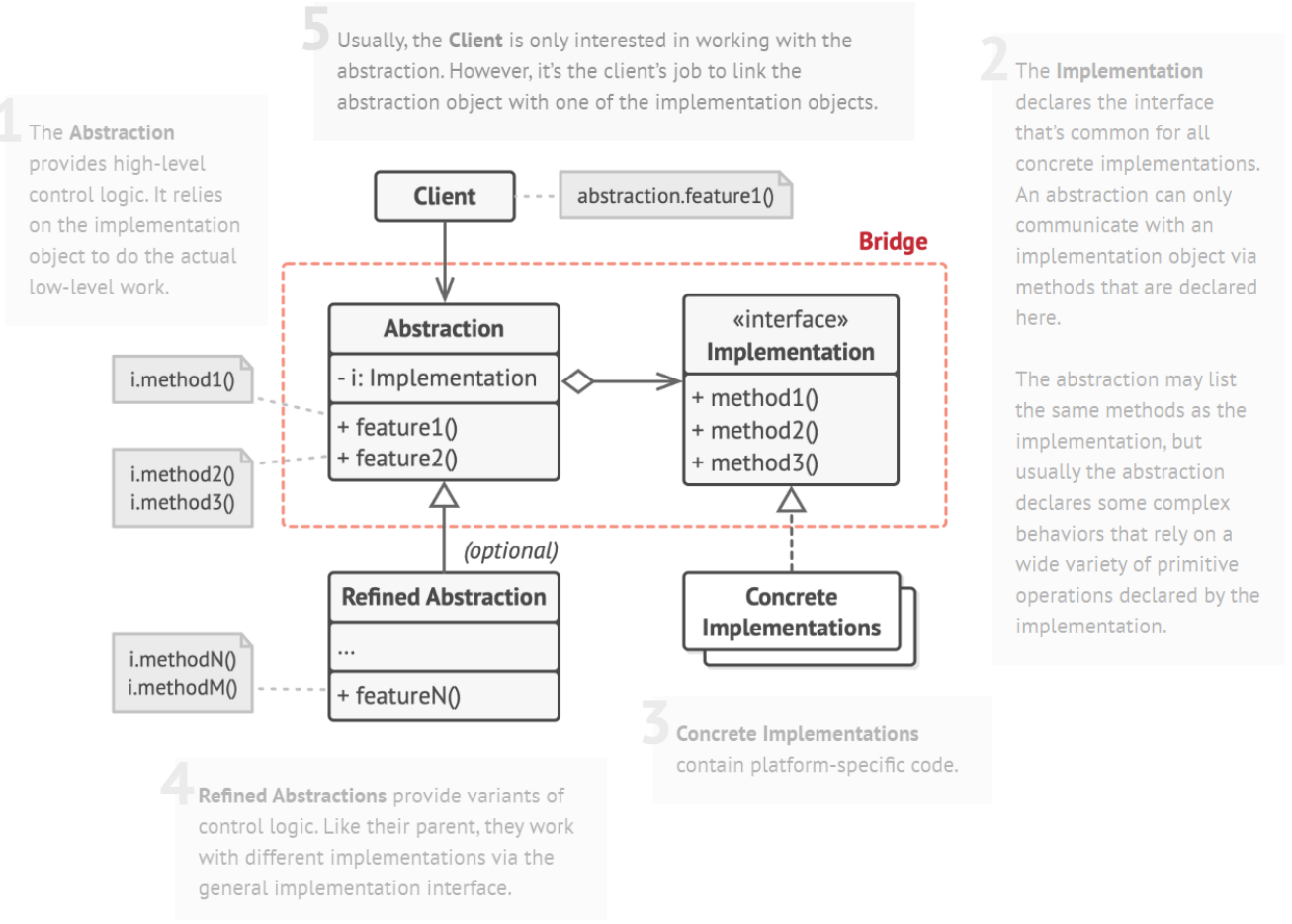
- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.
- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.
- Following this approach, we can extract the color-related code into its own class with two subclasses: Red and Blue. The Shape class then gets a reference field pointing to one of the color objects. Now the shape can delegate any color-related work to the linked color object. That reference will act as a bridge between the Shape and Color classes. From now on, adding new colors won't require changing the shape hierarchy, and vice versa.




*You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.*

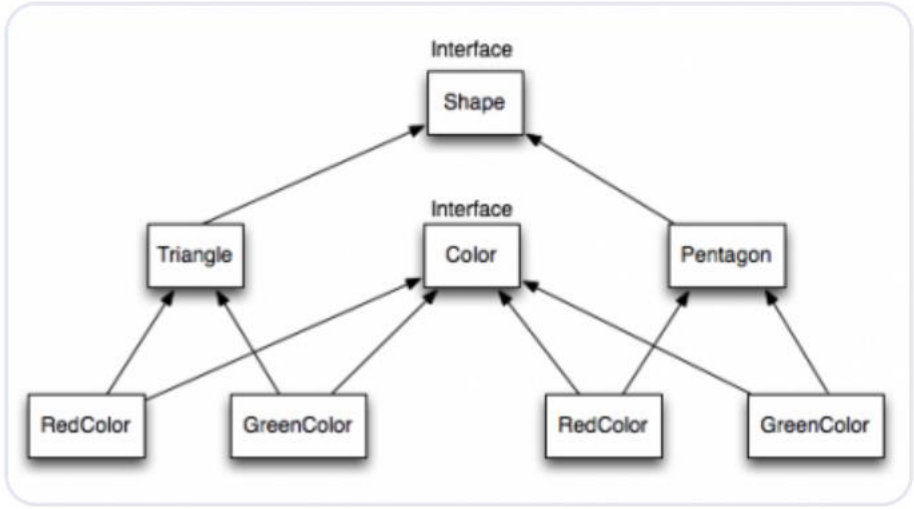
# 3.2 Bridge Pattern (Continued)

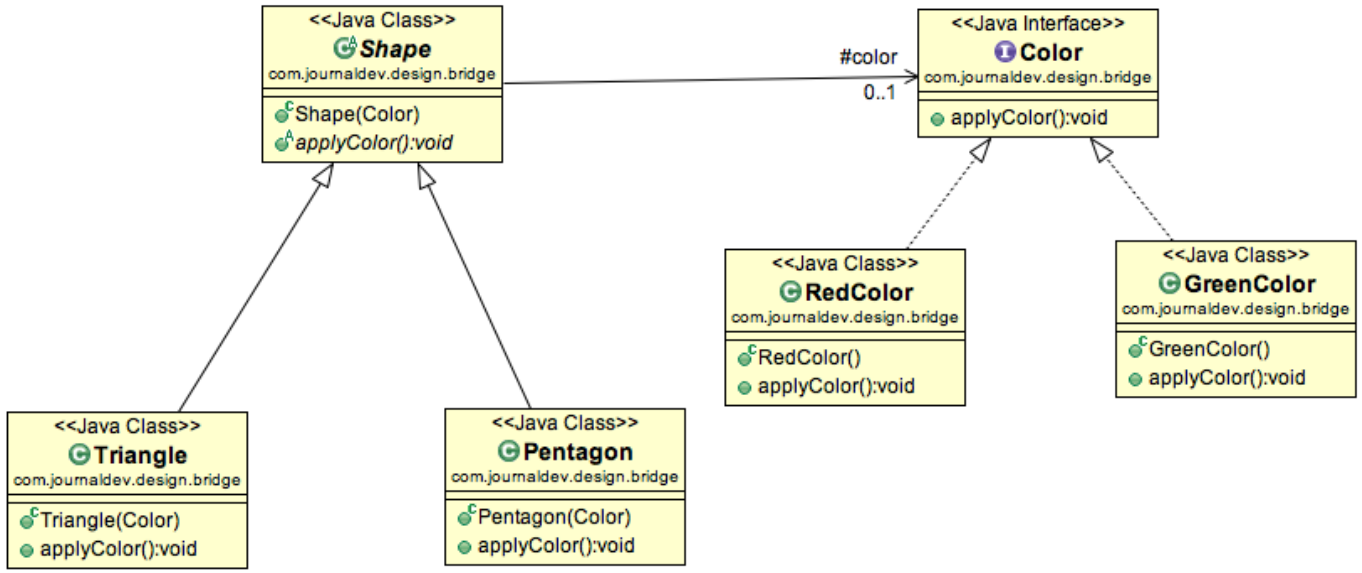
❖ **Structure**



# 3.2 Bridge Pattern (Continued)

- **Implementation**




**Client Contacts the Shape Abstract Class*

# 3.2 Bridge Pattern (Continued)

## ❖ Implementation - Continued

### Step 1

Create bridge implementer interface.

```
package com.journaldev.design.bridge;

public interface Color {

        public void applyColor();
}
```

```
package com.journaldev.design.bridge;

public abstract class Shape {
        //Composition - implementor
        protected Color color;

        //constructor with implementor as input argument
        public Shape(Color c){
                this.color=c;
        }

        abstract public void applyColor();
}
```

### Step 2

```
package com.journaldev.design.bridge;

public class Triangle extends Shape{

        public Triangle(Color c) {
                super(c);
        }

        @Override
        public void applyColor() {
                System.out.print("Triangle filled with color ");
                color.applyColor();
        }

}
```

```
package com.journaldev.design.bridge;

public class Pentagon extends Shape{

        public Pentagon(Color c) {
                super(c);
        }

        @Override
        public void applyColor() {
                System.out.print("Pentagon filled with color ");
                color.applyColor();
        }

}
```

# 3.2 Bridge Pattern (Continued)

- **Implementation - Continued**

### Step 3

```
package com.journaldev.design.bridge;

public class RedColor implements Color{

        public void applyColor(){
                System.out.println("red.");
        }
}
```

```
package com.journaldev.design.bridge;

public class GreenColor implements Color{

        public void applyColor(){
                System.out.println("green.");
        }
}
```

### Step 4

```
package com.journaldev.design.test;

import com.journaldev.design.bridge.GreenColor;
import com.journaldev.design.bridge.Pentagon;
import com.journaldev.design.bridge.RedColor;
import com.journaldev.design.bridge.Shape;
import com.journaldev.design.bridge.Triangle;

public class BridgePatternTest {

        public static void main(String[] args) {
                Shape tri = new Triangle(new RedColor());
                tri.applyColor();

                Shape pent = new Pentagon(new GreenColor());
                pent.applyColor();
        }

}
```

Output of above bridge pattern example program is:

```
Triangle filled with color red.
Pentagon filled with color green.
```

# 3.2 Bridge Pattern (Continued)

## ❖ Applicability

**Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).**

The bigger a class becomes, the harder it is to figure out how it works, and the longer it takes to make a change. The changes made to one of the variations of functionality may require making changes across the whole class, which often results in making errors or not addressing some critical side effects.

The Bridge pattern lets you split the monolithic class into several class hierarchies. After this, you can change the classes in each hierarchy independently of the classes in the others. This approach simplifies code maintenance and minimizes the risk of breaking existing code.

**Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.**

The Bridge suggests that you extract a separate class hierarchy for each of the dimensions. The original class delegates the related work to the objects belonging to those hierarchies instead of doing everything on its own.

**Use the Bridge if you need to be able to switch implementations at runtime.**

Although it's optional, the Bridge pattern lets you replace the implementation object inside the abstraction. It's as easy as assigning a new value to a field.

By the way, this last item is the main reason why so many people confuse the Bridge with the **Strategy** pattern. Remember that a pattern is more than just a certain way to structure your classes. It may also communicate intent and a problem being addressed.

# 3.2 Bridge Pattern (Continued)

## ❖ Pros and Cons

- ✓ You can create platform-independent classes and apps.
- ✓ The client code works with high-level abstractions. It isn't exposed to the platform details.
- ✓ *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other.
- ✓ *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

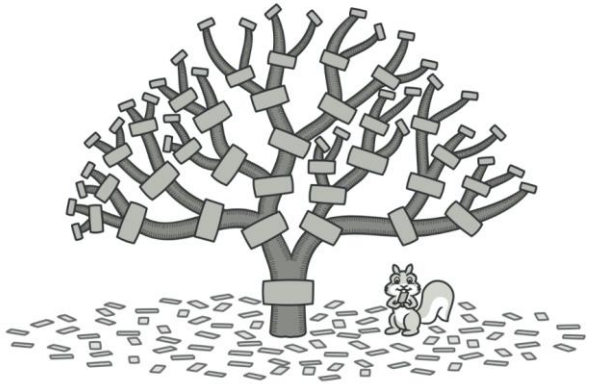
- ✗ You might make the code more complicated by applying the pattern to a highly cohesive class.

# 3.3 Composite Pattern

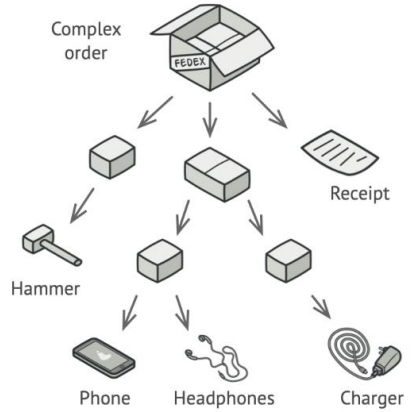
**Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.**

## ❖ Introduction

- Composite pattern is one of the Structural design patterns and is used when we have to represent a part-whole hierarchy.
- **When we need to create a structure in a way that the objects in the structure have to be treated the same way, we can apply the composite design pattern.**
- Let's understand it with a real-life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle, etc and when we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have the same operations.



# 3.3 Composite Pattern - Continued




*An order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so on. The whole structure looks like an upside down tree.*

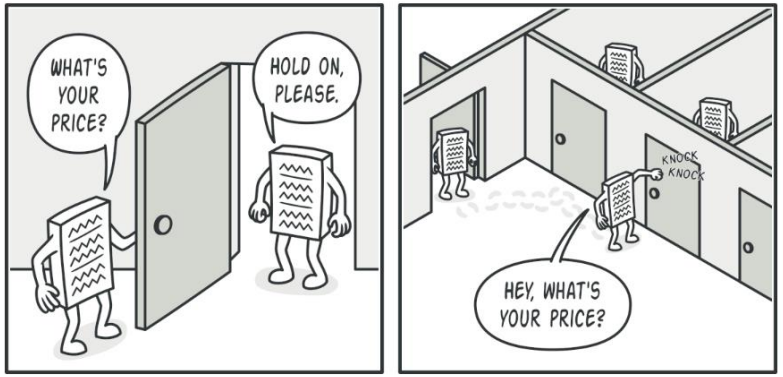
## ❖ Problem

- Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.
- For example, imagine that you have two types of objects: Products and Boxes. A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.
- Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?
- An order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so on. The whole structure looks like an upside down tree.
- You could try the direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it’s not as simple as running a loop. You have to know the classes of Products and Boxes you’re going through, the nesting level of the boxes and other nasty details beforehand. All of this makes the direct approach either too awkward or even impossible.

# 3.3 Composite Pattern - Continued

## ❖ Solution

- The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.
- How would this method work? For a product, it'd simply return the product's price. For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.
- The greatest benefit of this approach is that you don't need to care about the concrete classes of objects that compose the tree. You don't need to know whether an object is a simple product or a sophisticated box. You can treat them all the same via the common interface. When you call a method, the objects themselves pass the request down the tree.




*The Composite pattern lets you run a behavior recursively over all components of an object tree.*

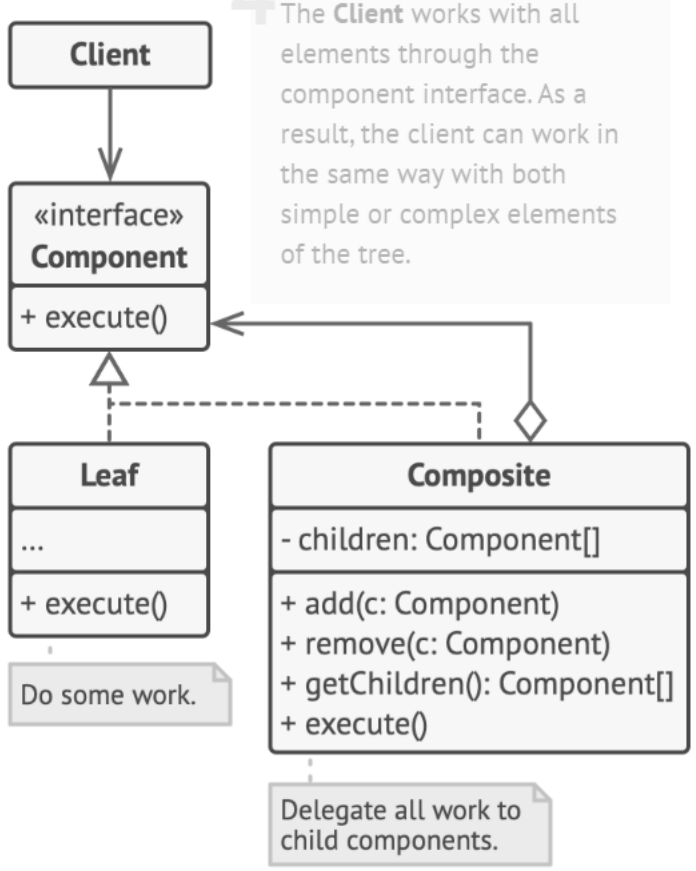
# 3.3 Composite Pattern - Continued

❖ **Structure**

1 The **Component** interface describes operations that are common to both simple and complex elements of the tree.

2 The **Leaf** is a basic element of a tree that doesn't have sub-elements.

Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.



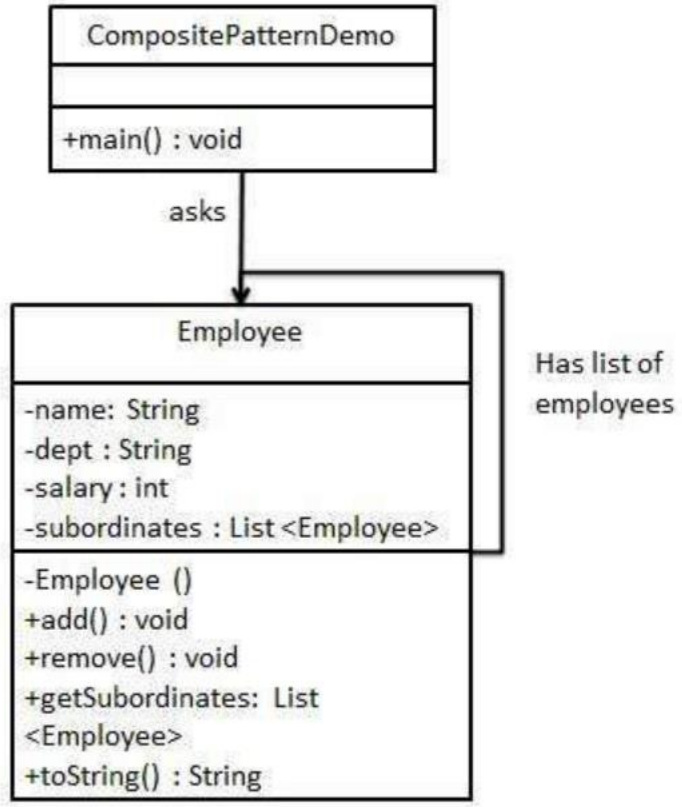

4 The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

3 The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

# 3.3 Composite Pattern - Continued

❖ **Implementation**




➢ **Implementation –**

- We have a class Employee which acts as composite pattern actor class. CompositePatternDemo, our demo class, will use Employee class to add department level hierarchy and print all employees.

# 3.3 Composite Pattern - Continued

- **Implementation - Continued**

### Step 1

Create *Employee* class having list of *Employee* objects.

*Employee.java*

```
import java.util.ArrayList;
import java.util.List;

public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;

   // constructor
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }

   public void remove(Employee e) {
      subordinates.remove(e);
   }

   public List<Employee> getSubordinates(){
     return subordinates;
   }

   public String toString(){
      return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");
   }
}
```

# 3.3 Composite Pattern - Continued

## ❖ Implementation - Continued

**Step 2**

Use the *Employee* class to create and print employee hierarchy.

*CompositePatternDemo.java*

```
public class CompositePatternDemo {
   public static void main(String[] args) {

      Employee CEO = new Employee("John","CEO", 30000);

      Employee headSales = new Employee("Robert","Head Sales", 20000);

      Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

      Employee clerk1 = new Employee("Laura","Marketing", 10000);
      Employee clerk2 = new Employee("Bob","Marketing", 10000);

      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
      Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

      CEO.add(headSales);
      CEO.add(headMarketing);

      headSales.add(salesExecutive1);
      headSales.add(salesExecutive2);

      headMarketing.add(clerk1);
      headMarketing.add(clerk2);

      //print all employees of the organization
      System.out.println(CEO);

      for (Employee headEmployee : CEO.getSubordinates()) {
         System.out.println(headEmployee);

         for (Employee employee : headEmployee.getSubordinates()) {
            System.out.println(employee);
         }
      }
   }
}
```

Verify the output.

```
Employee :[ Name : John, dept : CEO, salary :30000 ]
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]
Employee :[ Name : Richard, dept : Sales, salary :10000 ]
Employee :[ Name : Rob, dept : Sales, salary :10000 ]
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]
Employee :[ Name : Bob, dept : Marketing, salary :10000 ]
```

# 3.3 Composite Pattern - Continued

## ❖ Applicability

**Use the Composite pattern when you have to implement a tree-like object structure.**

The Composite pattern provides you with two basic element types that share a common interface: simple leaves and complex containers. A container can be composed of both leaves and other containers. This lets you construct a nested recursive object structure that resembles a tree.

---

**Use the pattern when you want the client code to treat both simple and complex elements uniformly.**

All elements defined by the Composite pattern share a common interface. Using this interface, the client doesn't have to worry about the concrete class of the objects it works with.

# 3.3 Composite Pattern - Continued

## ❖ Pros and Cons

- ✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- ✓ *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

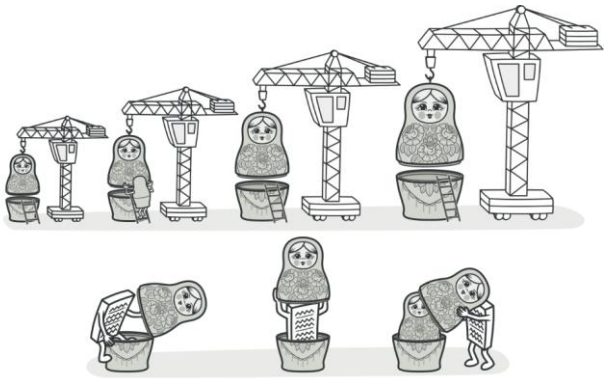
- ✗ It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

# 3.4 Decorator Pattern

**Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.**

## ❖ Introduction

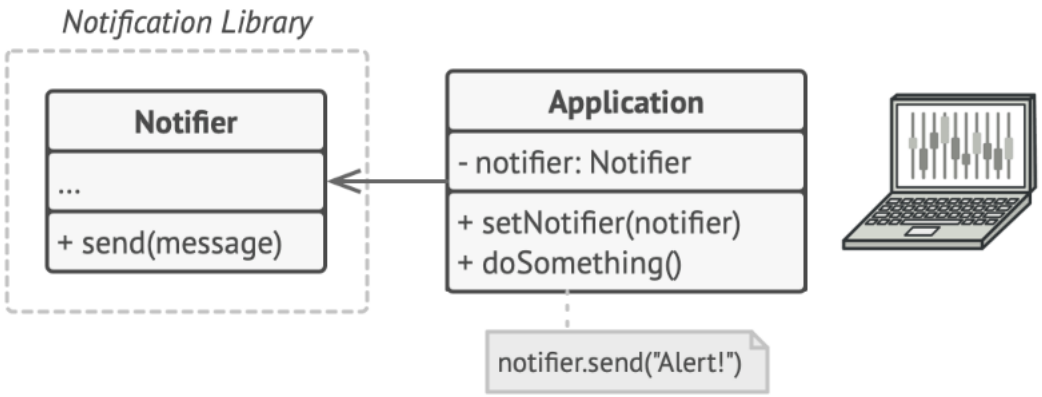
- Decorator design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior.
- Decorator design pattern is one of the structural design pattern (such as Adapter Pattern, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with composition to implement.
- Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.



# 3.4 Decorator Pattern (Continued)

## ❖ Problem

- Imagine that you're working on a notification library which lets other programs notify their users about important events.

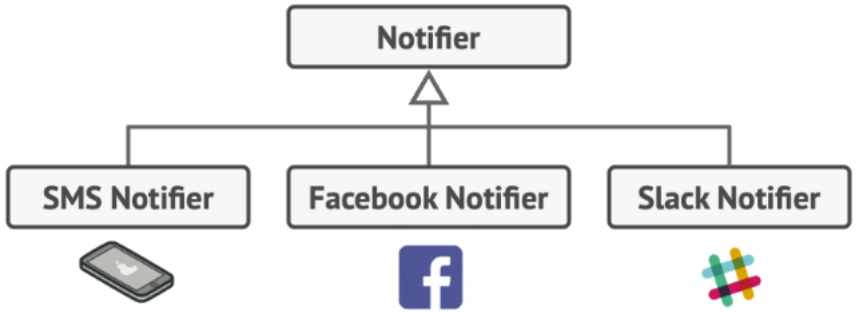



*A program could use the notifier class to send notifications about important events to a predefined set of emails.*

# 3.4 Decorator Pattern (Continued)

❖ **Problem**

- At some point, you realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.

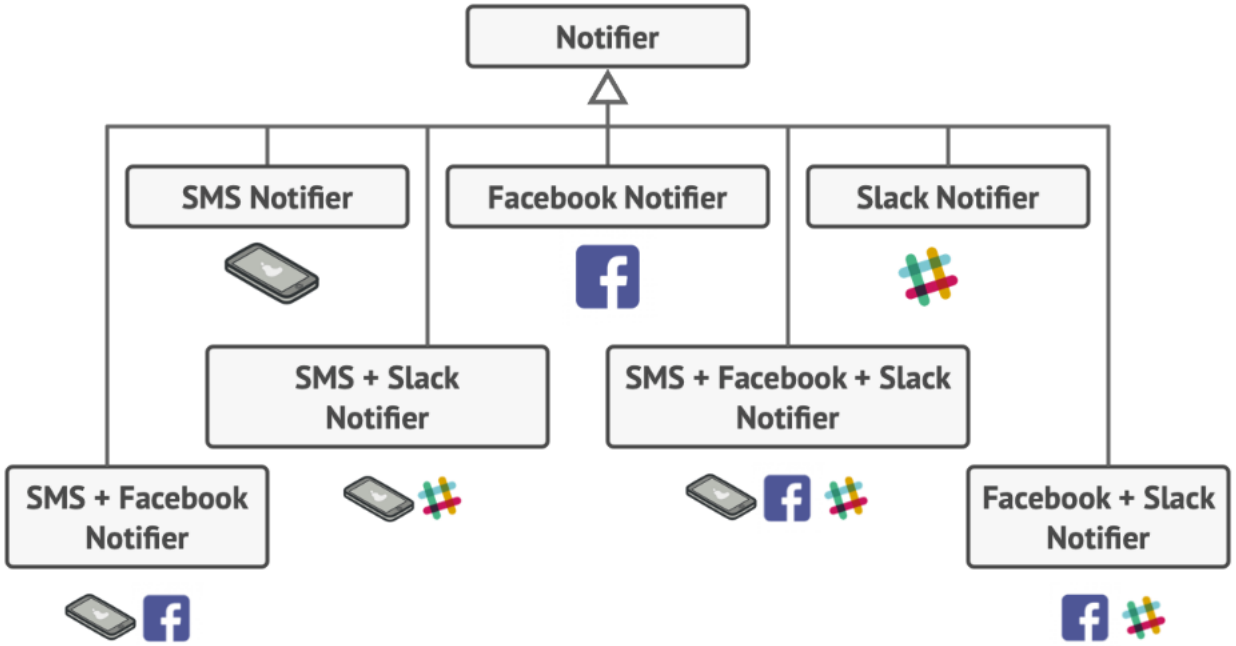



*Each notification type is implemented as a notifier's subclass.*

# 3.4 Decorator Pattern (Continued)

❖ **Problem**

- But then someone reasonably asked you, “Why can’t you use several notification types at once? If your house is on fire, you’d probably want to be informed through every channel.”




*Combinatorial explosion of subclasses.*

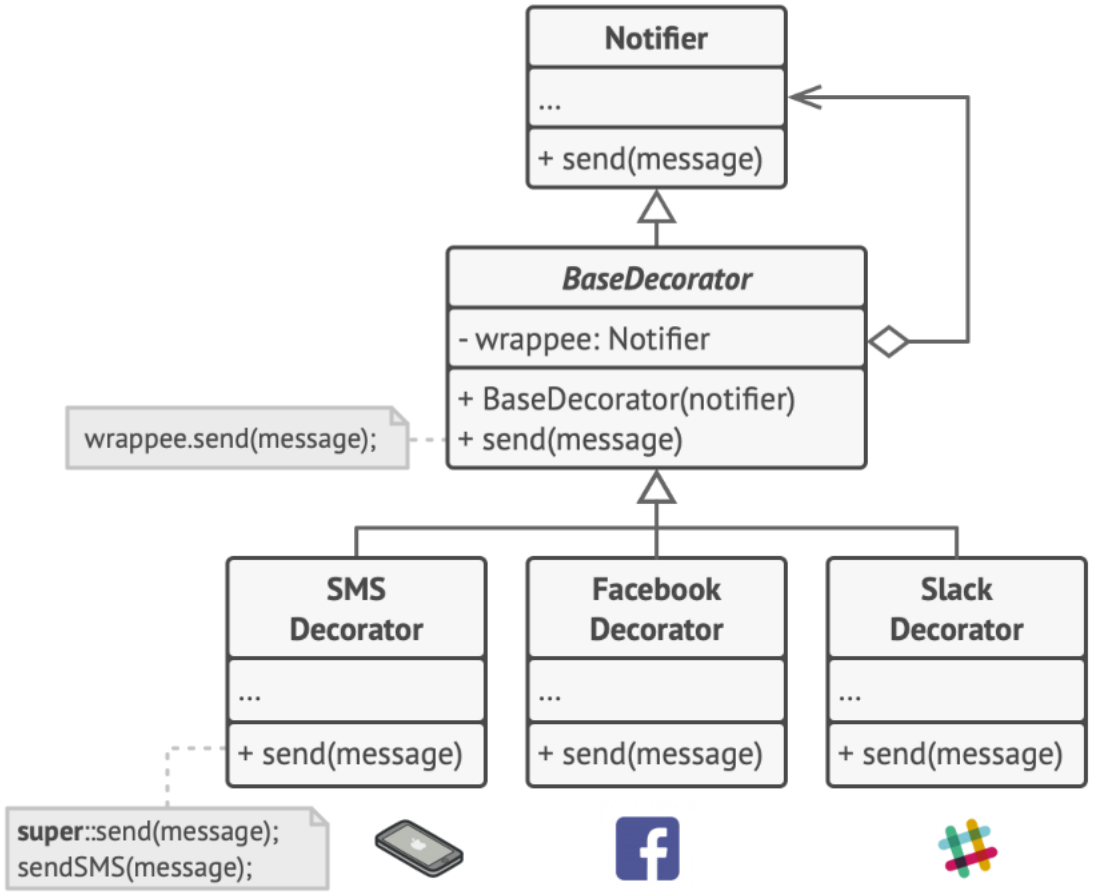
# 3.4 Decorator Pattern (Continued)

❖ **Solution**

- "Wrapper" is the alternative nickname for the Decorator pattern that clearly expresses the main idea of the pattern. A wrapper is an object that can be linked with some target object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.
- When does a simple wrapper become the real decorator? **The wrapper implements the same interface as the wrapped object.** That's why from the client's perspective these objects are identical. Make the wrapper's reference field accept any object that follows that interface. This will let you cover an object in multiple wrappers, adding the combined behavior of all the wrappers to it.
- In our notifications example, let's leave the simple email notification behavior inside the base Notifier class, but turn all other notification methods into decorators.

# 3.4 Decorator Pattern (Continued)

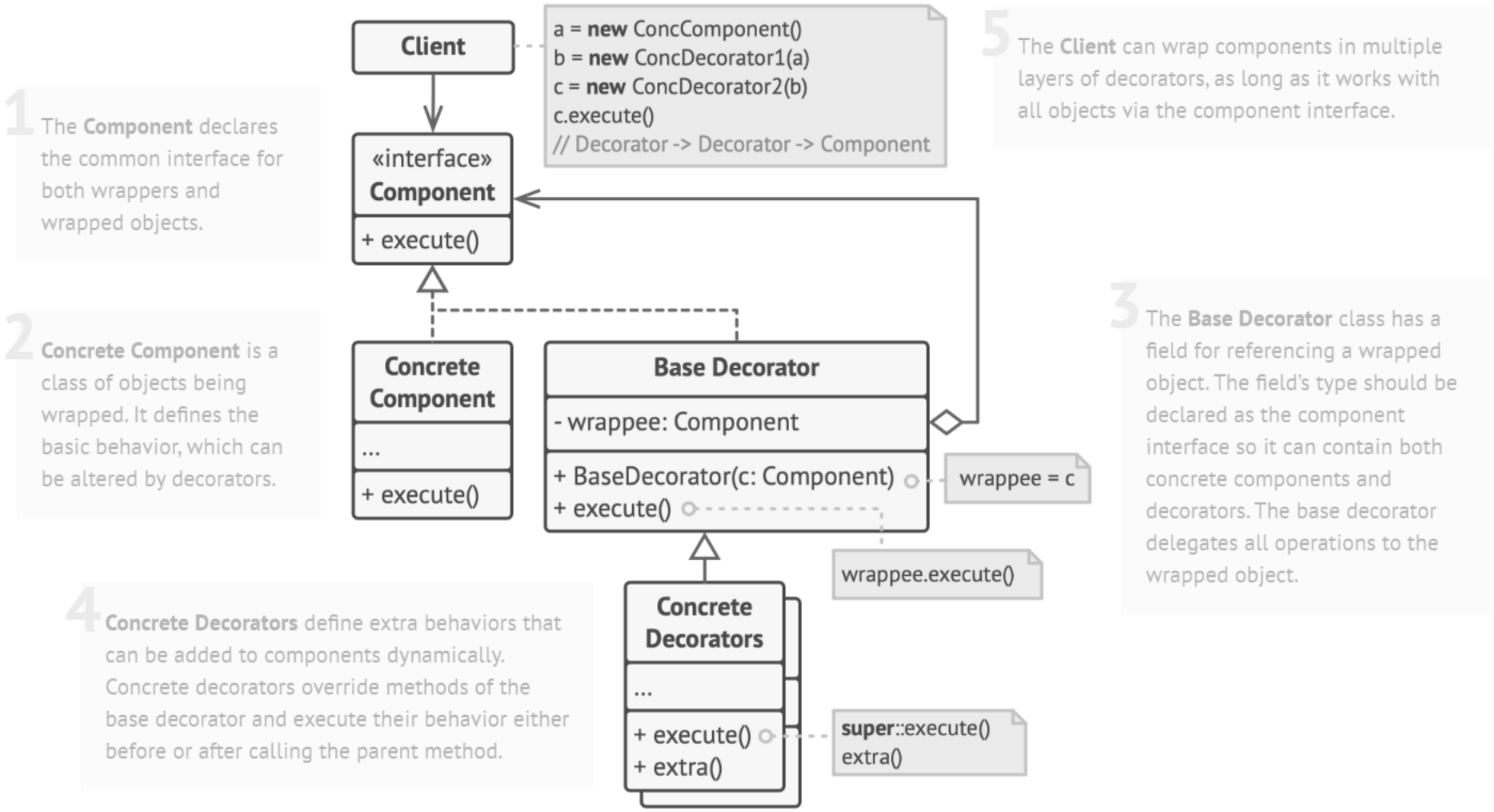
- **Solution (Continued)**




*Various notification methods become decorators.*

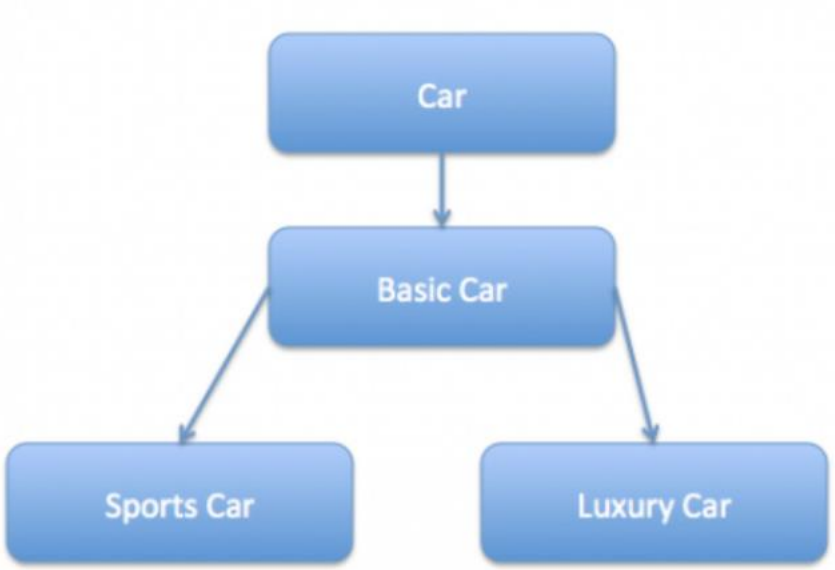
# 3.4 Decorator Pattern (Continued)

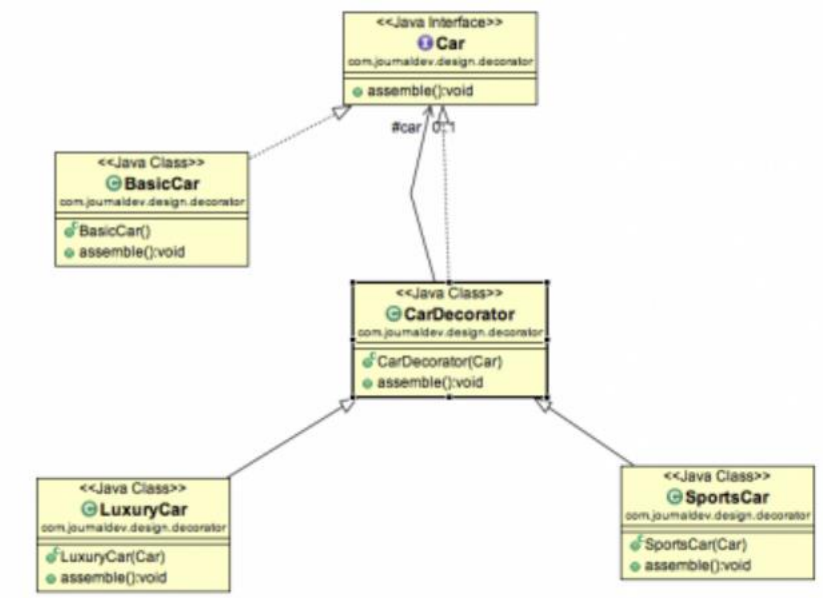
❖ **Structure**



# 3.4 Decorator Pattern (Continued)

- **Implementation**




**Client Contacts the Car Interface*

# 3.4 Decorator Pattern (Continued)

## ❖ Implementation (Continued)

1. **Component Interface** - The interface or **abstract class** defining the methods that will be implemented. In our case `Car` will be the component interface.

```
package com.journaldev.design.decorator;

public interface Car {

	public void assemble();
}
```

2. **Component Implementation** - The basic implementation of the component interface. We can have `BasicCar` class as our component implementation.

```
package com.journaldev.design.decorator;

public class BasicCar implements Car {

	@Override
	public void assemble() {
		System.out.print("Basic Car.");
	}

}
```

# 3.4 Decorator Pattern (Continued)

## ❖ Implementation (Continued)

3. **Decorator** - Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

```
package com.journaldev.design.decorator;

public class CarDecorator implements Car {

	protected Car car;

	public CarDecorator(Car c){
		this.car=c;
	}

	@Override
	public void assemble() {
		this.car.assemble();
	}

}
```

# 3.4 Decorator Pattern (Continued)

## ❖ Implementation (Continued)

4. **Concrete Decorators** - Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as `LuxuryCar` and `SportsCar`.

```
package com.journaldev.design.decorator;

public class SportsCar extends CarDecorator {

	public SportsCar(Car c) {
		super(c);
	}

	@Override
	public void assemble(){
		super.assemble();
		System.out.print(" Adding features of Sports Car.");
	}
}
```

```
package com.journaldev.design.decorator;

public class LuxuryCar extends CarDecorator {

	public LuxuryCar(Car c) {
		super(c);
	}

	@Override
	public void assemble(){
		super.assemble();
		System.out.print(" Adding features of Luxury Car.");
	}
}
```

# 3.4 Decorator Pattern (Continued)

## ❖ Implementation (Continued)

```
package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;
import com.journaldev.design.decorator.Car;
import com.journaldev.design.decorator.LuxuryCar;
import com.journaldev.design.decorator.SportsCar;

public class DecoratorPatternTest {

        public static void main(String[] args) {
                Car sportsCar = new SportsCar(new BasicCar());
                sportsCar.assemble();
                System.out.println("\n*****");

                Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
                sportsLuxuryCar.assemble();
        }

}
```

Notice that client program can create different kinds of Object at runtime and they can specify the order of execution too. Output of above test program is:

```
Basic Car. Adding features of Sports Car.
*****
Basic Car. Adding features of Luxury Car. Adding features of Sports Car.
```

# 3.4 Decorator Pattern (Continued)

## ❖ Applicability

**Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.**

The Decorator lets you structure your business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime. The client code can treat all these objects in the same way, since they all follow a common interface.

---

**Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.**

Many programming languages have the `final` keyword that can be used to prevent further extension of a class. For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

# 3.4 Decorator Pattern (Continued)

## ❖ Pros and Cons

- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.
- ✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

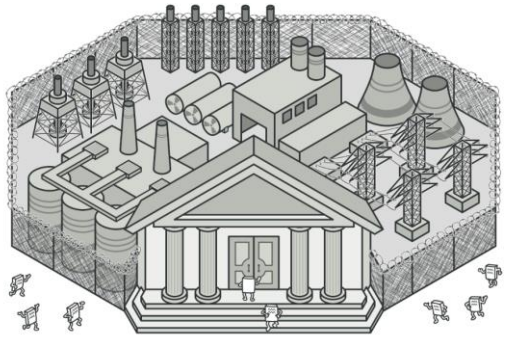
- ✗ It's hard to remove a specific wrapper from the wrappers stack.
- ✗ It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.
- ✗ The initial configuration code of layers might look pretty ugly.

# 3.5 Facade Pattern

**Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.**

## ❖ Introduction

- 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.
- Suppose 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 behavior 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.



# 3.5 Facade Pattern

### ❖ Problem

- Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

# 3.5 Facade Pattern

### ❖ Solution

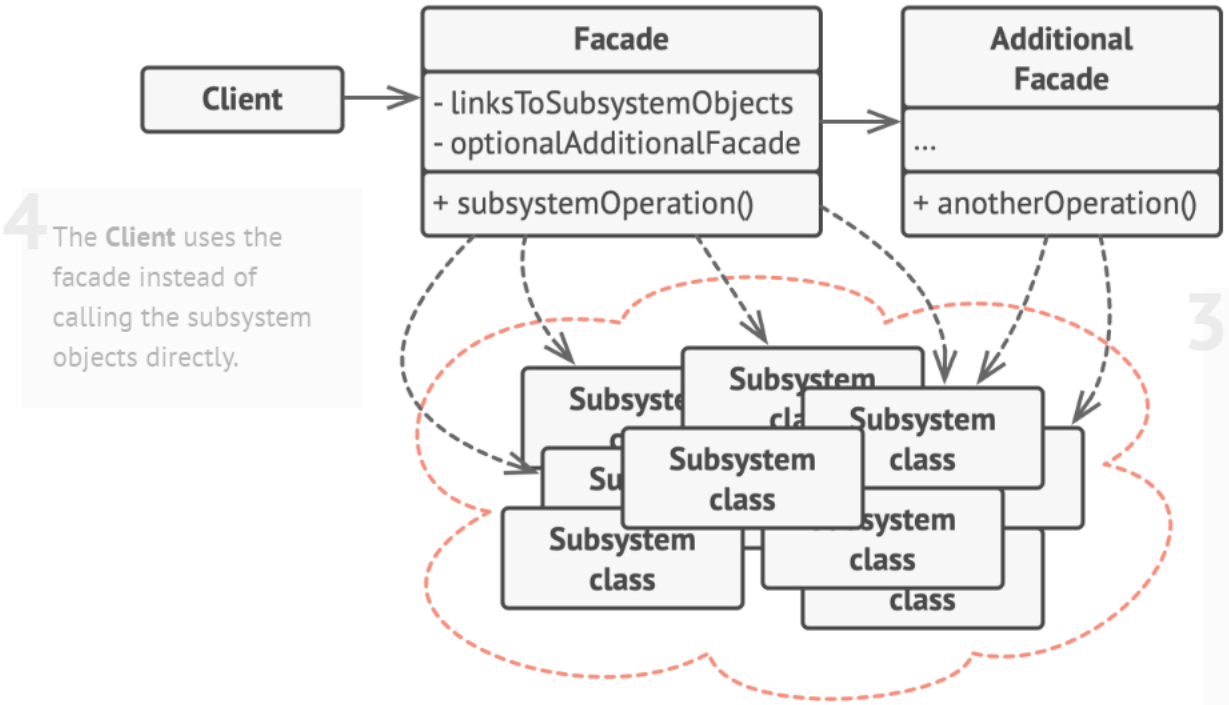
- **A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts**. A facade might provide limited functionality in comparison to working with the subsystem directly.
- However, it includes only those features that clients really care about.
- Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

# 3.5 Facade Pattern

## ❖ Structure

1 The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.

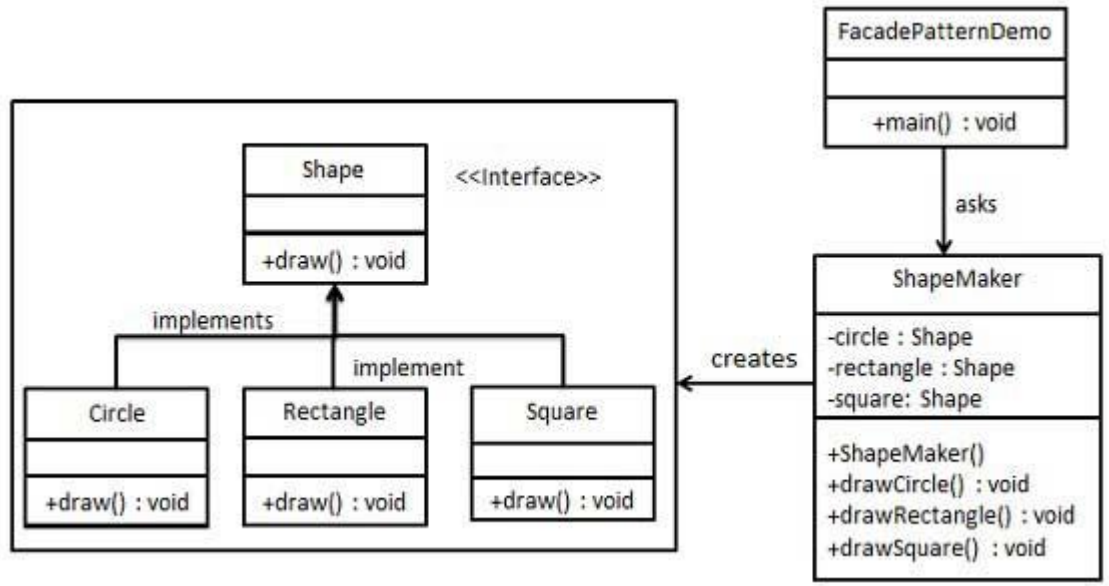
2 An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.




4 The **Client** uses the facade instead of calling the subsystem objects directly.

3 The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.

Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.


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

# 3.5 Facade Pattern

## ❖ Implementation

### 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()");
   }
}
```

# 3.5 Facade Pattern

- **Implementation (Continued)**

## 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();
   }
}
```

# 3.5 Facade Pattern

❖ **Implementation (Continued)**

### 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()
```

# 3.5 Facade Pattern

## ❖ Applicability

**Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.**

Often, subsystems get more complex over time. Even applying design patterns typically leads to creating more classes. A subsystem may become more flexible and easier to reuse in various contexts, but the amount of configuration and boilerplate code it demands from a client grows ever larger. The Facade attempts to fix this problem by providing a shortcut to the most-used features of the subsystem which fit most client requirements.

---

**Use the Facade when you want to structure a subsystem into layers.**

Create facades to define entry points to each level of a subsystem. You can reduce coupling between multiple subsystems by requiring them to communicate only through facades.

# 3.5 Facade Pattern

## ❖ Pros and Cons

✓ You can isolate your code from the complexity of a subsystem.

✕ A facade can become **a god object** coupled to all classes of an app.


# Recap…

# **Thank** You!