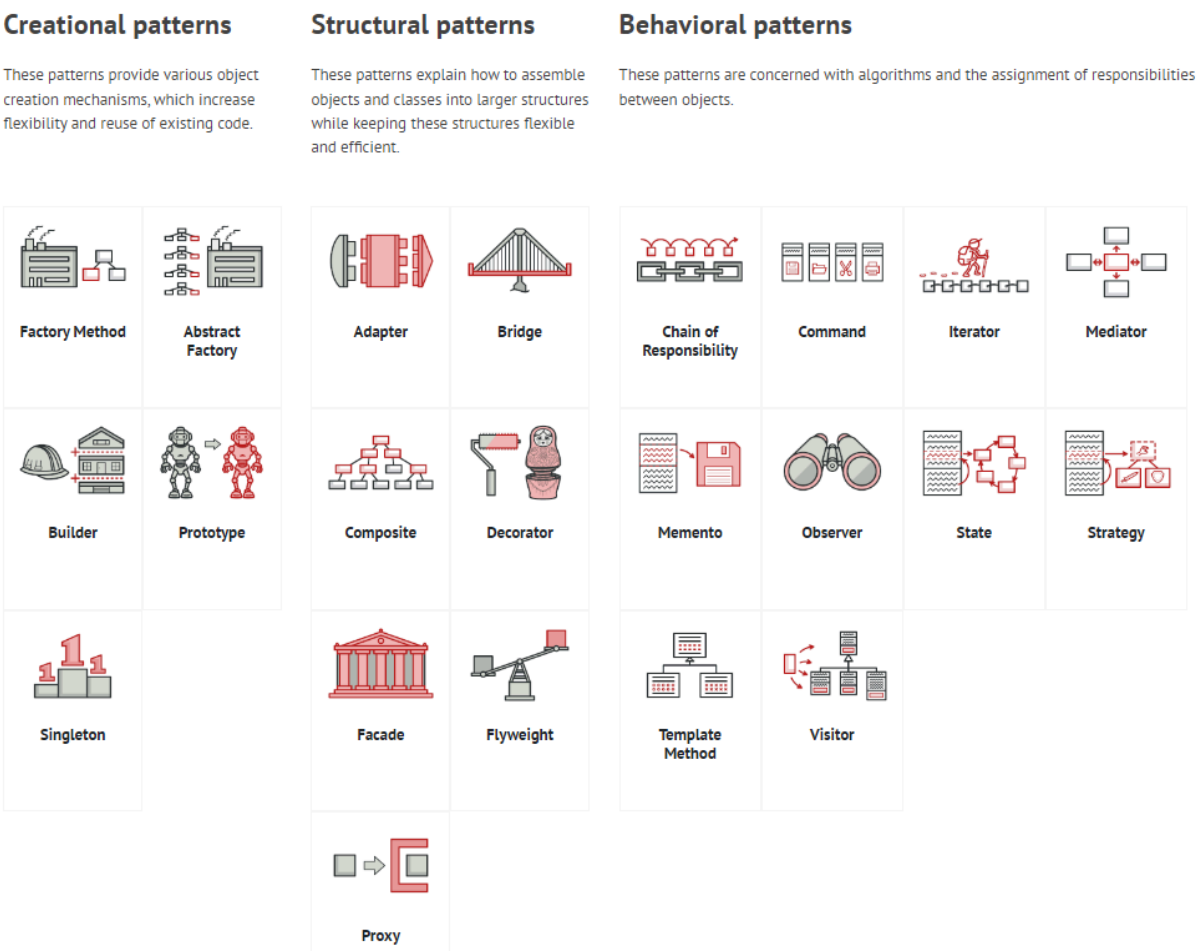
# Design Patterns

**Design Patterns:** Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

They were first introduced in the book **Design Patterns: Elements of Reusable Object-Oriented Software**, published in 1994. The book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, collectively known as **Gang of Four**. The design patterns in this book are also commonly known as **GoF design patterns**.

### Design Pattern Types



# Singleton

**Singleton** is a creational design pattern that lets you ensure that a class has only one instance while providing a global access point to this instance.

The Singleton pattern solves two problems at the same time

1. Ensure that a class has just a single instance
2. Provide a global access point to that instance

Solution

1. Make the default constructor private, to prevent other objects from using the `new` operator with the Singleton class.
2. Create a static creation method that acts as a constructor.

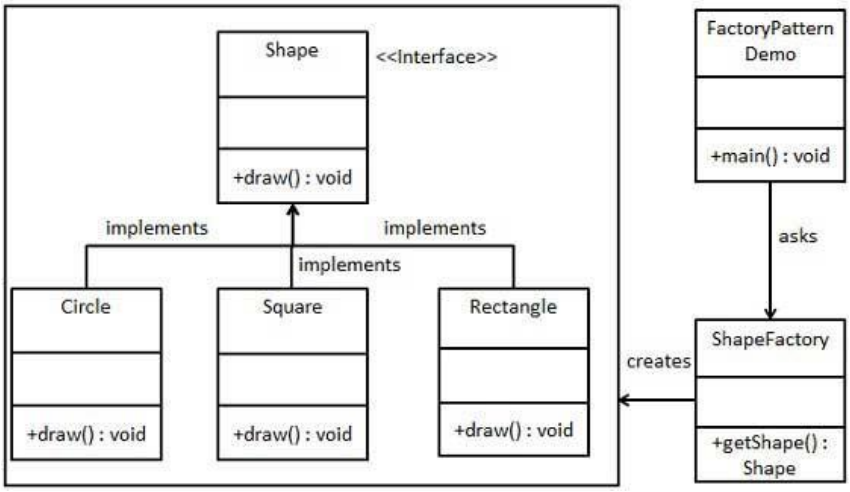
```
class Singleton {

        private static Singleton ins = null;
       private Singleton(){}
        public static Singleton getInstance()
        {
                if (ins == null)
                ins = new Singleton();

                return ins;
        }
}
```

```
class GFG {
        public static void main(String args[])
        {
                Singleton x = Singleton.getInstance();
                Singleton y = Singleton.getInstance();
                Singleton z = Singleton.getInstance();
                System.out.println("Hashcode of x is "
                                                + x.hashCode());
                System.out.println("Hashcode of y is "
                                                + y.hashCode());
                System.out.println("Hashcode of z is "
                                                + z.hashCode());
        }
}
```

**Output**:
Hashcode of x is 558638686
Hashcode of y is 558638686
Hashcode of z is 558638686

# Factory

The Factory Design Pattern is a creational design pattern that provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created.




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

```
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 ShapeFactory {
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }

      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

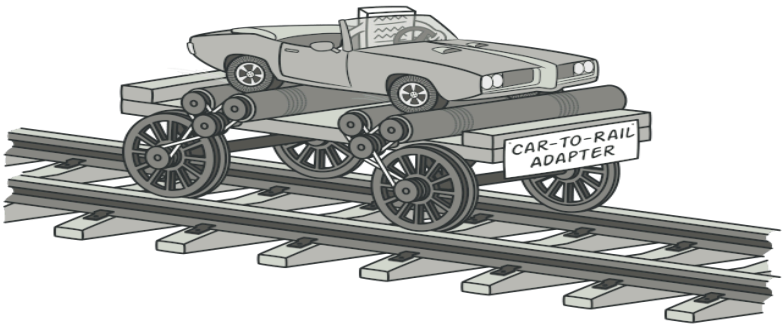
```
public class FactoryPatternDemo {

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

**Output**

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

# Adapter

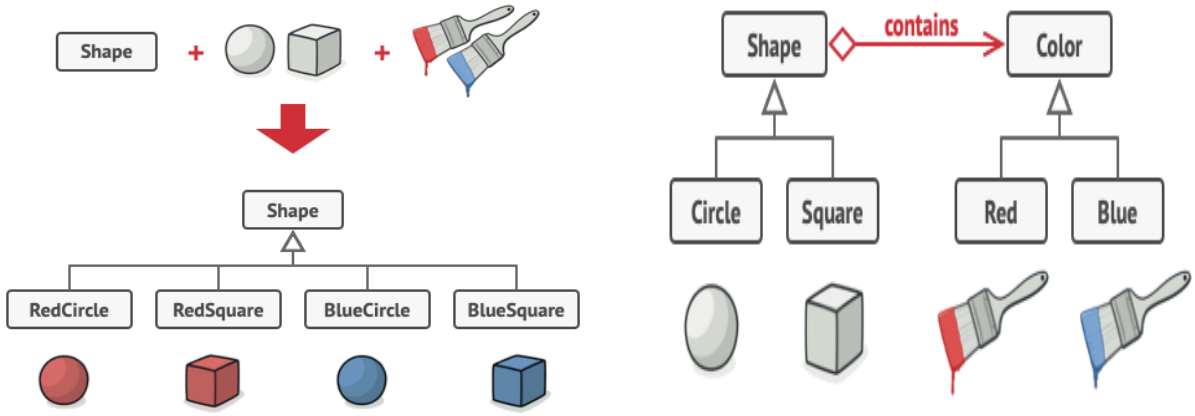
The adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.




```
// Target
interface Railroad{
    void vehicleMoving();
}
// Adaptee (the class we want to adapt)
class Car {
    public void Drive() {// drive the car}
    }
}
// Adapter class implementing the Target interface
class Adapter implements Target {
    private Car car;
    public Adapter(Car car) {this.car = car;}
    @Override
    public void vehicleMoving() {car.drive();}
}
// Client code
public class AdapterPatternExample {
    public static void main(String[] args) {
        Car car= new Car();
        Railroad railroad = new Adapter(car);
        railroad.vehiclemoving();
    }
}
```

# Bridge

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.

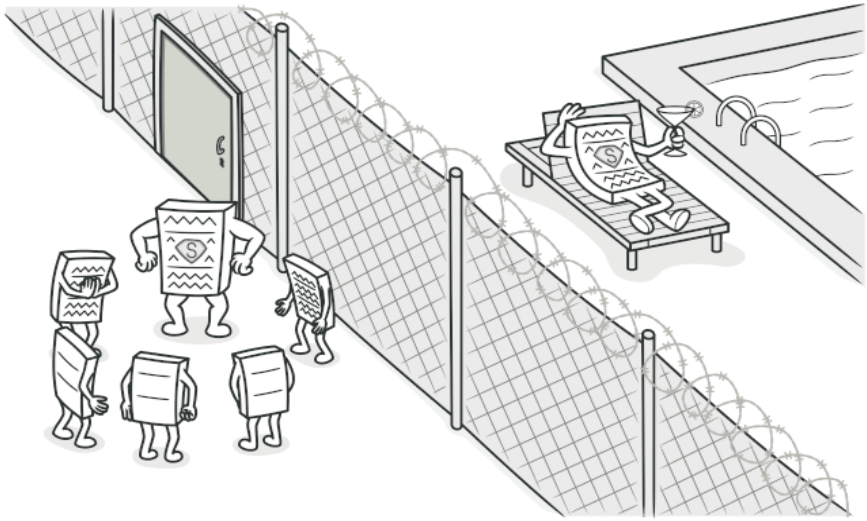



```
// Implementor: Color
interface Color {
    String applyColor();
}
// Concrete Implementor: RedColor
class RedColor implements Color {
    @Override
    public String applyColor() {return "Red";}
}
// Concrete Implementor: BlueColor
class BlueColor implements Color {
    @Override
    public String applyColor() {return "blue";}
}
// Abstraction: Shape
abstract class Shape {
    protected Color color;
    public Shape(Color color) {this.color = color;}
    abstract String draw();
}
// Concrete Abstraction: Circle
class Circle extends Shape {
    public Circle(Color color) { super(color);}
    @Override
    String draw() {
```

```
        return "Drawing a Circle with color " + color.applyColor();
    }
}

// Concrete Abstraction: Square
class Square extends Shape {
    public Square(Color color) {
        super(color);
    }

    @Override
    String draw() {
        return "Drawing a Square with color " + color.applyColor();
    }
}

// Usage
public class BridgePatternExample {
    public static void main(String[] args) {
        Color redColor = new RedColor();
        Color blueColor = new BlueColor();

        Circle circle = new Circle(redColor);
        Square square = new Square(blueColor);

        System.out.println(circle.draw());  // Output: Drawing a Circle with
color Red
        System.out.println(square.draw());  // Output: Drawing a Square with
color Blue
    }
}
```

# Proxy

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object.



```
public interface Image {
   void display();
}
public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
```

```
}
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      //image will be loaded from disk
      image.display();
      System.out.println("");

      //image will not be loaded from disk
      image.display();
   }
}
```

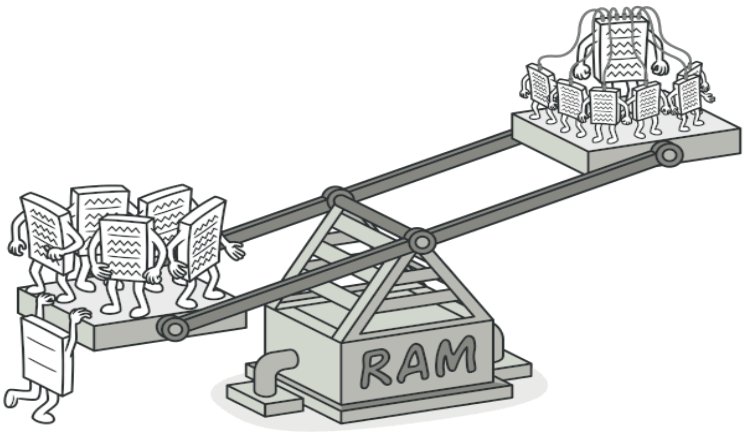
**Output**

Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg

# Flyweight

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of the state between multiple objects instead of keeping all of the data in each object.




```
interface Shape {
    void draw();
}                          bullet
class Circle implements Shape {
    private String color;

    public Circle(String color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.println("Drawing Circle with color: " + color);
    }
}
// Flyweight Factory
class ShapeFactory {
    private static final Map<String, Shape> circleMap = new HashMap<>();
```

```
    public static Shape getCircle(String color) {
        Circle circle = (Circle) circleMap.get(color);

        if (circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Creating new Circle with color: " + color);
        }

        else {
             System.out.println("Returning the already existing circle of
color: " + color);

        }
         return circle;
    }
}

// Client
public class FlyweightPatternExample {
    public static void main(String[] args) {
        // Using the flyweight factory to get circles with different colors
        Shape redCircle = ShapeFactory.getCircle("Red");
        Shape greenCircle = ShapeFactory.getCircle("Green");
        Shape blueCircle = ShapeFactory.getCircle("Blue");
        Shape redCircleAgain = ShapeFactory.getCircle("Red"); // Reusing
    }
}
```

**Output**:
Creating new Circle with color: Red
Creating new Circle with color: Green
Creating new Circle with color: Blue
Returning the already existing circle of color: Red