What is a Design Pattern?

Design patterns are well-established solutions to common problems in software design. Think of them as pre-designed blueprints or templates that you can adapt to solve frequent design challenges in your programs.

Explaination:

1. **Blueprints, Not Code**: A design pattern is not a specific piece of code you can just copy and paste. Instead, it's a general idea or concept for addressing a specific design issue. You take the pattern's concept and implement it in a way that fits your own project's needs.
2. **Not the Same as Algorithms**: Patterns are often mixed up with algorithms because both address common problems. However, an algorithm is a specific set of steps to achieve a result, much like a recipe with clear instructions. In contrast, a pattern is more like a blueprint: it shows what the end result should look like and how different parts are related, but you decide the exact details of how to implement it in your code.

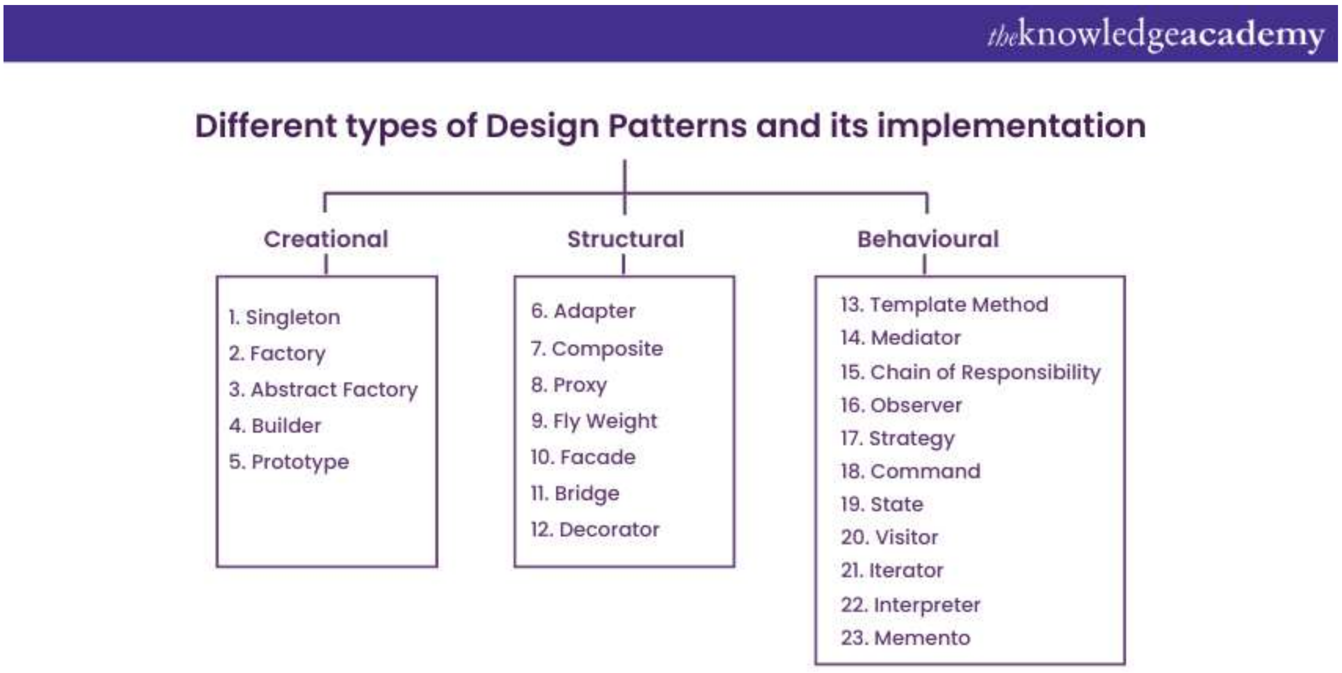
What Does a Design Pattern Include?

When a design pattern is described, it usually includes several key sections to make it clear and easy to use:

1. **Intent**: This section explains the purpose of the pattern. It briefly describes the problem it solves and how it provides a solution.
2. **Motivation**: This part goes deeper into the problem and the solution. It provides more details on why the pattern is needed and how it effectively addresses the issue.
3. **Structure**: This section shows the components of the pattern and how they interact with each other. It's often illustrated with diagrams to help visualize how different parts of the pattern fit together.
4. **Code Example**: To make the pattern easier to understand, a code example in a popular programming language is provided. This

example demonstrates how the pattern can be implemented in code.

In summary, design patterns are like blueprints for solving common design problems. They help you understand how to approach a problem and provide a flexible guide for creating solutions in your software.



Singleton Design Pattern

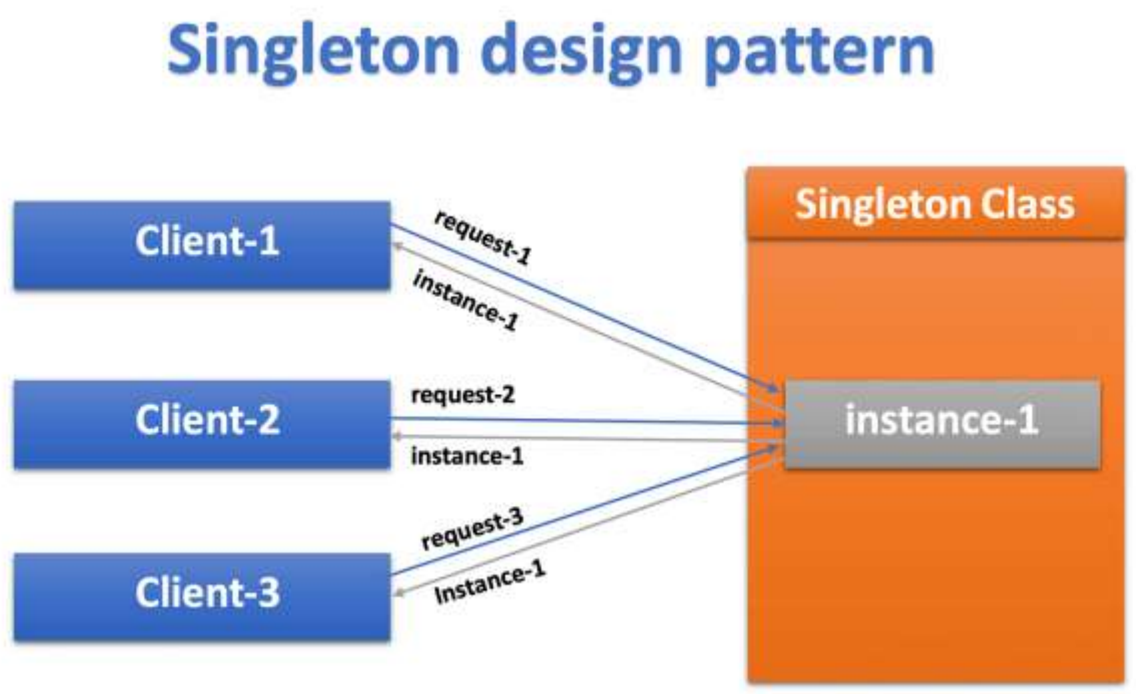
Category-Creational

The **Singleton Design Pattern** ensures that a class has only one instance and provides a global point of access to it. Here is a simple example of how to implement the Singleton pattern in Java:

Step-by-Step Implementation

1. **Private Constructor**: Prevents other classes from instantiating the Singleton class.
2. **Static Field**: Holds the single instance of the Singleton class.

3. **Static Method**: Provides the global point of access to the instance.



Example Code

```
package com.mphasis.dp.singleton;

public class Singleton {
    // Step 1: Create a private static instance of the class
    private static Singleton instance;

    // Step 2: Make the constructor private so that this class 
cannot be
    // instantiated from outside this class

    private Singleton() {
        // Initialize any resources if needed
    }

    // Step 3: Provide a public static method to get the 
instance of the class
    public static Singleton getInstance() {
        if (instance == null) {
            // Create a new instance if it doesn't exist
            instance = new Singleton();
        }
        return instance;
    }

    // Example method
```

```java
    public void showMessage() {
        System.out.println("Hello, I am a Singleton!");
    }

    public static void main(String[] args) {
        // Get the only object available
        Singleton singleton = Singleton.getInstance();

        // Show the message
        singleton.showMessage();
    }
}
```

Explanation

1. **Private Static Instance**: The instance variable holds the single instance of the Singleton class. It is static so that it belongs to the class itself, not to any instance.
2. **Private Constructor**: The constructor is private to prevent instantiation of the class from outside.
3. **Public Static Method**: The getInstance method provides a way to get the single instance of the class. If the instance does not exist, it creates one; otherwise, it returns the existing instance.
4. **Usage in Main Method**: In the main method, we obtain the instance of the Singleton class using getInstance and then call the showMessage method to demonstrate the functionality.

This pattern ensures that there is only one instance of the Singleton class throughout the application's lifecycle.

===

2. Factory Design Pattern

Category- creational

2. **The Factory Design Pattern is a creational pattern** that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. This pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code. The client code works with interfaces or abstract classes rather than concrete implementations.

Step-by-Step Implementation

1. **Product Interface**: Defines the interface of objects the factory method creates.
2. **Concrete Product Classes**: Implement the Product interface.
3. **Factory Class**: Contains a method to create objects.

Example Code

Let's create an example of a simple factory that creates different types of shapes.

1. Product Interface

```java
package com.mphasis.factorypattern;
// Product Interface
public interface Shape {
    void draw();

}
```

Product Classes

```java
package com.mphasis.factorypattern;
// Concrete Product Class 1
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
```

// Concrete Product Class 2

```java
package com.mphasis.factorypattern;
// Product Interface
public interface Shape {
    void draw();
}
```

// Concrete Product Class 3

```java
package com.mphasis.factorypattern;
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }

}
```

3. Factory Class

```
package com.mphasis.factorypattern;
// Factory Class
public class ShapeFactory {

    // Factory method to create and return different shapes
    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;
    }
}
```

4. Client Code

```
package com.mphasis.factorypattern;
// Client Code
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new
ShapeFactory();

        // Get an object of Circle and call its draw
method
        Shape shape1 =
shapeFactory.getShape("CIRCLE");
        shape1.draw();

        // Get an object of Rectangle and call its
draw method
        Shape shape2 =
shapeFactory.getShape("RECTANGLE");
        shape2.draw();
        // Get an object of Square and call its draw
method
        Shape shape3 =
shapeFactory.getShape("SQUARE");
        shape3.draw();
    }
}
```

```
+----------------------+
|       Shape          |
|----------------------|
| + draw(): void       |
+----------------------+
             ^
             |
  +-----------+-----------+
  |           |           |
+------------+ +------------+ +------------+
|   Circle   | | Rectangle  | |   Square   |
|------------| |------------| |------------|
| + draw()   | | + draw()   | | + draw()   |
+------------+ +------------+ +------------+
```

```
   +----------------------+
   |    ShapeFactory      |
   |----------------------|
   | + getShape(type:     |
   |     String): Shape   |
   +----------------------+
          ^
          |
   +----------------------+
   | FactoryPatternDemo   |
   |----------------------|
   | + main(args:         |
   |     String[]): void  |
   +----------------------+
```

Explanation

1. **Product Interface**: Shape is the interface with a method draw() that all shapes must implement.
2. **Concrete Product Classes**: Circle, Rectangle, and Square are concrete classes that implement the Shape interface.
3. **Factory Class**: ShapeFactory contains a method getShape(String shapeType) which returns an instance of a Shape based on the provided type.
4. **Client Code**: FactoryPatternDemo demonstrates the usage of the factory. It uses ShapeFactory to get instances of different shapes and calls their draw methods.

By using the Factory Design Pattern, the client code (main method) does not need to know the specifics of how the shapes are created and can work with any shape type by interacting with the Shape interface. This makes the code more flexible and easier to extend.

Benefits of Using the Factory Design Pattern

1. **Encapsulation of Object Creation**:
    - The factory method encapsulates the creation logic for different shapes. The client code does not need to know how the shapes are created, only how to request them.
2. **Code Flexibility and Maintainability**:
    - If you need to add a new shape, you only need to add a new class that implements the Shape interface and modify the ShapeFactory class. The client code does not need to change.
3. **Loose Coupling**:
    - The client code is decoupled from the specific classes of shapes it uses. It only depends on the Shape interface and the ShapeFactory.

4. **Easier Testing**:
    - It becomes easier to test the client code because you can pass mock or stub implementations of the Shape interface if needed.

By using the Factory Design Pattern, you make your code more modular, easier to understand, extend, and maintain. This pattern is particularly useful when the exact types and dependencies of objects need to be specified at runtime.

Abstract Factory Design Pattern

Category- **Creational**

Concept

The **Abstract Factory Design Pattern** is used when you need to create families of related objects without specifying their exact types. Think of it as a factory for factories.

Example Scenario

Imagine you are making a drawing application that can draw shapes. You want to support different types of shapes (e.g., circles and squares), and you might want to draw them in different styles (e.g., simple or fancy). The Abstract Factory Pattern helps you create the shapes without worrying about which specific class you are using.

Key Components

1. **Abstract Factory**: An interface for creating abstract products.
2. **Concrete Factory**: Implements the Abstract Factory to create specific products.
3. **Abstract Product**: An interface for a type of product.
4. **Concrete Product**: Implements the Abstract Product interface.
5. **Client**: Uses the Abstract Factory to create products.

1. //Abstract Factory

```java
package com.mphasis.abstractfactory;
// Abstract Factory for creating shapes
interface ShapeFactory {
    Circle createCircle();
    Square createSquare();
}
```

2. //Concrete Factory

```java
package com.mphasis.abstractfactory;
// Concrete Factory for creating simple shapes
class SimpleShapeFactory implements ShapeFactory {
    public Circle createCircle() {
        return new SimpleCircle();
    }

    public Square createSquare() {
        return new SimpleSquare();
    }
}

// Concrete Factory for creating fancy shapes
class FancyShapeFactory implements ShapeFactory {
    public Circle createCircle() {
        return new FancyCircle();
    }

    public Square createSquare() {
        return new FancySquare();
    }
}
```

3// Abstract Product

```java
package com.mphasis.abstractfactory;
// Abstract Product for Circle
interface Circle {
    void draw();
}

// Abstract Product for Square
interface Square {
    void draw();
}
```

4// Concrete Product

```java
package com.mphasis.abstractfactory;
// Concrete Product for a simple circle
class SimpleCircle implements Circle {
    public void draw() {
        System.out.println("Drawing a simple circle.");
    }
}

// Concrete Product for a simple square
class SimpleSquare implements Square {
    public void draw() {
        System.out.println("Drawing a simple square.");
    }
}

// Concrete Product for a fancy circle
class FancyCircle implements Circle {
    public void draw() {
        System.out.println("Drawing a fancy circle.");
    }
}

// Concrete Product for a fancy square
class FancySquare implements Square {
    public void draw() {
        System.out.println("Drawing a fancy square.");
    }
}
```

5// Client Code

```java
package com.mphasis.abstractfactory;
public class DrawingApp {
    public static void main(String[] args) {
        // Choose a factory
        ShapeFactory factory = new
SimpleShapeFactory();

        // Use the factory to create shapes
        Circle circle = factory.createCircle();
        Square square = factory.createSquare();

        // Draw the shapes
        circle.draw();
        square.draw();
    }
}
```

1. **Abstract Factory (ShapeFactory)**:
    - An interface with methods for creating abstract products (shapes like circles and squares).
2. **Concrete Factory (SimpleShapeFactory, FancyShapeFactory)**:
    - Classes that implement the ShapeFactory interface to create specific types of shapes (simple or fancy).
3. **Abstract Product (Circle, Square)**:
    - Interfaces for different types of products (shapes).
4. **Concrete Product (SimpleCircle, SimpleSquare, FancyCircle, FancySquare)**:
    - Classes that implement the product interfaces to create specific shapes.
5. **Client (DrawingApp)**:
    - The client code that uses the ShapeFactory to create and draw shapes. It doesn't need to know the exact class names of the shapes, making the code flexible.

Benefits

1. **Flexibility**: Easily switch between different families of products (simple vs. fancy shapes) without changing the client code.
2. **Decoupling**: The client code is decoupled from the actual product classes, making it easier to maintain and extend.

This basic example demonstrates how the Abstract Factory Design Pattern helps manage the creation of related objects while keeping the code flexible and decoupled.

==

Builder Design Pattern

Category- creational

The **Builder Design Pattern is a creational pattern** used to construct complex objects step by step. It allows you to produce different types and representations of an object using the same construction process. The pattern is especially useful when an object has a large number of optional parameters or when the construction process involves several steps.

1. Product (`Phone`)

```java
package com.mphasis.builder;
public class Phone {
    // Required parameters
    private final String brand;
    private final String model;

    // Optional parameters
    private final int batteryLife;
    private final int cameraResolution;

    // Private constructor to ensure object creation
through the builder only
    Phone(PhoneBuilder builder) {
        this.brand = builder.brand;
        this.model = builder.model;
        this.batteryLife = builder.batteryLife;
        this.cameraResolution =
builder.cameraResolution;
    }

    @Override
    public String toString() {
        return "Phone [Brand=" + brand + ", Model=" +
model +
                    ", Battery Life=" + batteryLife +
"mAh, Camera Resolution=" + cameraResolution + "MP]";
    }
}
```

2. Separate Builder Class (`PhoneBuilder`)

```
. package com.mphasis.builder;
public class PhoneBuilder {
    // Required parameters
    public final String brand;
    public final String model;

    // Optional parameters with default values
    public int batteryLife = 0;
    public int cameraResolution = 0;

    // Constructor for required parameters
    public PhoneBuilder(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

    // Methods for setting optional parameters
    public PhoneBuilder setBatteryLife(int
batteryLife) {
        this.batteryLife = batteryLife;
        return this;
    }

    public PhoneBuilder setCameraResolution(int
cameraResolution) {
        this.cameraResolution = cameraResolution;
        return this;
    }

    // Build method to create the Phone object
    public Phone build() {
        return new Phone(this);
    }
}
```

3. Client Code

```
package com.mphasis.builder;
```

```
public class BuilderPatternDemo {
    public static void main(String[] args) {
        // Create a Phone with only required
parameters
        Phone phone1 = new PhoneBuilder("Samsung",
"Galaxy S21").build();

        // Create a Phone with both required and
optional parameters
        Phone phone2 = new PhoneBuilder("Apple",
"iPhone 13")
                .setBatteryLife(4000)
                .setCameraResolution(12)
                .build();

        // Print the phones
        System.out.println(phone1);
        System.out.println(phone2);
    }
}
```

1. **Product (Phone)**:

    - This is the class we want to create.
    - It has a private constructor to ensure it can only be created through the builder.

2. **Builder (PhoneBuilder)**:

    - A separate class to help build Phone objects.
    - It has required parameters (brand and model) which are set through the constructor.
    - It has optional parameters (batteryLife and cameraResolution) with default values.
    - Methods for setting optional parameters return this to allow chaining.
    - The build() method creates a Phone object using the builder.

3. **Client Code**:

- Demonstrates how to create Phone objects using the builder.
- Shows how to create a Phone with just required parameters and another Phone with both required and optional parameters.

Benefits

1. **Easy to Read and Understand**: The builder pattern makes the code more readable and understandable by clearly showing what parameters are being set.
2. **Immutable Objects**: The created Phone objects are immutable because all fields are set during construction, and there are no setters.
3. **Flexible**: You can easily add more optional parameters without changing existing client code.

By using this simplified version of the Builder Design Pattern without a static nested class, you can still achieve the benefits of the pattern in a clear and flexible way.

Template Method Design Pattern

Category- **Behavioral**

The Template Method Design Pattern is a behavioral pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This pattern lets subclasses redefine certain steps of an algorithm without changing its structure.

Components of Template Method Pattern

1. **Abstract Class**: Contains the template method which defines the algorithm's structure and the abstract methods to be implemented by subclasses.

2. **Concrete Classes**: Implement the abstract methods defined in the abstract class.

Example Code

Let's consider an example where we have an abstract class Game that defines the skeleton of playing a game, and concrete classes Cricket and Football that implement the steps defined in Game.

1. Abstract Class

```
package com.mphasis.template;
abstract class Game {
    // Template method
    public final void play() {
        initialize();
        startPlay();
        endPlay();
    }

    // Abstract methods to be implemented by subclasses
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

}
```

2. Concrete Classes

```
package com.mphasis.template;
class Chess extends Game {
    void initialize() {
        System.out.println("Chess Game Initialized! Start 
playing.");
    }

    void startPlay() {
        System.out.println("Chess Game Started. Enjoy the 
game!");
    }

    void endPlay() {
        System.out.println("Chess Game Finished!");
    }
}

class Football extends Game {
    void initialize() {
        System.out.println("Football Game Initialized! Start 
playing.");
```

```
    }

    void startPlay() {
        System.out.println("Football Game Started. Enjoy the
game!");
    }

    void endPlay() {
        System.out.println("Football Game Finished!");
    }
}
```

3. *Client Code*

```
package com.mphasis.template;
public class TemplateMethodPatternDemo {
    public static void main(String[] args) {
        Game game = new Chess();
        game.play(); // Playing Chess

             System.out.println();

             game = new Football();
             game.play(); // Playing Football
         }
     }
```

Explanation

1. **Abstract Class (Game)**:

    - This class defines a template method (play()) that outlines the steps of the algorithm.
    - It has three abstract methods (initialize(), startPlay(), and endPlay()) that need to be implemented by subclasses.

2. **Concrete Classes (Chess, Football)**:

    - These classes extend the Game class and provide specific implementations for the abstract methods.
    - Each class implements the steps in a way that is appropriate for the specific game.

3. **Client Code (TemplateMethodPatternDemo)**:

- The client code creates instances of the concrete classes and calls the template method (play()).
- The template method executes the algorithm, calling the concrete implementations of the steps as defined in the concrete classes.

Benefits

1. **Code Reuse**: Common code for the algorithm is in the base class, reducing duplication.
2. **Flexibility**: Subclasses can provide different implementations for specific steps, allowing for easy customization.
3. **Consistency**: The overall structure of the algorithm is maintained, ensuring that all subclasses follow the same process.

The Template Method Design Pattern is useful when you have a consistent process that can be customized by subclasses, ensuring that the overall structure remains the same while allowing for flexibility in the details.

This pattern ensures that the algorithm's structure remains unchanged while allowing subclasses to provide specific implementations for certain steps. It promotes code reuse and avoids code duplication.

Bridge Design Pattern

Category: Structural

The **Bridge Design Pattern** is a **structural pattern** that decouples an abstraction from its implementation so that the two can vary independently. This pattern is useful when both the abstraction and the implementation can have multiple variations and need to be extended independently.

Components of Bridge Pattern

1. **Abstraction**: Defines the abstract interface and maintains a reference to an implementer.
2. **Refined Abstraction**: Extends the abstraction interface.

3. **Implementor**: Defines the interface for implementation classes.
4. **Concrete Implementor**: Implements the Implementor interface.

Example Code

Let's consider an example where we have different types of shapes (Shape) that can be drawn with different drawing APIs (DrawingAPI).

1. Implementor Interface

```java
package com.mphasis.bridge;
// Implementer
interface Device {
    void turnOn();
    void turnOff();
    void setVolume(int volume);
}
```

2. Concrete Implementors

```java
package com.mphasis.bridge;
// Concrete Implementer for TV
class TV implements Device {
    private int volume;

    public void turnOn() {
        System.out.println("TV is turned ON");
    }

    public void turnOff() {
        System.out.println("TV is turned OFF");
    }

    public void setVolume(int volume) {
        this.volume = volume;
        System.out.println("TV volume set to " + volume);
    }
}

// Concrete Implementer for Radio
class Radio implements Device {
    private int volume;

    public void turnOn() {
        System.out.println("Radio is turned ON");
    }

    public void turnOff() {
        System.out.println("Radio is turned OFF");
    }
    public void setVolume(int volume) {
        this.volume = volume;
        System.out.println("Radio volume set to " + volume);
    }
}
```

3. Abstraction

```java
package com.mphasis.bridge;
// Abstraction
abstract class RemoteControl {
    protected Device device;

    protected RemoteControl(Device device) {
        this.device = device;
    }

    abstract void turnOn();
    abstract void turnOff();
    abstract void setVolume(int volume);
}
```

4. Refined Abstraction

```java
package com.mphasis.bridge;
// Refined Abstraction
class BasicRemote extends RemoteControl {

    public BasicRemote(Device device) {
        super(device);
    }

    public void turnOn() {
        device.turnOn();
    }

    public void turnOff() {
        device.turnOff();
    }

    public void setVolume(int volume) {
        device.setVolume(volume);
    }
}
```

5. Client Code

```java
package com.mphasis.bridge;
public class BridgePatternDemo {
    public static void main(String[] args) {
        Device tv = new TV();
            RemoteControl tvRemote = new BasicRemote(tv);

            tvRemote.turnOn();
            tvRemote.setVolume(10);
            tvRemote.turnOff();

            System.out.println();

            Device radio = new Radio();
            RemoteControl radioRemote = new BasicRemote(radio);

            radioRemote.turnOn();
            radioRemote.setVolume(20);
            radioRemote.turnOff();
        }
    }
```

1. **Implementer (Device)**:
   - This is an interface that declares methods for turning a device on and off and setting the volume.
2. **Concrete Implementer (TV, Radio)**:
   - These classes implement the Device interface and provide specific implementations for a TV and a Radio.
3. **Abstraction (RemoteControl)**:
   - This abstract class contains a reference to a Device object and declares methods to control the device. The RemoteControl class does not implement the methods directly but delegates them to the Device.
4. **Refined Abstraction (BasicRemote)**:
   - This class extends RemoteControl and provides specific implementations for controlling the device. It delegates the operations to the Device object.
5. **Client Code (BridgePatternDemo)**:
   - The client code creates instances of TV and Radio (Concrete Implementers) and controls them using the BasicRemote (Refined Abstraction).

Benefits

1. **Decoupling**: Separates the abstraction from its implementation, allowing them to vary independently.
2. **Flexibility**: Easily switch between different implementations without changing the abstraction code.
3. **Scalability**: Add new abstractions and implementations independently without modifying existing code.

The Bridge Design Pattern is particularly useful when you want to avoid a permanent binding between an abstraction and its implementation, allowing both to evolve independently and promoting greater flexibility and scalability.

==

Proxy Design Pattern

Category: **Structural**

The Proxy Design Pattern is a structural pattern that provides a surrogate or placeholder for another object to control access to it. This pattern is useful in scenarios such as lazy initialization, access control, logging, and more. The Proxy Design Pattern falls under the **Structural Design Patterns** category

Types of Proxy

1. **Virtual Proxy**: Controls access to a resource that is expensive to create.
2. **Protection Proxy**: Controls access to a resource based on permissions.
3. **Remote Proxy**: Represents an object in a different address space.
4. **Smart Proxy**: Adds additional actions when an object is accessed.

Key Components

1. **Subject**: An interface or abstract class that declares the methods that the real object and proxy will implement.

2. **RealSubject**: The actual object that the proxy represents. It implements the Subject interface and contains the real business logic.
3. **Proxy**: Implements the Subject interface and holds a reference to the RealSubject. It can add additional functionality before or after forwarding requests to the RealSubject.

Example Code

Let's consider an example of a virtual proxy where we have an Image interface, and a RealImage class that loads and displays an image. The ProxyImage class controls access to the RealImage object and loads it only when needed.

1. Subject Interface

```
package com.mphasis.proxy;
// Subject
interface Document {
    void display();
}
```

2. Real Subject

```
package com.mphasis.proxy;
// RealSubject
class RealDocument implements Document {
    private String content;

    public RealDocument(String content) {
        this.content = content;
        loadFromDisk(); // Simulate loading a document from 
disk
    }

    private void loadFromDisk() {
        System.out.println("Loading document from disk: " + 
content);
    }

    public void display() {
        System.out.println("Displaying document: " + content);
    }
}
```

3. Proxy

```java
package com.mphasis.proxy;
// Proxy
class ProxyDocument implements Document {
    private RealDocument realDocument;
    private String content;

    public ProxyDocument(String content) {
        this.content = content;
    }

    public void display() {
        if (realDocument == null) {
            realDocument = new RealDocument(content); // Load the real document on demand
        }
        realDocument.display(); // Forward the request to the real document
    }
}
```

4. Client Code

```java
package com.mphasis.proxy;
public class ProxyPatternDemo {
    public static void main(String[] args) {
        Document document = new ProxyDocument("My Document");

            // The document is not loaded yet
            System.out.println("ProxyDocument created. Document not loaded yet.");

            // Displaying the document will load it from disk
            document.display(); // This will load the document and then display it
        }
}
```

Explanation

1. **Subject (Document)**:

    - This interface declares the methods that both the real document and the proxy will implement. In this case, it has a display() method.

2. **RealSubject (RealDocument)**:

- This class implements the Document interface and provides the actual implementation. It simulates loading a document from disk and displaying it.

3. **Proxy (ProxyDocument)**:

    - This class also implements the Document interface and holds a reference to a RealDocument. It controls access to the real document. For instance, it only creates the RealDocument object when it's actually needed (lazy initialization).

4. **Client Code (ProxyPatternDemo)**:

    - The client code interacts with the ProxyDocument instead of the RealDocument. The proxy manages the creation and access to the real document, demonstrating lazy initialization.

Benefits

1. **Lazy Initialization**: The real object is created only when it's needed, which can save resources and time.
2. **Access Control**: The proxy can control access to the real object, adding additional security or access control logic.
3. **Additional Functionality**: The proxy can add extra functionality such as logging, caching, or validation without changing the real object.

The Proxy Design Pattern is useful when you want to control access to an object, perform operations before or after accessing the real object, or manage resource usage more efficiently. It helps in separating concerns and adding flexibility to the object management process.

.

In this example, the proxy pattern provides a way to delay the loading of an image until it is actually needed, which can save resources if the image is never displayed.

Immutable classes

Category: Design Principles

Creating **immutable classes** in Java involves designing classes whose instances cannot be modified once they are created. This is a common practice to ensure thread safety and simplicity in concurrent programming. Immutable objects are inherently thread-safe since their state cannot change after construction.

Key Principles for Creating Immutable Classes

1. **Declare the Class as final**: This prevents subclasses from altering immutability.
2. **Make All Fields private and final**: This ensures that fields are only assigned once.
3. **Do Not Provide "Setter" Methods**: This ensures that fields cannot be modified after object creation.
4. **Initialize All Fields via Constructor**: Ensure that all fields are set during object creation.
5. **Ensure Deep Copies for Mutable Fields**: If the class has fields that refer to mutable objects, ensure to make copies of these objects to prevent modifications from the outside.

Example Code

Let's create an immutable class Person.

```
package com.mphasis.immutable;
public final class Person {
    private final String name;
    private final int age;

    // Constructor to initialize the fields
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter for name
    public String getName() {
        return name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }
}
```

// Mutable Address class

```
package com.mphasis.immutable;
public class Main {
    public static void main(String[] args) {
        // Creating an immutable Person object
        Person person1 = new Person("Vijay", 30);
        Person person2 = new Person("Shweta", 25);

        // Accessing the fields using getter methods
        System.out.println("Person 1: " + person1.getName() + ",
Age: " + person1.getAge());
        System.out.println("Person 2: " + person2.getName() + ",
Age: " + person2.getAge());

        // Attempting to change the state would require creating a
new object
        // since the existing objects are immutable
        Person person3 = new Person("Venkat", 31); // New object
with updated age

        // Displaying the new object
        System.out.println("Updated Person 1: " + person3.getName()
+ ", Age: " + person3.getAge());
    }
}
```

Explanation

1. **Person Class Declaration**:

    - Declared as final to prevent subclassing.
    - All fields (name, age, address) are private and final to ensure they are only set once.

2. **Constructor**:

- Initializes all fields. For the mutable Address object, a deep copy is created to ensure the original object cannot be modified from outside.

3. **Getters**:

    - Return the field values. For the mutable Address field, the getter returns a new Address object (a copy) to prevent modification of the internal state.

4. **No Setters**:

    - No methods are provided to modify the fields after object creation, ensuring immutability.

Explanation

Concept of Immutability

- **Immutability** means that an object's state cannot be changed after it is created.
- In this example, Person objects are immutable because their fields are final, and no setters are provided to change their state.
- **Benefits of Immutability**:
    - **Thread Safety**: Immutable objects are inherently thread-safe because their state cannot change after they are created.
    - **Simplicity**: They simplify code by avoiding side effects and making it easier to reason about the program's behavior.
    - **Security**: They help maintain the integrity of the object's state, making it less prone to accidental or malicious modifications.

Summary

- The Main class demonstrates the creation and use of immutable Person objects.

- Modifying the state of an immutable object requires creating a new object with the desired state.
- The original Person objects remain unchanged, highlighting the benefits and characteristics of immutability in Java.