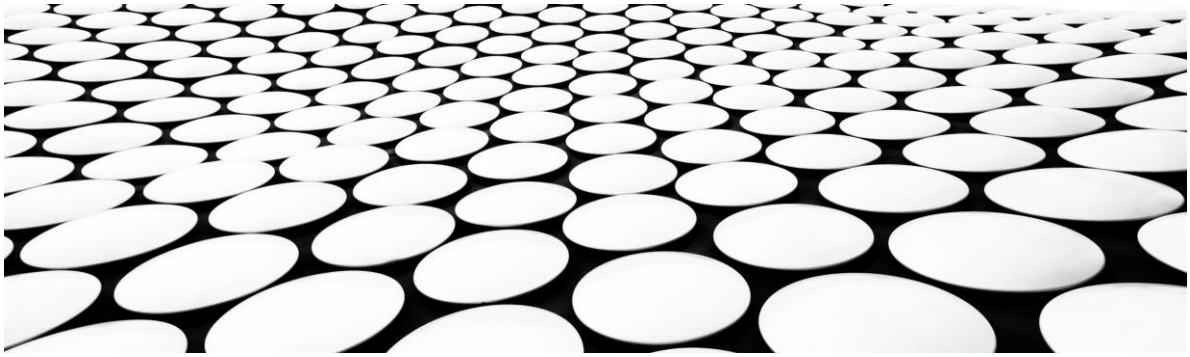
STRUCTURAL DESIGN PATTERNS

ADAPTER, BRIDGE



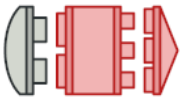
THE CONCEPT OF STRUCTURAL DESIGN PATTERNS

- Structural design patterns are those that simplify the design of large object structures by identifying relationships between them. They describe common ways of composing classes and objects so that they become repeatable as solutions.
- Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.
- Structural design patterns show you how to glue different pieces of a system together in a flexible and extensible fashion. They help you guarantee that when one of the parts changes, the entire structure does not need to change.

TYPES OF STRUCTURAL DESIGN PATTERN

- The Gang of Four has described seven such structural ways or patterns
  1. Adapter Pattern Adapting an interface into another according to client expectation.
  2. Bridge Pattern Separating abstraction (interface) from implementation.
  3. Composite Pattern Allowing clients to operate on hierarchy of objects.
  4. Decorator Pattern Adding functionality to an object dynamically.
  5. Facade Pattern Providing an interface to a set of interfaces.
  6. Flyweight Pattern Reusing an object by sharing it.
  7. Proxy Pattern Representing another object.

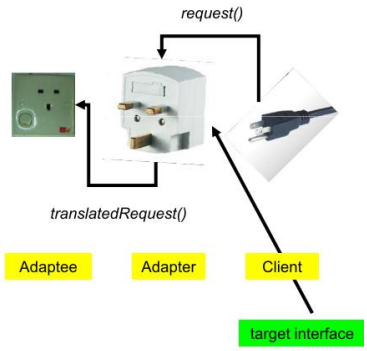
ADAPTER PATTERN

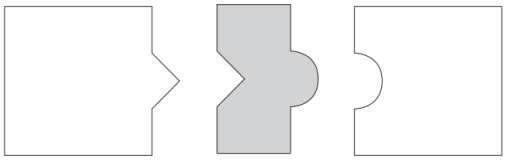


- Ever tried to use your *camera memory card* in your laptop. You cannot use it directly simply because there is no port in laptop which accept it. You must use a compatible card reader. You put your memory card into the card reader and then inject the card reader into the laptop. This card reader can be called the adapter.
- A similar example is your *mobile charger* or your *laptop charger* which can be used with any power supply without fear of the variance power supply in different locations. That is also called power "adapter".
- In programming as well, adapter pattern is used for similar purposes. It enables two incompatible interfaces to work smoothly with each other.
- Every day millions of lines of code is written. Most of the code is rewritten. Many a times there is a small change in requirement and old code doesn't fit. What if we adapt the code so that it becomes reusable? **Adapter design pattern in java helps to reuse the existing code** even if it is incompatible.

CONCEPT

- **Adapter Pattern acts as an intermediary to convert an otherwise incompatible interface to one that a client expects.**
- This is useful in cases where we want to take an existing class whose source code cannot be modified and make it work with another class.
- The Adapter Pattern is also known as **Wrapper.** Wrap an existing class with a new interface.

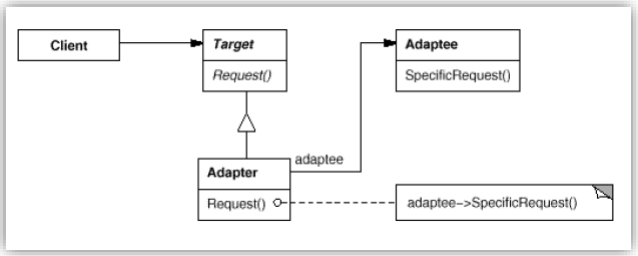




WHEN TO USE ADAPTER PATTERN

- When an outside component provides captivating functionality that we'd like to reuse, but it's incompatible with our current application. A suitable Adapter can be developed to make them compatible with each other
- When our application is not compatible with the interface that our client is expecting
- When we want to reuse legacy code in our application without making any modification in the original code

STRUCTURE & PARTICIPANT



- Target defines the domain-specific interface that Client uses.
- Client collaborates with objects conforming to the Target interface. This class will interact with the Adapter class.
- Adaptee defines an existing interface that needs adapting. This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.
- Adapter adapts the interface of Adaptee to the Target interface. This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

HOW TO IMPLEMENT

- Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
- Identify the interface that the client requires.
- Design a "wrapper" class that can "impedance match" the adaptee to the client.
- The adapter/wrapper class "has a" instance of the adaptee class.
- The adapter/wrapper class "maps" the client interface to the adaptee interface.
- The client uses (is coupled to) the new interface

EXAMPLE:

- In this example, we can calculate the area of a rectangle easily using the Calculator class and its getArea() method that uses a rectangle as an input .
- Now suppose we want to calculate the area of a triangle, but we need to get the area of the triangle through the getArea() method of Calculator. How can we do that?
- To do that we have made a CalculatorAdapter for the triangle and passed a triangle in its getArea() method.
- The method will translate the triangle input to rectangle input and in turn, it will call the getArea() of Calculator to get the area of it.
- From the user's point of view, it seems to the user that he is passing a triangle to get the area of that triangle.

ADAPTEE

```
class Calculator {

    Rect rectangle;

    public double getArea(Rect r) {
        rectangle = r;
        return rectangle.l * rectangle.w;
    }
}
```

ADAPTER

```
class CalculatorAdapter {

    Calculator calculator;
    Triangle triangle;

    public double getArea(Triangle t) {
        calculator = new Calculator();
        triangle = t;
        Rect r = new Rect();
        //Area of Triangle=0.5*base*height
        r.l = triangle.b;
        r.w = 0.5 * triangle.h;
        return calculator.getArea(r);
    }

}
```

TARGET

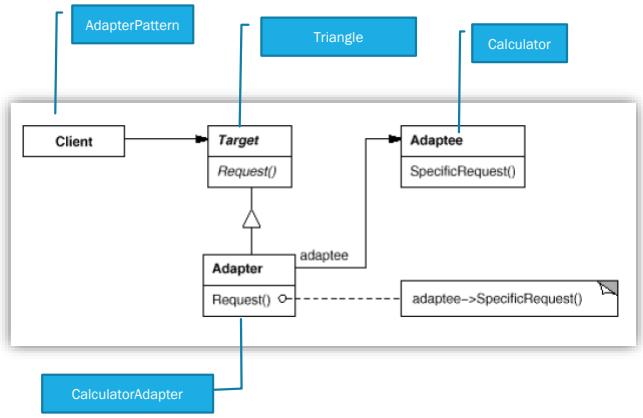
```
class Triangle
{
    public double b;//base
    public double h;//height
    public Triangle(int b, int h)
    {
        this.b = b;
        this.h = h;
    }
}
```

CLIENT

```
public class AdapterPattern {

    public static void main(String[] args) {
        System.out.println("***Adapter Pattern Demo***");
        CalculatorAdapter cal = new CalculatorAdapter();
        Triangle t = new Triangle(20, 10);
        System.out.println("\nAdapter Pattern Example\n");
        System.out.println("Area of Triangle is :" + cal.getArea(t));
    }
}
```

```
run:
***Adapter Pattern Demo***

Adapter Pattern Example

Area of Triangle is :100.0
```

PARTICIPANTS:



EXAMPLES OF STANDARD ADAPTERS IN JDK

- There are some standard Adapters in Java core libraries such as:

- java.util.Arrays#asList() :This method accepts multiple strings and return a list of input strings. Though it's very basic usage, but it's what an adapter does, right?
- java.io.InputStreamReader(InputStream) (returns a Reader object)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer object)

```
Writer writer = new OutputStreamWriter(new
FileOutputStream("c:\\data\\output.txt"));
writer.write("Hello World");
```

PROS AND CONS

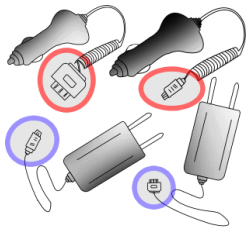
- Pros
  - Firstly, helps to reuse existing code
  - Secondly, incompatible code can communicate with each other
  - Also, adapter makes things work after they're designed
  - We can provide data in requested format even if we do not have
- Cons
  - Performance affected due to extra processes
  - All the processes communicate through adapters only

BRIDGE PATTERN

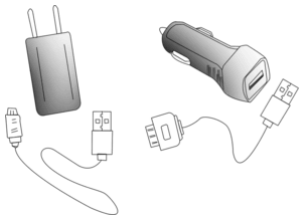
- **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.
- The format definition, directly from the GoF book, Design Patterns.
  - *"Decouples an abstraction from its implementation so that the two can vary independently."*

BRIDGE DESIGN PATTERN REAL LIFE EXAMPLE

- For understanding bridge design pattern we can take a real life example. It is very common now a days. Whenever we purchase any smartphone, we get a charger. The charger cable now-a-days can be separated, so we can use it to connect as USB cable to connect other devices.



As shown in image, we have 4 different chargers. 2 for house use and 2 for Car use.



If we implement bridge pattern then we just need 1 adapter each for house and for car use.

- Similarly, in case of software development there are so many cases, where we need separate things to make more reusable code. In such cases bridge design pattern is used to have communication between the components.

THE CONCEPT

- Let's clarify this using the abstraction and concrete implementation strategies. Assume you have two systems with abstract and concrete classes in each of the system. System X consists of 'Abstract A' and it's concrete implementation 'Concrete A'. Then, the System Y consists of 'Abstract B' and it's concrete implementation 'Concrete B'. Abstract A connects with the Abstract B by a 'Has-A' relationship. The **Has-A** relationship is achieved through composition where the abstraction maintains a reference of the implementation and forwards client requests to it. .Abstract A maintains an instance of Abstract B within itself. This 'Has-A' relationship makes a bridge between the two systems. Hence, the pattern contains two layers of abstraction. Thus, the pattern was called as the bridge pattern.

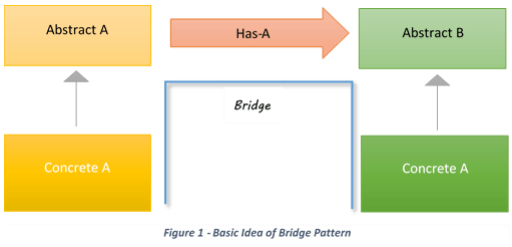


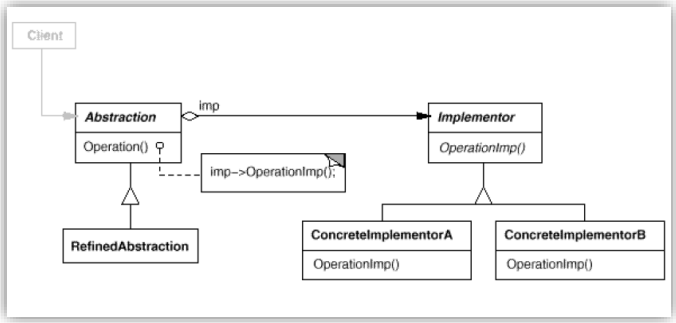
Figure 1 - Basic Idea of Bridge Pattern

WHERE TO USE

- When you want to separate the abstract structure and its concrete implementation.
- When you want to hide implementation details from clients. Changes in implementation should have no impact on clients.
- If any function is doing multiple things
  - We can separate into two functions
- If class is doing multiple tasks
  - We can make separate classes
- When we need code to be loosely coupled
  - Code should be less dependent on each other
- Functionality or behavior can be changed dynamically
  - Run time behavior can be changes by providing different implementer class

STRUCTURE & PARTICIPANTS

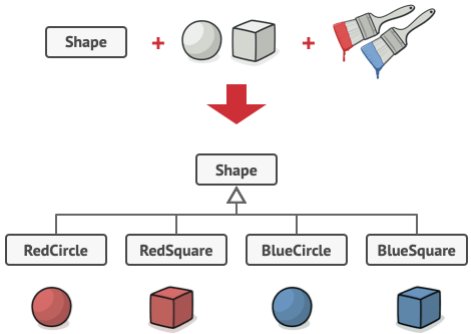
- **Abstraction**
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementer.
  - **RefinedAbstraction**
    - Extends the interface defined by Abstraction.
- **Implementer**
  - defines the interface for implementation classes.
- **ConcreteImplementor**
  - implements the Implementer interface and defines its concrete implementation.



EXAMPLE

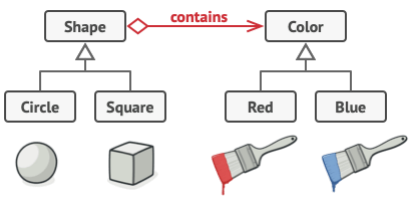
- Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as 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.

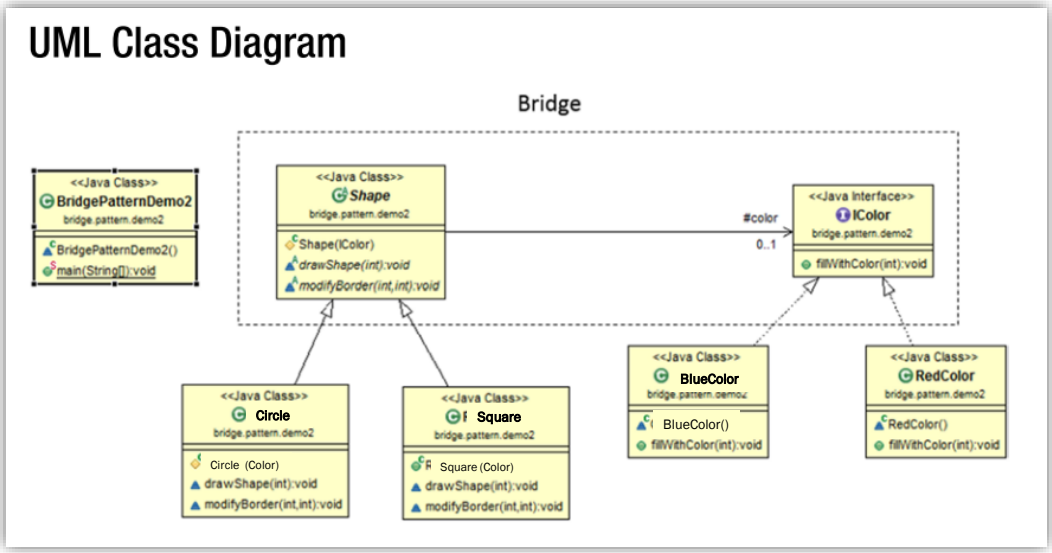


- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.
- The Bridge pattern attempts to solve this problem by switching from inheritance to **the object composition**. What this means 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.



EXAMPLE CONT...



ABSTRACTION

```
abstract class Shape {
    //Composition

    protected IColor color;

    protected Shape(IColor c) {
        this.color = c;
    }

    abstract void drawShape(int border);

    abstract void modifyBorder(int border, int increment);
}
```

REFINED ABSTRACTION

```
class Circle extends Shape {

    public Circle (IColore c){
        super(c);
    }
    @Override
    void drwaShape(int border) {
        System.out.print("This circle is colored with:");
        colore.fillWithColore(border);
    }

    @Override
    void modifyBorder(int border, int increment) {
        System.out.println("\n now we are changing the Diameter "
                + increment + "times");
    border= border* increment;
        drwaShape(border);
    }

}
```

```
class Square extends Shape {

    public Square (IColore c){
        super(c);
    }
    @Override
    void drwaShape(int border) {
        System.out.print("This square is colored with:");
        colore.fillWithColore(border);
    }

    @Override
    void modifyBorder(int border, int increment) {
        System.out.println("\n now we are changing the Diameter "
                + increment + "times");
    border= border* increment;
        drwaShape(border);
    }

}
```

IMPLEMENTER

```
interface IColor {

    void fillWithColor(int border);
}
```

CONCRETE IMPLEMENTER

```
class RedColor implements IColore{

    @Override
    public void fillWithColore(int border) {
        System.out.println("Red Color with" + border+ "inch border");    }

}
```

```
class BlueColor implements IColore{

    @Override
    public void fillWithColore(int border) {
        System.out.println("Blue Color with" + border+ "inch border");    }

}
```

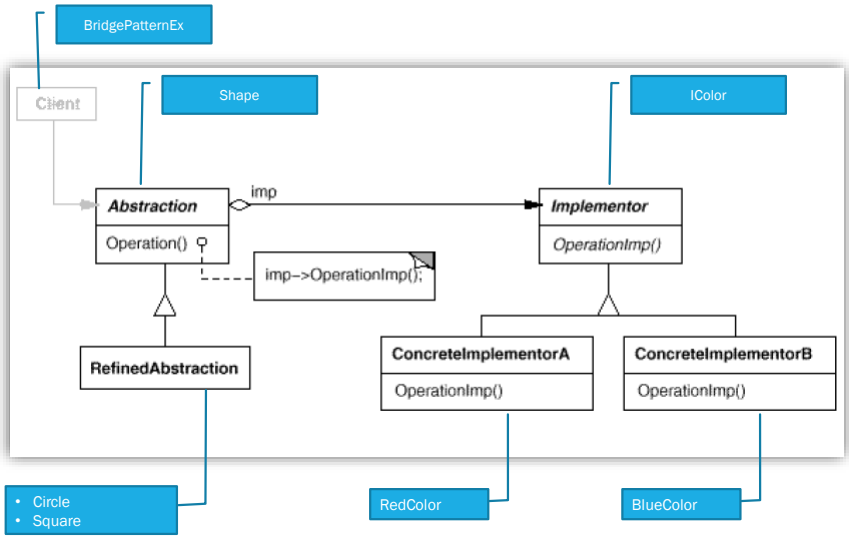
CLIENT

```
public class BridgePatternEx {

    public static void main(String[] args) {
        System.out.println("**********BRIDGE PATTERN**********");

        System.out.println("\nColoring Circle:");
        IColore red= new RedColor();
        Shape circleShape= new Circle(red);
        circleShape.drwaShape(20);
        circleShape.modifyBorder(20, 3);

        // blue square
        System.out.println("\nColoring Square:");
        IColore blue= new BlueColor();
        Shape squareShape= new Square(blue);
        squareShape.drwaShape(20);
        squareShape.modifyBorder(20, 3);
    }
}
```

Output:

```
Output - BridgePatternEx (run)
run:
**********BRIDGE PATTERN**********

Coloring Circle:
This circle is colored with:Red Color with20inch border

 now we are changing the Diameter 3times
This circle is colored with:Red Color with60inch border

Coloring Square:
This square is colored with:Blue Color with20inch border

 now we are changing the Diameter 3times
This square is colored with:Blue Color with60inch border
BUILD SUCCESSFUL (total time: 0 seconds)
```

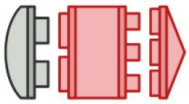
PARTICIPANTS:



PROS AND CONS

- Pros
  - Increases code re-usability
  - Reduces the duplicate code
  - Increases code maintainability
    - if we need to change something, change at one place only
  - If something breaks then it does not break everything
- Cons
  - The two processes must be separable
  - This must be planned before development

QUESTIONS!



Adapter

Provides a unified interface that allows objects with incompatible interfaces to collaborate.

Bridge

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.