DESIGN PATTERNS

WITH Java

Priyanshu Routh

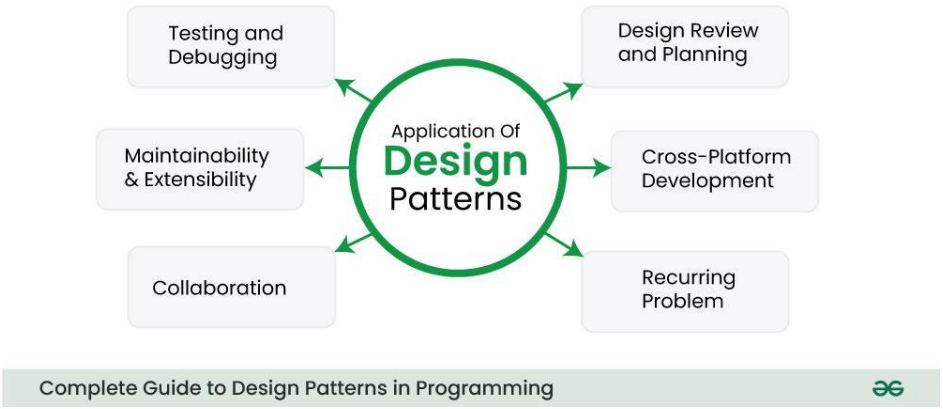
7/28/24

JAVA

DESIGN PATTERNS

Introduction to Design Patterns

Design patterns are essential tools in software engineering, providing proven solutions to common design problems. They represent best practices refined over time, enabling developers to create more maintainable, scalable, and efficient software. Understanding and applying design patterns can significantly enhance the quality and robustness of your code.



What Are Design Patterns?

Design patterns are typical solutions to common problems in software design. They are like blueprints that you can customize to solve a particular design problem in your code. Patterns are not specific pieces of code but rather general reusable solutions that can be applied in various contexts to address recurring issues.

Benefits of Design Patterns

1. **Reusability**: Design patterns promote code reuse by providing templates for solving common problems. This reduces the need to reinvent the wheel and allows developers to leverage existing solutions.
2. **Maintainability**: By providing a clear structure and common vocabulary, design patterns make it easier to maintain and understand code. This is especially beneficial in large projects or teams.
3. **Scalability**: Patterns help design flexible and scalable software architectures that can adapt to changing requirements and growth over time.
4. **Efficiency**: Leveraging well-known patterns can accelerate the development process, as developers can rely on established solutions rather than creating new ones from scratch.

Categories of Design Patterns

Design patterns are generally categorized into three main types: creational, structural, and behavioral.

1. **Creational Patterns**
    - **Purpose**: Deal with object creation mechanisms, trying to create objects in a manner suitable for the situation.
    - **Examples**: Singleton, Factory Method, Abstract Factory, Builder, Prototype.
2. **Structural Patterns**
    - **Purpose**: Concerned with how classes and objects are composed to form larger structures.
    - **Examples**: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
3. **Behavioral Patterns**
    - **Purpose**: Focus on communication between objects and the delegation of responsibilities.
    - **Examples**: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

1. Creational Design Patterns

These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

- **Singleton**: Ensures that a class has only one instance and provides a global point of access to it.
- **Factory Method**: Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.
- **Abstract Factory**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder**: Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.
- **Prototype**: Specifies the kind of objects to create using a prototypical instance and creates new objects by copying this prototype.

2. Structural Design Patterns

These patterns deal with object composition or the structure of classes and objects to form larger structures.

- **Adapter**: Allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces.
- **Bridge**: Separates an object’s interface from its implementation, allowing the two to vary independently.

- **Composite**: Composes objects into tree-like structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.
- **Decorator**: Adds additional responsibilities to an object dynamically, without affecting other objects.
- **Facade**: Provides a simplified interface to a complex subsystem.
- **Flyweight**: Reduces the cost of creating and manipulating a large number of similar objects by sharing common parts.
- **Proxy**: Provides a surrogate or placeholder for another object to control access to it.

3. Behavioral Design Patterns

These patterns deal with object collaboration and responsibility, focusing on communication between objects.

- **Chain of Responsibility**: Passes a request along a chain of handlers. Each handler can either process the request or pass it to the next handler in the chain.
- **Command**: Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.
- **Interpreter**: Defines a grammatical representation for a language and provides an interpreter to interpret sentences in that language.
- **Iterator**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator**: Defines an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from referring to each other explicitly.
- **Memento**: Captures and externalizes an object's internal state, so the object can be restored to this state later without violating encapsulation.
- **Observer**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State**: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The strategy allows the algorithm to vary independently from clients that use it.
- **Template Method**: Defines the skeleton of an algorithm in a method, deferring some steps to subclasses without changing the algorithm's structure.
- **Visitor**: Represents an operation to be performed on elements of an object structure. It allows you to define a new operation without changing the classes of the elements on which it operates.

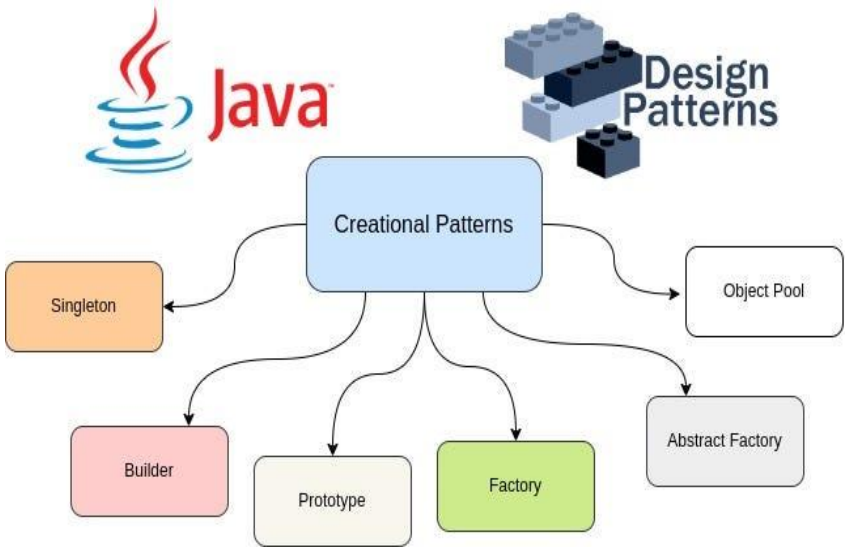
Applying Design Patterns

Applying design patterns involves understanding the problem at hand and then selecting the appropriate pattern that addresses the specific challenge. It's essential to have a solid grasp of the various patterns and their use cases to make informed decisions.

1. **Identify the Problem**: Clearly define the problem you are facing in your software design. Understand the requirements and constraints.
2. **Choose the Appropriate Pattern**: Select a design pattern that best fits the problem. Consider the pattern's intent, structure, and applicability.
3. **Implement the Pattern**: Apply the pattern to your codebase, customizing it as needed to fit your specific context.
4. **Refactor and Optimize**: Once the pattern is implemented, refactor your code to ensure it is clean, efficient, and maintainable. Test the implementation to verify it solves the problem effectively.

Creational Design Patterns

Creational design patterns are a category of design patterns in software development that deal with object creation mechanisms. They abstract the instantiation process, helping to make a system independent of how its objects are created, composed, and represented. The goal is to control object creation in a way that is suitable for the situation, often to ensure the right type of object is created or to manage the complexity of object creation.



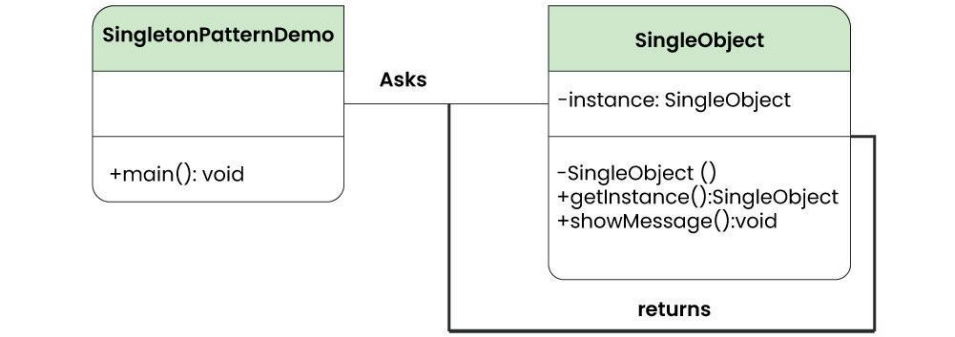
Benefits of Creational Patterns

- **Flexibility**: They provide flexibility in deciding which objects need to be created for a given instance.
- **Encapsulation**: They encapsulate the instantiation logic, which can hide complex creation logic from the client.
- **Reuse**: Patterns like Singleton and Prototype promote reuse of instances, which can be beneficial for resource management.

Creational patterns are fundamental to object-oriented design, helping to manage the complexity and scalability of creating objects in software systems.

Singleton Pattern

The Singleton pattern is one of the simplest and most commonly used design patterns in software development. It ensures that a class has only one instance and provides a global point of access to that instance. This is particularly useful when exactly one object is needed to coordinate actions across the system.



Singleton Design Pattern in Java

Key Characteristics of Singleton Pattern

1. **Single Instance**: Only one instance of the class is created.
2. **Global Access**: Provides a global point of access to the instance.
3. **Controlled Access**: The instance is created and managed within the class itself, ensuring controlled access.

When to Use Singleton Pattern

- When you need exactly one instance of a class to coordinate actions across the system.

- When you need to control access to shared resources, such as a configuration object, log file, or database connection.
- When a single instance of a class needs to be accessible from multiple places in your application.

Rules to Create a Singleton Pattern

1. **Private Constructor**:
    - The constructor of the Singleton class should be private to prevent direct instantiation from outside the class. This ensures that the only way to create an instance is through the class itself.

```
2. private Singleton() {
3.     // Private constructor
4. }
5.
```

    -
6. **Static Instance Variable**:
    - A static variable should be used to hold the single instance of the class. This static variable should be initialized to `null` (or using a lazy initialization approach) and should only be assigned a value once.

```
7. private static Singleton instance;
```

    -
8. **Public Static Method for Access**:
    - Provide a public static method that returns the single instance of the class. This method is responsible for creating the instance if it does not already exist and returning it. This method is often named `getInstance()`.

```
9. public static Logger getInstance() {
       return instance;
   }
```

10. **Thread Safety**:
    - Ensure that the Singleton pattern is thread-safe, especially in multi-threaded environments. This can be achieved through synchronization or other concurrency mechanisms.
    - **Lazy Initialization with Synchronization**:
    - **Double-Checked Locking**:
    - **Bill Pugh Singleton Design (Initialization-on-demand holder idiom)**:

11. **Safety**:
    - Ensure that the Singleton class handles serialization properly to maintain its single instance property. This can be achieved by implementing `readResolve()` in the class to prevent creation of a new instance during deserialization.
12. **Avoid Reflection**:

- Prevent the Singleton instance from being created through reflection, which can break the Singleton property. This can be achieved by throwing an exception in the constructor if an instance already exists.

13. **Lazy Initialization (if needed)**:
    - If the Singleton instance is not immediately needed, use lazy initialization to delay the creation of the instance until it is actually required. This helps in saving resources if the instance is never used.
14. **Enum Singleton (Recommended Approach)**:
    - For most cases, using an enum is the simplest and most robust way to implement the Singleton pattern. It is inherently thread-safe, ensures a single instance, and handles serialization automatically.

Implementation

The Singleton pattern can be implemented in various ways. Here are some common implementations in Java:

1. Eager Initialization

In this method, the instance is created at the time of class loading.

```
public class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {
        // Private constructor to prevent instantiation
    }

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

2. Lazy Initialization

In this method, the instance is created when it is first requested.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Private constructor to prevent instantiation
```

```
    }

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

3. Double-Checked Locking

An optimization of the lazy initialization method that reduces the overhead of acquiring a lock by first checking if the instance is already created.

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {
        // Private constructor to prevent instantiation
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

4. Bill Pugh Singleton Design

Uses a static inner helper class to create the Singleton instance. This approach leverages the class loading mechanism to ensure that the instance is created only when it is first used.

```java
public class Singleton {
    private Singleton() {
        // Private constructor to prevent instantiation
    }

    private static class SingletonHelper {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

5. Enum Singleton

An implementation using an enum, which is the simplest and most robust way to implement the Singleton pattern. It ensures thread safety and serialization handling.

```java
public enum Singleton {
    INSTANCE;

    public void someMethod() {
        // Some method implementation
    }
}
```

ava's `enum` type provides a convenient way to implement singletons. The `enum` type automatically handles the instantiation, thread-safety, and serialization of the singleton instance.

Why Use `enum` for Singleton

1. **Thread-Safety**: The enum type in Java guarantees thread-safety of the instance.
2. **Serialization**: The serialization mechanism for enums is built-in and ensures that the singleton property is maintained.
3. **Simplicity**: Enums in Java provide a simple and elegant way to implement singletons.

Advantages of Using `enum` Singleton

- **Prevents Multiple Instantiation**: The enum type inherently ensures that only one instance is created.

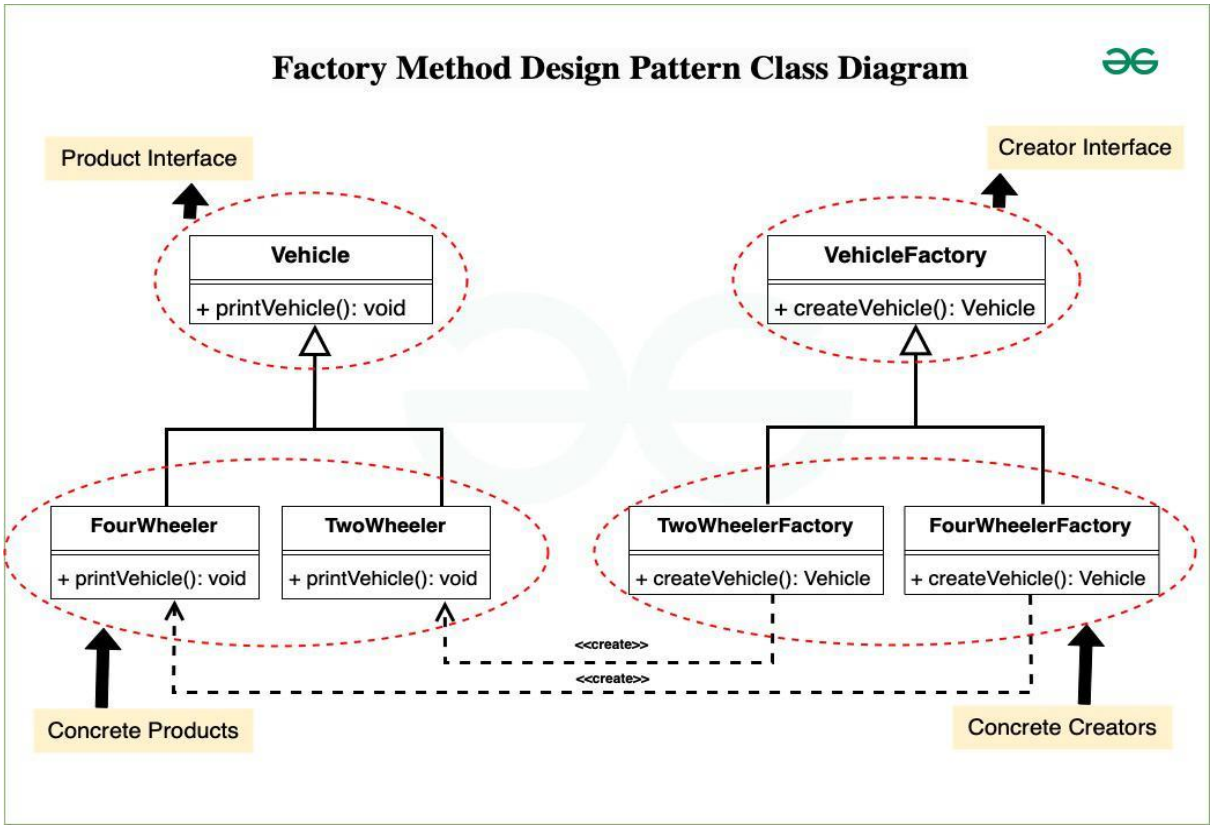
- **Thread-Safe**: Enum singletons are thread-safe by default without requiring explicit synchronization.
- **Serialization Guaranteed**: Enum instances are serialized correctly, preventing the creation of a new instance during deserialization.
- **Simple and Clear Syntax**: Using enums for singletons is straightforward and concise.

Considerations

- **Not Suitable for All Scenarios**: Enum singletons are ideal for simple singletons but may not be suitable for more complex scenarios where lazy initialization or dependency injection is required.

Factory Design Patterns

Factory Design Patterns are creational design patterns that provide a way to create objects while hiding the creation logic from the client. They are used to instantiate objects without exposing the creation logic to the client and refer to the newly created object through a common interface.



Types of Factory Design Patterns

1. **Simple Factory**
2. **Factory Method**
3. **Abstract Factory**

1. Simple Factory

The Simple Factory pattern is not a standard GoF (Gang of Four) design pattern but is widely used. It encapsulates the object creation process.

```
public interface Product {
    void use();
}

public class ConcreteProductA implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA");
    }
}

public class ConcreteProductB implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductB");
    }
}

public class SimpleFactory {
    public Product createProduct(String type) {
        if (type.equals("A")) {
            return new ConcreteProductA();
        } else if (type.equals("B")) {
            return new ConcreteProductB();
        } else {
            throw new IllegalArgumentException("Unknown product type");
        }
    }
}

public class Client {
    public static void main(String[] args) {
        SimpleFactory factory = new SimpleFactory();
        Product productA = factory.createProduct("A");
```

```
        productA.use(); // Output: Using ConcreteProductA

        Product productB = factory.createProduct("B");
        productB.use(); // Output: Using ConcreteProductB
    }
}
```

2. Factory Method

The Factory Method pattern defines an interface for creating an object but allows subclasses to alter the type of objects that will be created. It relies on inheritance: the object creation is delegated to subclasses that implement the factory method to create objects.

```
public interface Product {
    void use();
}

public class ConcreteProductA implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA");
    }
}

public class ConcreteProductB implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductB");
    }
}

public abstract class Factory {
    public abstract Product createProduct();

    public void someOperation() {
        Product product = createProduct();
        product.use();
    }
}

public class ConcreteFactoryA extends Factory {
    @Override
    public Product createProduct() {
```

```
        return new ConcreteProductA();
    }
}

public class ConcreteFactoryB extends Factory {
    @Override
    public Product createProduct() {
        return new ConcreteProductB();
    }
}

public class Client {
    public static void main(String[] args) {
        Factory factoryA = new ConcreteFactoryA();
        factoryA.someOperation(); // Output: Using ConcreteProductA

        Factory factoryB = new ConcreteFactoryB();
        factoryB.someOperation(); // Output: Using ConcreteProductB
    }
}
```

3. Abstract Factory

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It involves multiple factories each producing different sets of products.

```
public interface ProductA {
    void useA();
}

public interface ProductB {
    void useB();
}

public class ConcreteProductA1 implements ProductA {
    @Override
    public void useA() {
        System.out.println("Using ConcreteProductA1");
    }
}

public class ConcreteProductA2 implements ProductA {
    @Override
```

```
    public void useA() {
        System.out.println("Using ConcreteProductA2");
    }
}

public class ConcreteProductB1 implements ProductB {
    @Override
    public void useB() {
        System.out.println("Using ConcreteProductB1");
    }
}

public class ConcreteProductB2 implements ProductB {
    @Override
    public void useB() {
        System.out.println("Using ConcreteProductB2");
    }
}

public interface AbstractFactory {
    ProductA createProductA();
    ProductB createProductB();
}

public class ConcreteFactory1 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}

public class ConcreteFactory2 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA2();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB2();
```

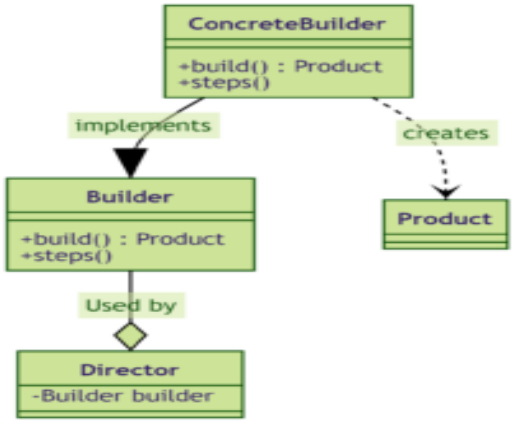
```
    }
}

public class Client {
    private ProductA productA;
    private ProductB productB;

    public Client(AbstractFactory factory) {
        productA = factory.createProductA();
        productB = factory.createProductB();
    }

    public void useProducts() {
        productA.useA();
        productB.useB();
    }

    public static void main(String[] args) {
        AbstractFactory factory1 = new ConcreteFactory1();
        Client client1 = new Client(factory1);
        client1.useProducts(); // Output: Using ConcreteProductA1 \n Using
ConcreteProductB1

        AbstractFactory factory2 = new ConcreteFactory2();
        Client client2 = new Client(factory2);
        client2.useProducts(); // Output: Using ConcreteProductA2 \n Using
ConcreteProductB2
    }
}
```

- **Simple Factory**: Encapsulates object creation in a factory method but does not conform to GoF patterns.
- **Factory Method**: Defines an interface for creating objects and lets subclasses decide which class to instantiate. It promotes extensibility.

- **Abstract Factory**: Provides an interface for creating families of related or dependent objects. It abstracts the creation process and promotes consistency among products in a family.

Builder Pattern

The Builder Pattern is a creational design pattern used to construct complex objects step by step. It allows for the creation of different representations of an object using the same construction process. The Builder Pattern is particularly useful when an object needs to be created with many optional components or when the construction process is complex and needs to be separated from the representation.



Key Characteristics

1. **Separation of Concerns**: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
2. **Incremental Construction**: Allows for step-by-step construction of an object, providing more control over the construction process.
3. **Immutability**: Can be used to create immutable objects by ensuring that once the object is built, it cannot be modified.

Components

1. **Builder**: Defines an abstract interface for building parts of a product.
2. **ConcreteBuilder**: Implements the Builder interface to construct and assemble the parts of the product.
3. **Product**: Represents the complex object that is being built.
4. **Director**: Constructs an object using the Builder interface.

Example Implementation

Here's a typical implementation of the Builder Pattern in Java:

1. Define the Product

The `Product` class represents the complex object being built.

```
public class Product {
    private String partA;
    private String partB;
    private String partC;

    // Getters and toString() for display
    public String getPartA() { return partA; }
    public String getPartB() { return partB; }
    public String getPartC() { return partC; }

    @Override
    public String toString() {
        return "Product [partA=" + partA + ", partB=" + partB + ", partC=" +
partC + "]";
    }
}
```

2. Define the Builder Interface

The `Builder` interface declares the methods for creating different parts of the `Product`.

```
public interface Builder {
    void buildPartA();
    void buildPartB();
    void buildPartC();
    Product getResult();
}
```

3. Implement the Concrete Builder

The `ConcreteBuilder` class implements the `Builder` interface and constructs the `Product`.

```
public class ConcreteBuilder implements Builder {
    private Product product = new Product();

    @Override
    public void buildPartA() {
        product.setPartA("Part A");
```

```
    }

    @Override
    public void buildPartB() {
        product.setPartB("Part B");
    }

    @Override
    public void buildPartC() {
        product.setPartC("Part C");
    }

    @Override
    public Product getResult() {
        return product;
    }
}
```

4. Define the Director

The `Director` class constructs the product using the `Builder`.

```
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    public void construct() {
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
    }
}
```

5. Use the Builder Pattern

Finally, use the `Director` and `Builder` to create a `Product`.

```
public class Client {
    public static void main(String[] args) {
        Builder builder = new ConcreteBuilder();
        Director director = new Director(builder);
        director.construct();

        Product product = builder.getResult();
        System.out.println(product);
    }
}
```

Advantages

1. **Encapsulation of Construction Logic**: Separates the construction logic from the object representation, making the code more modular.
2. **Control Over Object Construction**: Provides fine control over the construction process, allowing for different configurations.
3. **Immutability**: Can be used to create immutable objects by ensuring that once built, the object cannot be modified.

Disadvantages

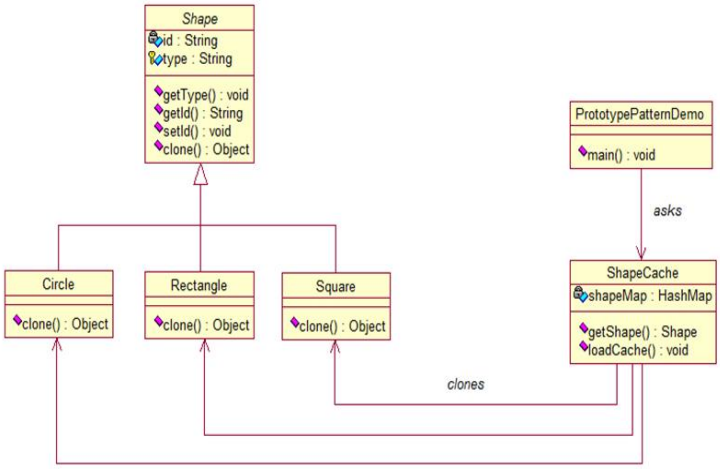
1. **Complexity**: Introduces additional classes and interfaces, which can increase the complexity of the codebase.
2. **Overhead**: May involve additional overhead when only simple object creation is required.

Prototype Design Pattern

The Prototype Design Pattern is a creational design pattern that allows an object to create a copy (clone) of itself. This pattern is useful when the cost of creating a new instance of an object is high, or when an object needs to be duplicated with minor modifications. Instead of creating a new instance from scratch, a prototype object is cloned.

Key Characteristics

1. **Cloning**: The core idea is to use an existing object as a prototype and clone it to create new objects.
2. **Performance**: It can improve performance when creating an object is expensive or complex.
3. **Flexibility**: Allows for easy modification of the cloned object without affecting the original.



Participants

- **Prototype (Interface or Abstract Class)**: Declares a method for cloning itself.
- **ConcretePrototype (Concrete Class)**: Implements the cloning method, returning a copy of itself.
- **Client**: Uses the prototype to create new objects by invoking the cloning method.

Implementation in Java

1. Define the Prototype Interface

The `Prototype` interface declares the `clone()` method.

```
public interface Prototype extends Cloneable {
    Prototype clone();
}
```

2. Implement the ConcretePrototype Class

The `ConcretePrototype` class implements the `Prototype` interface and provides the cloning logic.

```
public class ConcretePrototype implements Prototype {
    private String name;
    private int age;

    public ConcretePrototype(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public Prototype clone() {
        try {
            // Shallow copy using Object's clone method
            return (ConcretePrototype) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "ConcretePrototype [name=" + name + ", age=" + age + "]";
    }

    // Getters and setters
    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

3. Use the Prototype in Client Code

The client code uses the `clone()` method to create a copy of an object.

```
public class Client {
    public static void main(String[] args) {
        // Create an original object
        ConcretePrototype original = new ConcretePrototype("John Doe", 30);

        // Clone the original object
        ConcretePrototype clone = (ConcretePrototype) original.clone();

        // Modify the clone
        clone.setName("Jane Doe");
        clone.setAge(25);

        // Display both objects
        System.out.println("Original: " + original);
        System.out.println("Clone: " + clone);
    }
}
```

Output

```
Original: ConcretePrototype [name=John Doe, age=30]
Clone: ConcretePrototype [name=Jane Doe, age=25]
```

Shallow vs. Deep Copy

- **Shallow Copy**: The `clone()` method performs a shallow copy by default. This means that the cloned object and the original object share references to the same objects for any fields that are objects themselves.
- **Deep Copy**: If you want to create a deep copy (where all objects are fully cloned), you need to implement this manually, ensuring that all nested objects are also cloned.

DeepCopy

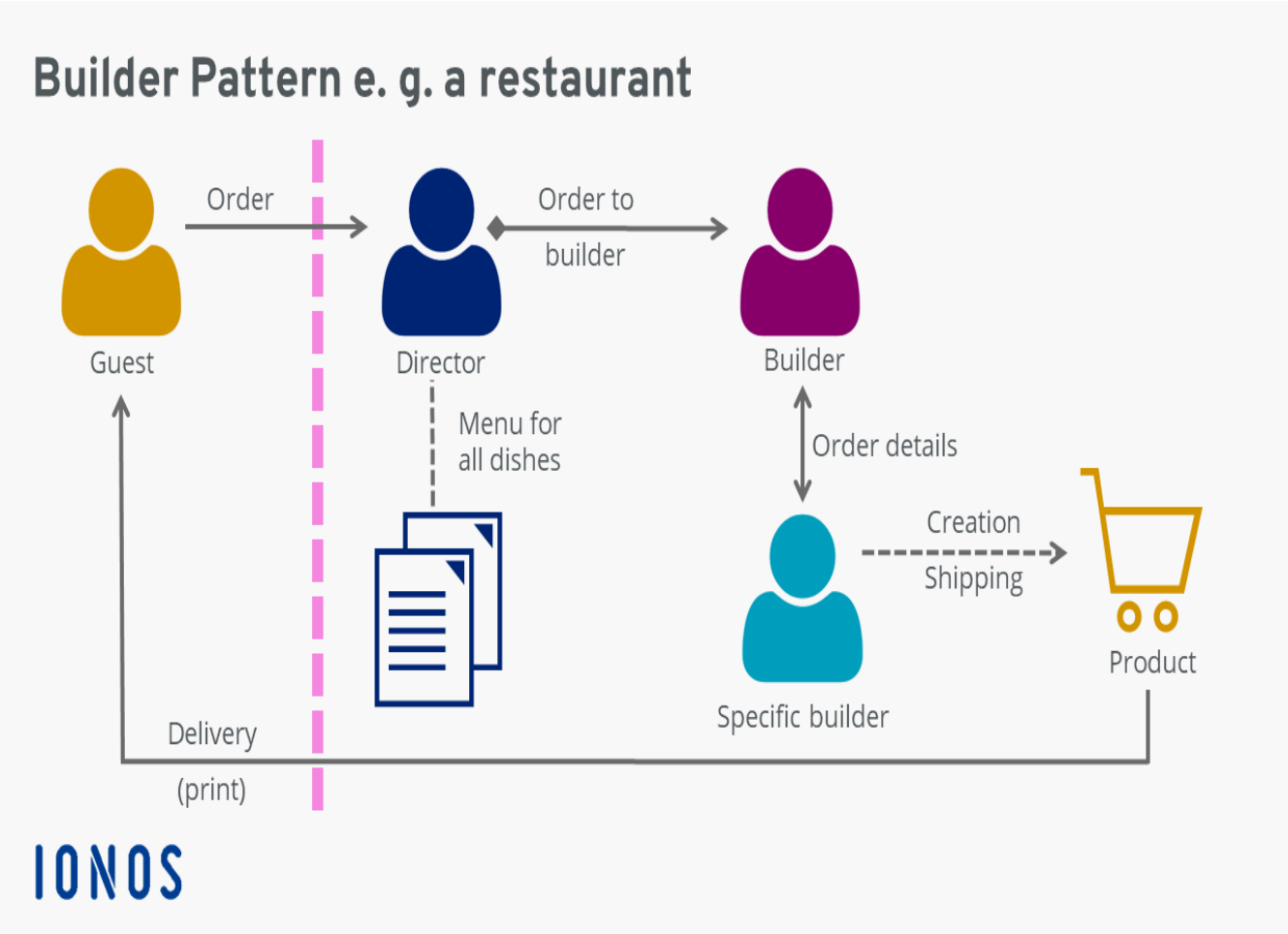
```
public class ConcretePrototype implements Prototype {
    private String name;
    private int age;
    private List<String> hobbies;

    public ConcretePrototype(String name, int age, List<String> hobbies) {
        this.name = name;
        this.age = age;
        this.hobbies = new ArrayList<>(hobbies);
    }

    @Override
    public Prototype clone() {
        try {
            // Shallow copy
            ConcretePrototype cloned = (ConcretePrototype) super.clone();
            // Deep copy of mutable field
            cloned.hobbies = new ArrayList<>(this.hobbies);
            return cloned;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "ConcretePrototype [name=" + name + ", age=" + age + ", hobbies="
+ hobbies + "]";
    }

    // Getters and setters...
}
```

Advantages

1. **Performance**: Reduces the cost of creating objects by reusing existing ones.
2. **Simplicity**: Simplifies the creation of complex objects with many configurations.
3. **Extensibility**: New prototypes can be added without changing existing code.

Disadvantages

1. **Cloning Complexity**: Cloning complex objects with deep copy requirements can be tricky and error-prone.
2. **Deep Copy**: Implementing deep copies can lead to performance overhead if not managed carefully.



Key Structural Design Patterns

1. **Adapter (Wrapper)**:
    - Converts the interface of a class into another interface that a client expects. The Adapter allows classes to work together that couldn't otherwise because of incompatible interfaces.
    - Useful when you want to use an existing class, but its interface does not match the one you need.
2. **Bridge**:
    - Decouples an abstraction from its implementation, allowing them to vary independently.
    - Useful when both the abstractions and their implementations should be extensible by subclassing, and you need to avoid a permanent binding between the two.
3. **Composite**:
    - Composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
    - Useful when you need to treat individual objects and compositions of objects uniformly, like in a file directory system where files and folders are treated similarly.
4. **Decorator**:
    - Adds behavior to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.
    - Useful when you want to add responsibilities to objects dynamically and transparently, without affecting other objects.
5. **Facade**:
    - Provides a simplified interface to a complex subsystem, making the subsystem easier to use.
    - Useful when you want to provide a simple interface to a complex subsystem, like providing a unified interface for a library of functions.
6. **Flyweight**:
    - Reduces the cost of creating and manipulating a large number of similar objects by sharing as much data as possible with similar objects.
    - Useful when you have a large number of objects that are almost identical, like characters in a text editor, where most characters share the same font and size.
7. **Proxy**:
    - Provides a surrogate or placeholder for another object to control access to it.
    - Useful when you need to control access to an object, like in lazy initialization, access control, or remote proxies that represent objects in different address spaces.

Structural Design Patterns: An Overview

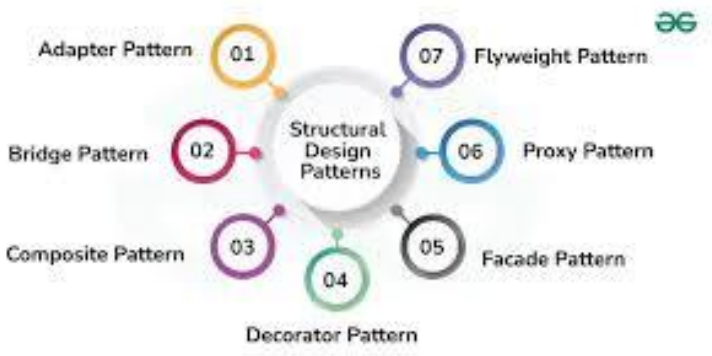
Structural design patterns are a vital aspect of software design, addressing how objects and classes are composed to form larger, more complex structures. These patterns focus on

simplifying relationships between entities, promoting code reuse, and ensuring that the system is adaptable to change.

The Essence of Structural Patterns

At the core of structural design patterns is the principle of composition over inheritance. While inheritance is a powerful feature in object-oriented programming, it can lead to tightly coupled systems where changes in one part of the hierarchy ripple through the entire structure. Structural patterns, by emphasizing composition, encourage the building of systems where objects are composed or assembled to perform complex tasks, rather than relying on deep inheritance hierarchies.

This approach to design supports the creation of flexible and maintainable systems. By focusing on the composition of objects, structural patterns allow for different parts of a system to evolve independently. This means that changes to one part of the system are less likely to require changes to other parts, thereby reducing the potential for bugs and making the system easier to maintain.



Objectives of Structural Design Patterns

1. **Simplification of Complex Structures**: Structural patterns aim to simplify complex relationships between objects and classes. They do this by introducing intermediary structures or modifying existing structures in a way that makes interactions clearer and more manageable. For instance, a pattern might introduce a single point of interaction for a subsystem, thereby reducing the complexity that a client must handle.
2. **Promoting Reusability**: By decoupling objects and promoting composition, structural patterns make it easier to reuse code across different parts of a system. Reusability is enhanced because objects are not tightly bound to specific implementations, allowing them to be reused in different contexts without modification.
3. **Encapsulation of Implementation Details**: Structural patterns often encapsulate the implementation details of a system. By doing so, they protect the client from the

complexities of the underlying implementation, exposing only the necessary interfaces. This not only simplifies the interaction with the system but also allows for the underlying implementation to change without affecting the clients.

4. **Facilitation of System Scalability**: Structural design patterns contribute to system scalability by allowing systems to grow without requiring significant changes to existing code. As new functionality is added, structural patterns provide the framework for incorporating these changes in a way that minimizes disruption to the existing system.

The Role of Composition

Composition, as emphasized by structural patterns, allows objects to be assembled from smaller, more manageable components. This approach stands in contrast to inheritance, where behavior is extended through subclassing. Composition offers several advantages:

- **Flexibility**: Objects can be dynamically composed at runtime, enabling different behaviors depending on the situation.
- **Modularity**: By breaking down functionality into smaller components, composition promotes modularity, making it easier to isolate and fix issues.
- **Reduced Coupling**: Unlike inheritance, where subclasses are tightly coupled to their parents, composition leads to loosely coupled systems where components interact through well-defined interfaces.

Structural Patterns in the Context of System Design

In the broader context of system design, structural design patterns are employed to address specific challenges related to object composition and interaction. These patterns are not isolated solutions but are part of a holistic approach to building systems that are easy to understand, maintain, and extend.

Structural patterns are particularly useful in large, complex systems where the interactions between objects can become difficult to manage. By introducing clear, well-defined structures, these patterns help to manage complexity, ensuring that the system remains robust and adaptable as it evolves over time.

Conclusion

Structural design patterns are a foundational concept in software design, providing strategies for composing objects and classes into larger, more complex structures. By focusing on composition over inheritance, these patterns promote flexibility, reusability, and maintainability. They simplify complex interactions, encapsulate implementation details, and facilitate scalability, making them essential tools in the design of robust, adaptable software systems.

Adapter Design Pattern

The Adapter Design Pattern is a structural pattern that allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface that a client expects. The pattern is often used when you want to use a class that doesn't fit into the existing class structure.

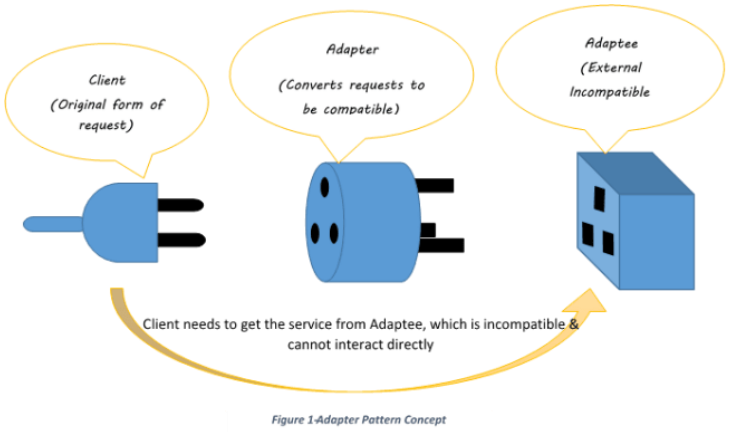


Figure 1-Adapter Pattern Concept

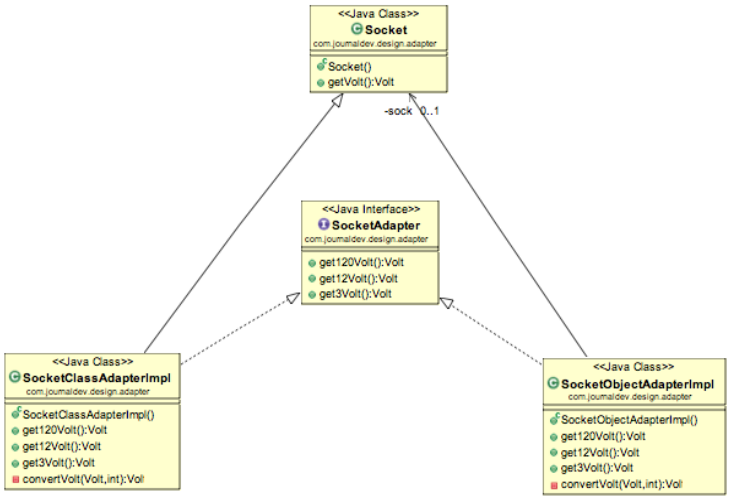
Key Characteristics

1. **Interface Compatibility**: The primary function of the Adapter Pattern is to allow two incompatible interfaces to work together.
2. **Reusability**: It promotes the reuse of existing classes by adapting their interface to fit the needs of the client.
3. **Flexibility**: Allows for the integration of legacy or third-party classes into a new system without modifying their source code.
4. **Decoupling**: Decouples the client code from the implementation details of the incompatible class by using the adapter as an intermediary.

Participants

1. **Target**: This is the interface that the client expects. The adapter must implement this interface.
2. **Adapter**: This is the class that implements the `Target` interface and adapts the `Adaptee` to the `Target`.
3. **Adaptee**: This is the existing class with an incompatible interface that needs to be adapted.
4. **Client**: The client interacts with the `Target` interface, unaware of the presence of the `Adapter`.

UML Diagram



Implementation in Java

1. Define the Target Interface

The `Target` interface represents the interface expected by the client.

```
public interface Target {
    void request();
}
```

2. Define the Adaptee

The `Adaptee` class is the existing class with an incompatible interface.

```
public class Adaptee {
    public void specificRequest() {
        System.out.println("Called specificRequest()");
    }
}
```

3. Implement the Adapter

The `Adapter` class implements the `Target` interface and adapts the `Adaptee` to the `Target`.

```
public class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        // Delegates the call to the adaptee's specificRequest method
        adaptee.specificRequest();
    }
}
```

4. Use the Adapter in Client Code

The client interacts with the Target interface, which is implemented by the Adapter.

```
public class Client {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target adapter = new Adapter(adaptee);

        // Client uses the adapter, unaware of the adaptee
        adapter.request();  // Output: Called specificRequest()
    }
}
```

Practical Use Cases

1. Web Development

In web development, the Adapter pattern can be useful in the following scenarios:

- **Legacy Code Integration**: If you are working on a new web application but need to integrate functionality from an older system with a different API, an adapter can be created to allow the new application to interact with the old system seamlessly.
- **API Wrapping**: When integrating third-party APIs that do not fit the interface required by your application, an adapter can be used to wrap the third-party API and expose a consistent interface to your application.

  Example:

  Suppose you're integrating a payment gateway into your e-commerce site. The payment gateway has a different interface than what your system uses. You can create an adapter that translates your system's payment request format into the format expected by the payment gateway.

2. Android Development

In Android development, the Adapter pattern is commonly used in the following contexts:

- **RecyclerView Adapter**: The Android `RecyclerView` uses an Adapter to bind data to views. The adapter is responsible for adapting the data from a data source (like a list) to a view that can be displayed in the `RecyclerView`.

```java
public class MyAdapter extends RecyclerView.Adapter<MyAdapter.ViewHolder> {
    private List<String> mData;

    public MyAdapter(List<String> data) {
        this.mData = data;
    }

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        View view = inflater.inflate(R.layout.my_text_view, parent, false);
        return new ViewHolder(view);
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        String item = mData.get(position);
        holder.myTextView.setText(item);
    }
```

```
    @Override
    public int getItemCount() {
        return mData.size();
    }

    public static class ViewHolder extends RecyclerView.ViewHolder {
        public TextView myTextView;

        public ViewHolder(View view) {
            super(view);
            myTextView = view.findViewById(R.id.my_text_view);
        }
    }
}
```

ListAdapter and SpinnerAdapter: In Android, `ListAdapter`, `ArrayAdapter`, and `SpinnerAdapter` are also examples of the Adapter pattern. They adapt a collection of data into a format that can be displayed by a UI component like `ListView` or `Spinner`.

Example:

```
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_list_item_1, myStringArray);
myListView.setAdapter(adapter);
```

Advantages

1. **Increased Reusability**: Allows for the reuse of existing classes even if their interfaces are incompatible.
2. **Flexibility**: Adapts an interface to meet new requirements without changing the existing code.
3. **Decoupling**: Reduces the dependency between client code and the underlying class structure.

Disadvantages

1. **Complexity**: Introduces additional layers, which can increase the complexity of the codebase.
2. **Overhead**: The use of adapters may introduce a slight performance overhead.

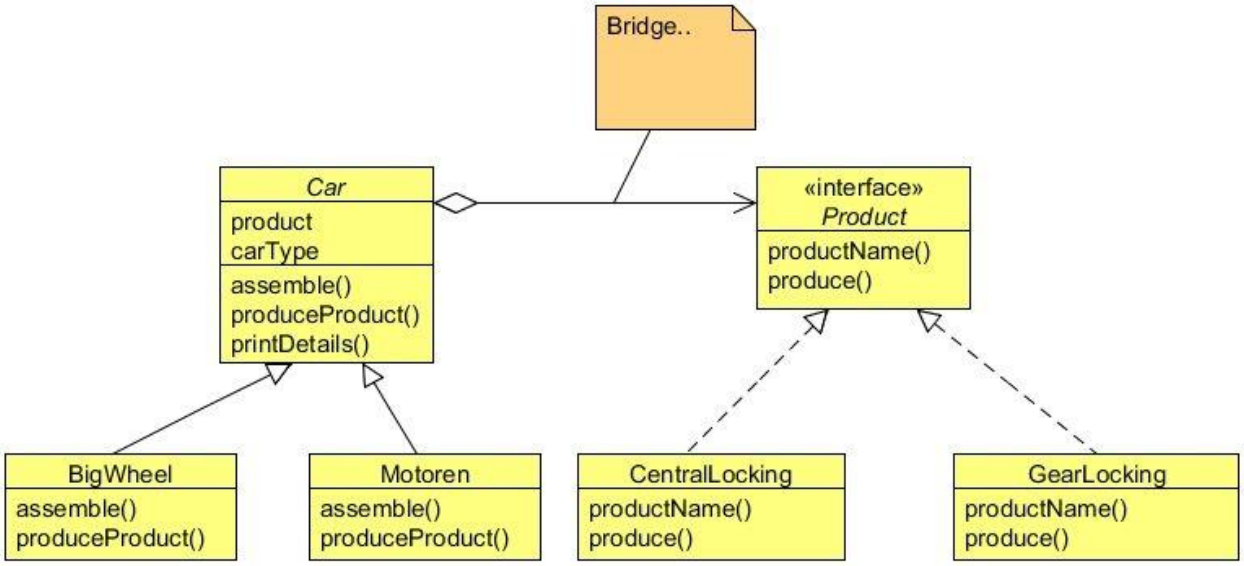
Conclusion

The Adapter Design Pattern is a powerful tool for making incompatible interfaces work together. It is widely used in both web and Android development to ensure that different parts of a system

1. Bridge Design Pattern

The **Bridge Pattern** is a structural design pattern that decouples an abstraction from its implementation so that the two can vary independently. This pattern is useful when you want to avoid a permanent binding between an abstraction and its implementation.

UML Diagram



Implementation in Java

1. Define the Implementor Interface

```
public interface Implementor {
    void operationImpl();
}
```

2. Implement ConcreteImplementors

```
public class ConcreteImplementorA implements Implementor {
    @Override
    public void operationImpl() {
        System.out.println("ConcreteImplementorA operation.");
    }
}

public class ConcreteImplementorB implements Implementor {
    @Override
    public void operationImpl() {
        System.out.println("ConcreteImplementorB operation.");
```

```
    }
}
```

Define the Abstraction

```
public abstract class Abstraction {
    protected Implementor implementor;

    protected Abstraction(Implementor implementor) {
        this.implementor = implementor;
    }

    public abstract void operation();
}
```

Implement RefinedAbstraction

```
public class RefinedAbstraction extends Abstraction {

    public RefinedAbstraction(Implementor implementor) {
        super(implementor);
    }

    @Override
    public void operation() {
        implementor.operationImpl();
    }
}
```

Client Code

```
 public class Client {
    public static void main(String[] args) {
        Implementor implA = new ConcreteImplementorA();
        Implementor implB = new ConcreteImplementorB();

        Abstraction abstraction = new RefinedAbstraction(implA);
        abstraction.operation();  // Output: ConcreteImplementorA operation.

        abstraction = new RefinedAbstraction(implB);
        abstraction.operation();  // Output: ConcreteImplementorB operation.
    }
}
```

1. Composite Design Pattern

Intent: The Composite pattern allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.

Benefits:

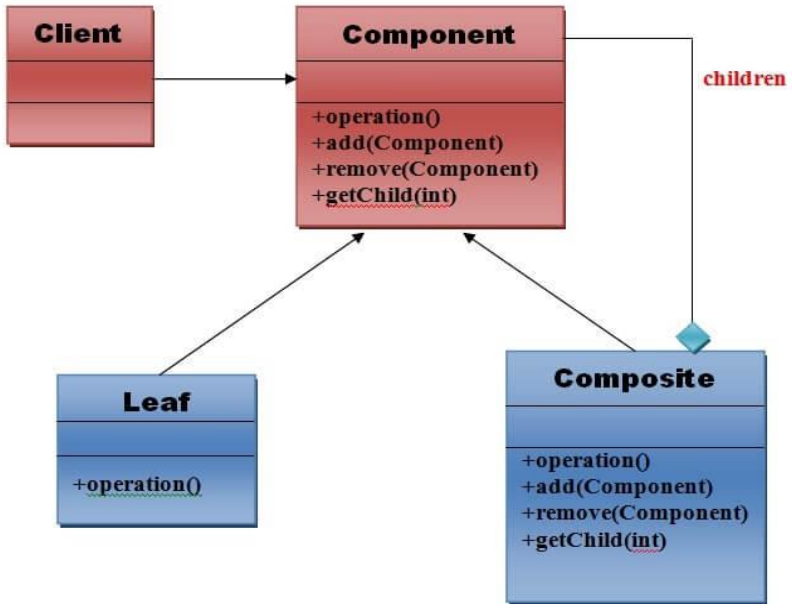
- **Uniformity:** Clients can treat individual objects and compositions of objects in a uniform manner.
- **Flexibility:** Easily add new types of components (leaf or composite) without changing existing code.
- **Ease of Use:** Simplifies client code by allowing it to work with both single objects and groups of objects in the same way.

Properties:

- **Component:** Defines the interface for all objects in the composition, including both leaf and composite objects.
- **Leaf:** Represents leaf objects in the composition. A leaf object does not have any children.
- **Composite:** Defines a behavior for components having children and stores child components.

Example:

In a graphic design application, both individual shapes (like circles and squares) and groups of shapes can be treated uniformly. This allows for hierarchical composition, such as a group of shapes being treated as a single entity.



Code Implementation

```
// Component
interface Graphic {
    void draw();
}

// Leaf
class Circle implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Square implements Graphic {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}

// Composite
class CompositeGraphic implements Graphic {
    private List<Graphic> graphics = new ArrayList<>();

    public void add(Graphic graphic) {
        graphics.add(graphic);
    }

    public void remove(Graphic graphic) {
        graphics.remove(graphic);
    }

    @Override
    public void draw() {
        for (Graphic graphic : graphics) {
            graphic.draw();
        }
    }
}

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

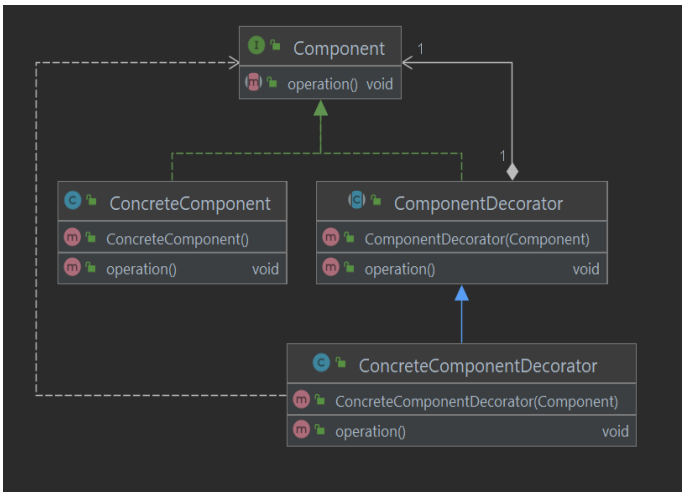
```
        Circle circle1 = new Circle();
        Square square1 = new Square();

        CompositeGraphic group = new CompositeGraphic();
        group.add(circle1);
        group.add(square1);

        group.draw();  // Output: Drawing a Circle, Drawing a Square
    }
}
```

Explanation:

- **Graphic:** The `Component` interface declares the `draw` method that is implemented by both `Leaf` (simple shapes like Circle and Square) and `Composite` (a group of shapes).
- **Circle and Square:** The `Leaf` objects that implement the `Graphic` interface.
- **CompositeGraphic:** The `Composite` class that can contain `Graphic` objects (both `Leaf` and `Composite`).
- **Client:** The client interacts with the objects through the `Graphic` interface, allowing for uniform treatment of individual and composite objects.

2. Decorator Design Pattern

The **Decorator Pattern** is a structural design pattern that allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. This pattern is often used to adhere to the **Single Responsibility Principle** (SRP) by allowing functionality to be divided between classes with unique areas of concern.



Key Concepts

- **Component**: The interface or abstract class that defines the common behavior for both the concrete component and decorators.
- **Concrete Component**: The original object to which additional behavior will be added.
- **Decorator**: An abstract class that implements the `Component` interface and contains a reference to a `Component` object. It forwards the requests to the component it wraps and may add some behavior before or after forwarding.
- **Concrete Decorator**: A class that extends the `Decorator` class and overrides the component's methods to add additional behavior.

How it Works

1. **Component Interface**: Defines the interface that will be implemented by both the `ConcreteComponent` and the decorators.
2. **ConcreteComponent**: The class that will have new behaviors added to it dynamically.
3. **Decorator Class**: Contains a reference to a `Component` object and implements the `Component` interface. It forwards requests to the `ConcreteComponent`, adding additional behavior.
4. **Concrete Decorators**: They extend the `Decorator` class and implement specific behavior to be added to the `ConcreteComponent`.

Example

Imagine a coffee shop where you can order different types of coffee and add extras like milk, sugar, or cream. In this scenario:

- The **Component** could be `Coffee`, an interface defining the `getDescription()` and `cost()` methods.
- The **ConcreteComponent** might be `SimpleCoffee`, implementing the `Coffee` interface and providing the basic coffee.
- **Decorators** could be `Milk`, `Sugar`, and `Cream`, each adding their own cost and description to the `SimpleCoffee`.

Step 1: Define the Component Interface

```
// The Component interface
public interface Coffee {
    String getDescription();
    double getCost();
}
```

Step 2: Implement the Concrete Component

```
// The Concrete Component
public class SimpleCoffee implements Coffee {

    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 5.0;
    }
}
```

Step 2: Implement the Concrete Component

```
// The Concrete Component
public class SimpleCoffee implements Coffee {

    @Override
    public String getDescription() {
        return "Simple Coffee";
    }
```

```
    @Override
    public double getCost() {
        return 5.0;
    }
}
```

Step 3: Create the Decorator Class

```
// The Decorator class
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

Step 4: Implement Concrete Decorators

```
// Concrete Decorator for Milk
public class MilkDecorator extends CoffeeDecorator {

    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
```

```
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 1.5;
    }
}

// Concrete Decorator for Sugar
public class SugarDecorator extends CoffeeDecorator {

    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}
```

Step 5: Use the Decorators

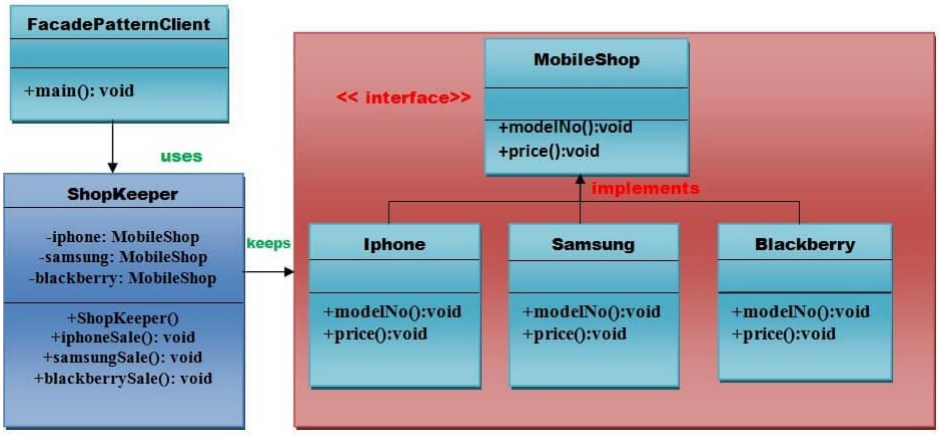
```
public class DecoratorPatternDemo {
    public static void main(String[] args) {
        // Create a simple coffee
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());

        // Add milk to the coffee
        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());

        // Add sugar to the coffee
        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());
    }
}
```

Facade Design Pattern

The **Facade Design Pattern** is a structural design pattern that provides a simplified interface to a complex subsystem. The main idea is to hide the complexities of the system and provide an easy-to-use interface to the client. The Facade Pattern is particularly useful when working with a complex set of classes or systems, as it reduces dependencies on the complex subsystem and promotes loose coupling between the client and the subsystem.

Key Concepts

- **Facade**: A class that provides a simplified interface to a complex subsystem. The Facade delegates the client requests to appropriate objects within the subsystem and may provide additional functionality that simplifies the client's work.
- **Subsystem**: The collection of classes, libraries, or functions that the Facade is meant to simplify. The subsystem itself is not aware of the Facade and does not communicate with the client directly.



Benefits of the Facade Pattern

1. **Simplifies Client Interaction**: The client interacts with the Facade rather than dealing with the complexities of the subsystem.
2. **Reduces Coupling**: The client is less dependent on the complex subsystem, making the system easier to maintain and modify.
3. **Promotes Reusability**: By hiding the subsystem behind a Facade, the subsystem can be reused in different contexts without exposing its complexities.

Example

Let's consider a home theater system, which involves various components like a DVD player, sound system, and projector. Without the Facade Pattern, a client would have to interact with each of these components separately to turn on the system and start a movie.

Step 1: Define the Subsystem Classes

```java
// Subsystem class for DVD Player
public class DVDPlayer {
    public void on() {
        System.out.println("DVD Player is on.");
    }

    public void play(String movie) {
        System.out.println("Playing movie: " + movie);
    }

    public void off() {
        System.out.println("DVD Player is off.");
    }
}

// Subsystem class for Sound System
public class SoundSystem {
    public void on() {
        System.out.println("Sound System is on.");
    }

    public void setVolume(int level) {
        System.out.println("Setting volume to " + level);
    }

    public void off() {
        System.out.println("Sound System is off.");
    }
}

// Subsystem class for Projector
public class Projector {
    public void on() {
        System.out.println("Projector is on.");
    }

    public void setInput(String input) {
        System.out.println("Projector input set to " + input);
    }

    public void off() {
        System.out.println("Projector is off.");
    }
```

```
}
```

Create the Facade Class

```
// Facade class
public class HomeTheaterFacade {
    private DVDPlayer dvdPlayer;
    private SoundSystem soundSystem;
    private Projector projector;

    public HomeTheaterFacade(DVDPlayer dvdPlayer, SoundSystem soundSystem,
Projector projector) {
        this.dvdPlayer = dvdPlayer;
        this.soundSystem = soundSystem;
        this.projector = projector;
    }

    public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        projector.on();
        projector.setInput("DVD");
        soundSystem.on();
        soundSystem.setVolume(10);
        dvdPlayer.on();
        dvdPlayer.play(movie);
    }

    public void endMovie() {
        System.out.println("Shutting down the home theater...");
        dvdPlayer.off();
        soundSystem.off();
        projector.off();
    }
}
```

Use the Façade

```
public class FacadePatternDemo {
    public static void main(String[] args) {
        // Creating subsystem objects
        DVDPlayer dvdPlayer = new DVDPlayer();
        SoundSystem soundSystem = new SoundSystem();
        Projector projector = new Projector();

        // Creating the facade
        HomeTheaterFacade homeTheater = new HomeTheaterFacade(dvdPlayer,
soundSystem, projector);

        // Using the facade to watch a movie
        homeTheater.watchMovie("Inception");

        // Using the facade to end the movie
        homeTheater.endMovie();
    }
}
```

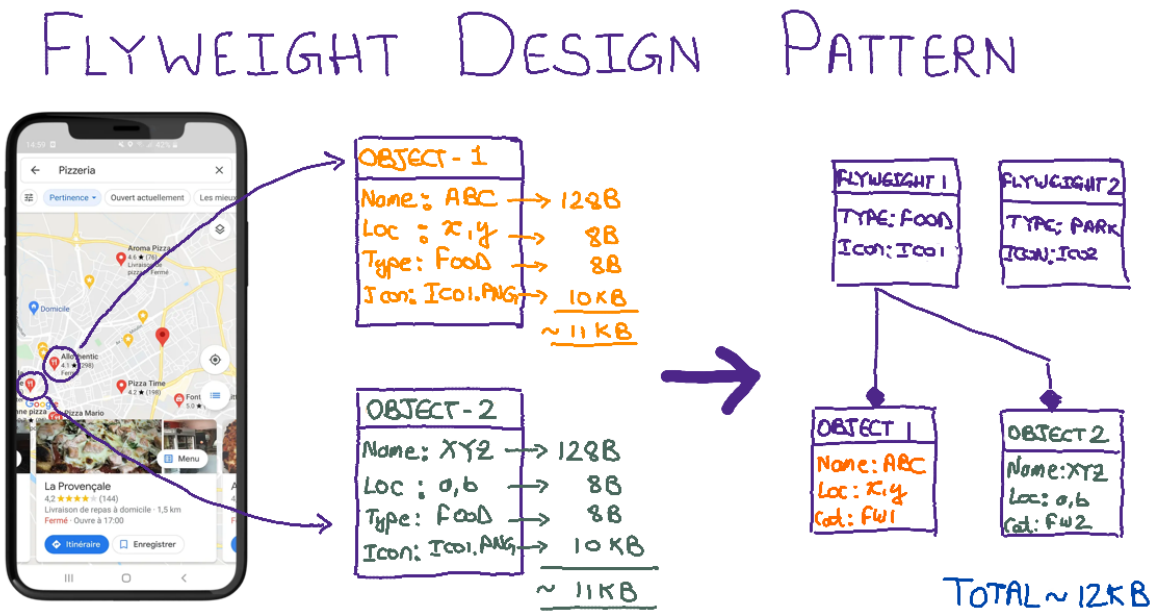
4. Flyweight Design Pattern

Intent: The Flyweight pattern reduces the cost of creating and manipulating a large number of similar objects by sharing common parts.

The **Flyweight Design Pattern** is a structural design pattern that is used to minimize memory usage or computational expenses by sharing as much data as possible with similar objects. It is particularly useful in applications where a large number of objects need to be created, but many of them can share common data.

Key Concepts

- **Flyweight**: The shared object that can be used in multiple contexts simultaneously. It is immutable and should contain intrinsic state (data that can be shared).
- **Intrinsic State**: The state that is shared across different objects. This data is stored in the Flyweight object and is independent of the context.
- **Extrinsic State**: The state that is unique to each object and cannot be shared. This data is passed to the Flyweight object when it is used.
- **Flyweight Factory**: A factory that manages and creates Flyweight objects. It ensures that Flyweights are shared rather than created anew each time.



Benefits of the Flyweight Pattern

1. **Memory Efficiency**: By sharing common data between objects, the pattern greatly reduces the amount of memory needed.
2. **Performance Improvement**: Reducing the number of objects can improve performance, particularly in resource-constrained environments.
3. **Simplification**: The pattern simplifies object creation by ensuring that common data is centralized and managed effectively.

Example

Consider a scenario where we need to create a large number of tree objects in a forest visualization application. Instead of creating a separate object for each tree, we can use the Flyweight Pattern to share common properties like tree type, color, and texture.

Step 1: Define the Flyweight Interface

```
// Flyweight Interface
public interface Tree {
    void display(int x, int y); // Extrinsic state: coordinates where the tree
will be displayed
}
```

Step 2: Implement the Concrete Flyweight Class

```
// Concrete Flyweight Class
public class TreeType implements Tree {
    private String name;    // Intrinsic state
    private String color;   // Intrinsic state
    private String texture; // Intrinsic state

    public TreeType(String name, String color, String texture) {
        this.name = name;
        this.color = color;
        this.texture = texture;
    }

    @Override
    public void display(int x, int y) {
        System.out.println("Displaying " + name + " tree at (" + x + ", " + y +
") with color " + color + " and texture " + texture);
    }
}
```

Step 3: Implement the Flyweight Factory

```
// Flyweight Factory
import java.util.HashMap;
import java.util.Map;

public class TreeFactory {
    private static final Map<String, Tree> treeTypes = new HashMap<>();

    public static Tree getTreeType(String name, String color, String texture) {
        String key = name + "-" + color + "-" + texture;
        if (!treeTypes.containsKey(key)) {
            treeTypes.put(key, new TreeType(name, color, texture));
            System.out.println("Creating new TreeType: " + key);
        }
        return treeTypes.get(key);
```

```
    }
}
```

Step 4: Use the Flyweight in a Client Class

```
public class FlyweightPatternDemo {
    public static void main(String[] args) {
        // Creating a forest with various trees
        Tree oakTree1 = TreeFactory.getTreeType("Oak", "Green", "Rough");
        oakTree1.display(10, 20);

        Tree oakTree2 = TreeFactory.getTreeType("Oak", "Green", "Rough");
        oakTree2.display(30, 40);

        Tree pineTree1 = TreeFactory.getTreeType("Pine", "Dark Green", "Smooth");
        pineTree1.display(50, 60);

        Tree oakTree3 = TreeFactory.getTreeType("Oak", "Green", "Rough");
        oakTree3.display(70, 80);

        Tree pineTree2 = TreeFactory.getTreeType("Pine", "Dark Green", "Smooth");
        pineTree2.display(90, 100);
    }
}
```

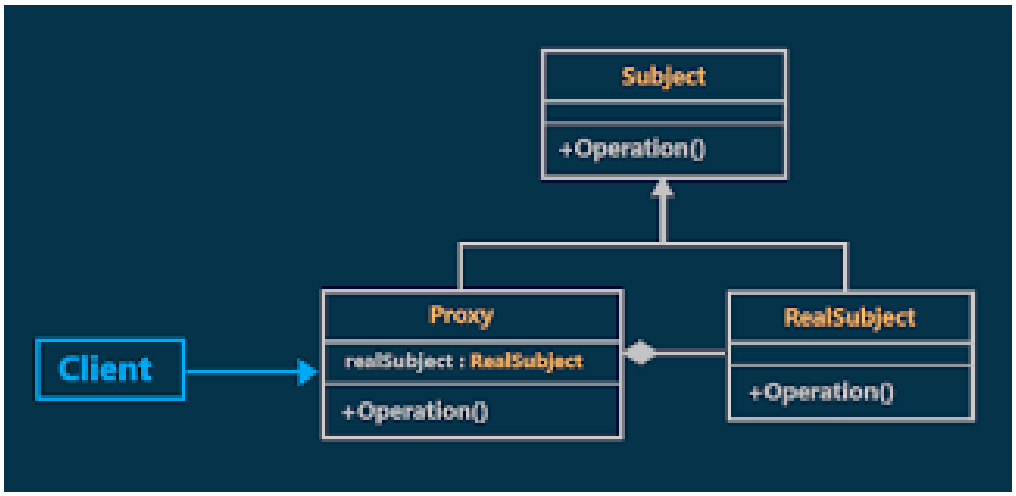
5. Proxy Design Pattern

Intent: The Proxy pattern provides a surrogate or placeholder for another object to control access to it.

The **Proxy Design Pattern** is one of the structural design patterns in software engineering. It provides a surrogate or placeholder for another object to control access to it. The Proxy pattern is useful when you need to add an additional level of control over the actual object, such as controlling access, managing resource-intensive operations, or adding security.

Types of Proxy

1. **Virtual Proxy:** Manages access to a resource that is expensive to create. The proxy creates the real object only when it's needed.
2. **Remote Proxy:** Acts as a local representative for an object that is in a different address space, like a network or another server.
3. **Protection Proxy:** Controls access to the original object, based on access rights.
4. **Smart Proxy:** Provides additional functionality when accessing an object, such as reference counting, lazy initialization, or logging.



Benefits of Proxy Design Pattern

- **Controlled Access:** Proxies can control access to the real object, ensuring that only authorized or appropriate operations are performed.
- **Lazy Initialization:** The proxy can delay the creation of an expensive object until it's needed, optimizing resource usage.
- **Security:** By providing a protection proxy, you can add an extra layer of security, ensuring only certain clients can access the real object.
- **Logging and Auditing:** Proxies can keep track of the usage of the real object, providing logs or audit trails.

- **Remote Access:** Allows access to objects located in different locations or systems through a remote proxy.

Properties of Proxy Design Pattern

- **Subject:** An interface or abstract class that defines the common operations for both the `RealSubject` and the `Proxy`.
- **RealSubject:** The actual object that performs the real operations or actions.
- **Proxy:** The object that controls access to the `RealSubject`. It implements the same interface as the `RealSubject` and maintains a reference to the `RealSubject`.

Example in Java

Let's consider an image viewer application where images are stored on disk, and loading them into memory is resource-intensive. The `ProxyImage` class will control access to the `RealImage` class, loading the image only when necessary.

```
// Subject Interface
interface Image {
    void display();
}

// RealSubject Class
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();  // Load image when object is created
    }

    private void loadImageFromDisk() {
        System.out.println("Loading " + filename);
    }

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

// Proxy Class
class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;
```

```
    public ProxyImage(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        // Load the real image only when needed
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}

// Client Code
public class ProxyPatternDemo {
    public static void main(String[] args) {
        Image image = new ProxyImage("test_image.jpg");

        // Image will be loaded from disk
        image.display();  // Output: Loading test_image.jpg, Displaying
test_image.jpg

        // Image will not be loaded again, only displayed
        image.display();  // Output: Displaying test_image.jpg
    }
}
```

Explanation

- **Subject (Image):** The `Image` interface defines the `display` method, which both `RealImage` and `ProxyImage` implement.
- **RealSubject (RealImage):** This class represents the actual image. It loads the image from the disk in its constructor and displays it when the `display` method is called.
- **Proxy (ProxyImage):** This class acts as a surrogate to `RealImage`. It defers the loading of the image until the `display` method is called, thus implementing lazy loading.

Key Points

- **Lazy Loading:** The `ProxyImage` class delays the loading of the `RealImage` until it's actually required (i.e., when `display` is called for the first time).
- **Memory Efficiency:** This pattern helps in reducing memory consumption and improves performance by not loading the image into memory until needed.

- **Controlled Access:** The proxy controls when and how the `RealImage` is accessed.

Behavioral Design Patterns

Behavioral Design Patterns focus on how objects interact and communicate with each other. Unlike structural patterns, which deal with object composition, and creational patterns, which deal with object creation, behavioral patterns are concerned with the assignment of responsibilities between objects, defining the ways in which they cooperate to achieve a goal. These patterns help make the communication between objects more flexible, efficient, and manageable.

Key Characteristics of Behavioral Design Patterns

1. **Communication Between Objects:** They define how objects communicate and collaborate. These patterns help organize complex flows of control, making the interaction between objects clear and manageable.
2. **Responsibility Distribution:** Behavioral patterns help determine which object should handle which part of the task, thereby promoting loose coupling and a clearer distribution of responsibilities.
3. **Flexibility:** They provide flexibility in carrying out complex tasks by allowing for the dynamic assignment of responsibilities.
4. **Encapsulation of Behavior:** Behavioral patterns often encapsulate algorithms and processes, allowing for code that is more modular and easier to modify or extend.

Common Behavioral Design Patterns

1. **Chain of Responsibility:**
    - **Intent:** Passes a request along a chain of handlers. Each handler decides either to process the request or pass it to the next handler in the chain.
    - **Example:** Logging frameworks where a log request might be passed from one logger to another until it is processed by the appropriate logger.
2. **Command:**
    - **Intent:** Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.
    - **Example:** Undo/redo functionality in applications like text editors.
3. **Interpreter:**
    - **Intent:** Defines a grammatical representation for a language and provides an interpreter to interpret sentences in the language.
    - **Example:** SQL query processing or mathematical expression evaluation.
4. **Iterator:**
    - **Intent:** Provides a way to access elements of a collection sequentially without exposing the underlying representation.
    - **Example:** Iterating over a collection of objects in Java using the `Iterator` interface.

5. **Mediator:**
    - **Intent:** Defines an object that encapsulates how a set of objects interact, promoting loose coupling by preventing direct interactions between objects.
    - **Example:** A chat room that handles communication between users, where the users don't communicate directly but through the mediator (chat room).
6. **Memento:**
    - **Intent:** Captures and externalizes an object's internal state without violating encapsulation, allowing the object to be restored to this state later.
    - **Example:** The save and restore functionality in text editors or games.
7. **Observer:**
    - **Intent:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
    - **Example:** The observer pattern is commonly used in implementing distributed event-handling systems, such as GUI frameworks (e.g., listeners for button clicks).
8. **State:**
    - **Intent:** Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
    - **Example:** A state machine where an object changes its behavior based on its current state, such as a TCP connection which can be in states like `Closed`, `Listening`, or `Established`.
9. **Strategy:**
    - **Intent:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The strategy pattern lets the algorithm vary independently from the clients that use it.
    - **Example:** Different sorting algorithms (e.g., quicksort, mergesort) that can be swapped out at runtime.
10. **Template Method:**
    - **Intent:** Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
    - **Example:** A method in a base class that sets up a game loop, with certain steps like initializing the game or ending the game left to be implemented by subclasses.
11. **Visitor:**
    - **Intent:** Represents an operation to be performed on the elements of an object structure. The Visitor pattern lets you define a new operation without changing the classes of the elements on which it operates.
    - **Example:** A tax calculation system where different types of tax calculations (visitor) can be applied to different types of products or income (elements).

Benefits of Behavioral Design Patterns

- **Loose Coupling:** By defining clear communication protocols between objects, these patterns reduce dependencies, making the system easier to maintain and extend.

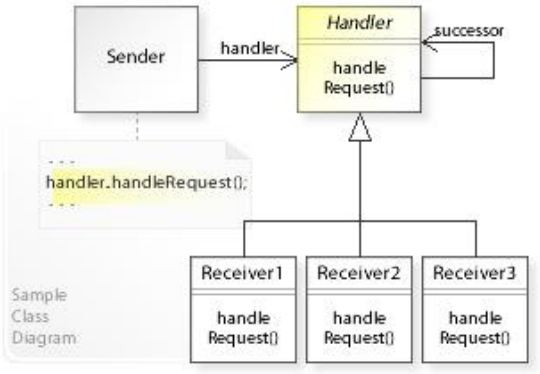
- **Code Reusability:** Behavioral patterns promote reusability by allowing different algorithms or operations to be used interchangeably or added without modifying existing code.
- **Enhanced Flexibility:** They allow for flexible and dynamic interaction between objects, making it easier to change the flow of control or the behavior of objects at runtime.
- **Improved Readability:** These patterns make the interaction between objects more explicit, improving the readability and understandability of the code.

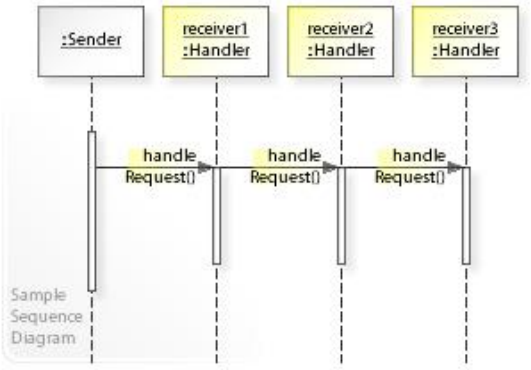
Summary

Behavioral Design Patterns are essential in software design because they offer powerful ways to manage and streamline the communication and interaction between objects. They help in organizing complex control flows, ensuring that objects remain loosely coupled, and responsibilities are well-distributed. These patterns are widely used in real-world applications to achieve more modular, maintainable, and extensible codebases.

1. Chain of Responsibility Pattern

Intent: The Chain of Responsibility pattern allows an object to pass a request along a chain of handlers. Each handler decides whether to process the request or pass it to the next handler in the chain.





Benefits:

- **Reduced Coupling:** Decouples the sender of a request from its receivers.
- **Flexible Request Handling:** You can dynamically change the chain of handlers.

Properties:

- **Handler:** Defines an interface for handling requests and optionally implements default behavior.
- **ConcreteHandler:** Handles requests and can forward them to the next handler in the chain.

- **Client:** Initiates the request to the first handler in the chain.

```
// Handler Interface
abstract class Handler {
    protected Handler nextHandler;

    public void setNextHandler(Handler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public abstract void handleRequest(String request);
}

// Concrete Handler 1
class ConcreteHandlerA extends Handler {
    @Override
    public void handleRequest(String request) {
        if (request.equals("A")) {
            System.out.println("Handler A processing request A");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

// Concrete Handler 2
class ConcreteHandlerB extends Handler {
    @Override
    public void handleRequest(String request) {
        if (request.equals("B")) {
            System.out.println("Handler B processing request B");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

// Client Code
public class ChainOfResponsibilityDemo {
    public static void main(String[] args) {
        Handler handlerA = new ConcreteHandlerA();
        Handler handlerB = new ConcreteHandlerB();
```

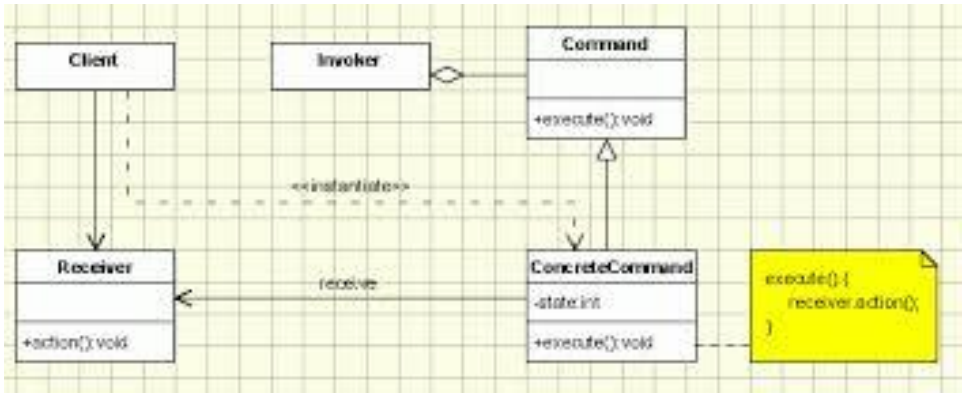
```
        handlerA.setNextHandler(handlerB);

        handlerA.handleRequest("A");  // Output: Handler A processing request A
        handlerA.handleRequest("B");  // Output: Handler B processing request B
        handlerA.handleRequest("C");  // No output, as there is no handler for
"C"
    }
}
```

2. Command

Purpose: The Command pattern encapsulates a request as an object, allowing for parameterization of clients with different requests, queuing, and logging of requests.

Key Concepts:

- **Command**: Interface for executing a command.
- **ConcreteCommand**: Implements the command interface and defines the binding between a receiver and an action.
- **Receiver**: Knows how to perform the operations.
- **Invoker**: Asks the command to execute.
- **Client**: Creates a ConcreteCommand object.



Example Code:

```
// Command interface
interface Command {
    void execute();
}

// Receiver
```

```java
class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }

    public void turnOff() {
        System.out.println("Light is off");
    }
}

// ConcreteCommand
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

// ConcreteCommand
class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}

// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }
```

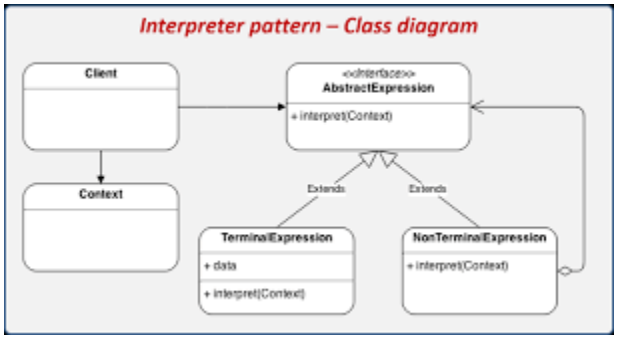
```
    public void pressButton() {
        command.execute();
    }
}

// Client
public class CommandPatternDemo {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);
        Command lightOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();
        remote.setCommand(lightOn);
        remote.pressButton();

        remote.setCommand(lightOff);
        remote.pressButton();
    }
}
```

3. Interpreter

Purpose: The Interpreter pattern defines a grammatical representation for a language and provides an interpreter to interpret sentences in that language.

Key Concepts:

- **Expression**: Interface for interpreting a context.
- **TerminalExpression**: Implements the interpretation for terminal symbols.
- **NonTerminalExpression**: Implements the interpretation for non-terminal symbols.
- **Context**: Contains information that is global to the interpreter.



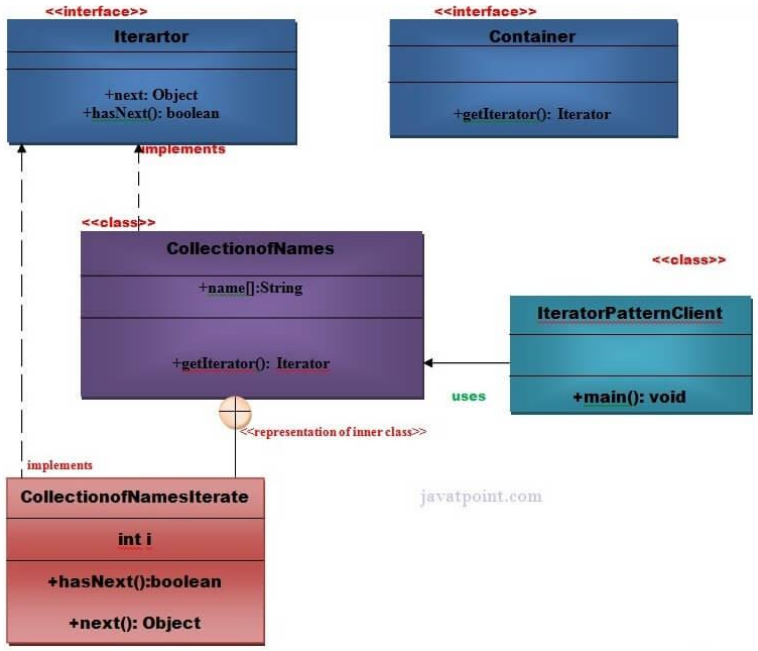
Example Code:

```
// Expression interface
interface Expression {
    boolean interpret(String context);
}

// TerminalExpression
class TerminalExpression implements Expression {
    private String data;

    public TerminalExpression(String data) {
        this.data = data;
    }

    @Override
    public boolean interpret(String context) {
        return context.contains(data);
    }
}

// NonTerminalExpression
class OrExpression implements Expression {
    private Expression expr1;
    private Expression expr2;

    public OrExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }

    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) || expr2.interpret(context);
    }
}

// Client
public class InterpreterPatternDemo {
    public static void main(String[] args) {
        Expression isJava = new TerminalExpression("Java");
```

```
        Expression isPython = new TerminalExpression("Python");

        Expression orExpression = new OrExpression(isJava, isPython);

        System.out.println("Does context contain Java? " +
orExpression.interpret("Java and Python"));
        System.out.println("Does context contain Java? " +
orExpression.interpret("Java and C++"));
    }
}
```

4. Iterator

Purpose: The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Key Concepts:

- **Iterator**: Interface for iterating over elements.
- **ConcreteIterator**: Implements the iterator interface and maintains the current position.
- **Aggregate**: Interface for creating an iterator.
- **ConcreteAggregate**: Implements the aggregate interface and returns an iterator.

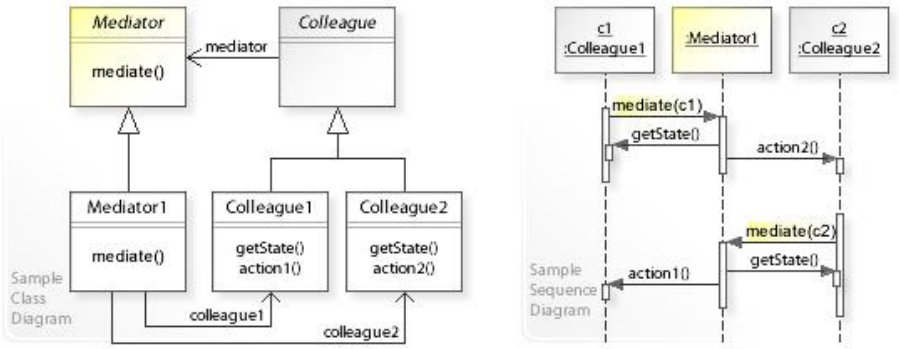
```
import java.util.ArrayList;
import java.util.List;
```

```java
// Iterator interface
interface Iterator<T> {
    boolean hasNext();
    T next();
}

// Aggregate interface
interface Aggregate<T> {
    Iterator<T> createIterator();
}

// ConcreteIterator
class ConcreteIterator<T> implements Iterator<T> {
    private List<T> items;
    private int position = 0;

    public ConcreteIterator(List<T> items) {
        this.items = items;
    }

    @Override
    public boolean hasNext() {
        return position < items.size();
    }

    @Override
    public T next() {
        return items.get(position++);
    }
}

// ConcreteAggregate
class ConcreteAggregate<T> implements Aggregate<T> {
    private List<T> items = new ArrayList<>();

    public void add(T item) {
        items.add(item);
    }

    @Override
    public Iterator<T> createIterator() {
        return new ConcreteIterator<>(items);
    }
}
```

```java
// Client
public class IteratorPatternDemo {
    public static void main(String[] args) {
        ConcreteAggregate<String> aggregate = new ConcreteAggregate<>();
        aggregate.add("Item1");
        aggregate.add("Item2");
        aggregate.add("Item3");

        Iterator<String> iterator = aggregate.createIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

5. Mediator Pattern

Intent: The Mediator pattern defines an object that encapsulates how a set of objects interact, promoting loose coupling by preventing direct interactions between objects.



Benefits:

- **Decoupling:** Reduces the dependency between interacting objects.
- **Centralized Control:** Centralizes communication logic, making it easier to manage.

Properties:

- **Mediator:** Defines an interface for communication between components.
- **ConcreteMediator:** Implements the mediator interface and coordinates communication between components.
- **Colleague:** Defines an interface for components that interact through the mediator.

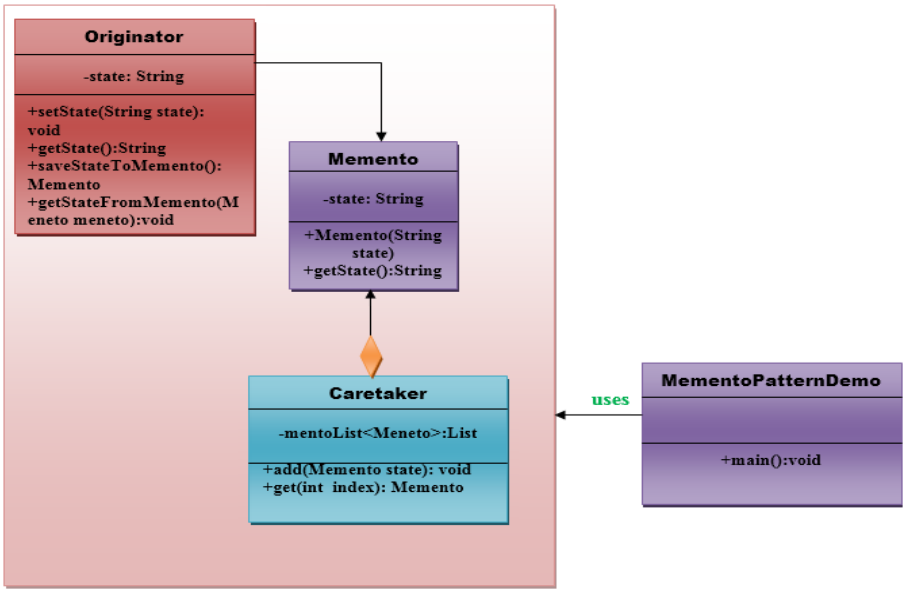
Example Code:

```
// Mediator Interface
interface Mediator {
    void send(String message, Colleague colleague);
}

// Concrete Mediator
class ConcreteMediator implements Mediator {
    private ColleagueA colleagueA;
    private ColleagueB colleagueB;

    public void setColleagueA(ColleagueA colleagueA) {
        this.colleagueA = colleagueA;
    }

    public void setColleagueB(ColleagueB colleagueB) {
        this.colleagueB = colleagueB;
    }

    @Override
    public void send(String message, Colleague colleague) {
        if (colleague == colleagueA) {
            colleagueB.receive(message);
        } else if (colleague == colleagueB) {
            colleagueA.receive(message);
        }
    }
}

// Colleague Interface
abstract class Colleague {
    protected Mediator mediator;

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }

    public abstract void receive(String message);
}

// Concrete Colleague A
class ColleagueA extends Colleague {
    public ColleagueA(Mediator mediator) {
        super(mediator);
```

```
    }

    @Override
    public void receive(String message) {
        System.out.println("Colleague A received: " + message);
    }

    public void send(String message) {
        mediator.send(message, this);
    }
}

// Concrete Colleague B
class ColleagueB extends Colleague {
    public ColleagueB(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void receive(String message) {
        System.out.println("Colleague B received: " + message);
    }

    public void send(String message) {
        mediator.send(message, this);
    }
}

// Client Code
public class MediatorPatternDemo {
    public static void main(String[] args) {
        ConcreteMediator mediator = new ConcreteMediator();

        ColleagueA colleagueA = new ColleagueA(mediator);
        ColleagueB colleagueB = new ColleagueB(mediator);

        mediator.setColleagueA(colleagueA);
        mediator.setColleagueB(colleagueB);

        colleagueA.send("Hello from A");
        colleagueB.send("Hello from B");
    }
}
```

6. Memento Pattern

Intent: The Memento pattern captures and externalizes an object's internal state without violating encapsulation, allowing the object to be restored to this state later.



Benefits:

- **Encapsulation Preservation:** Allows you to save and restore object states without exposing its internal structure.
- **Undo/Redo Operations:** Useful for implementing undo/redo functionality.

Properties:

- **Memento:** Stores the internal state of the originator object.
- **Originator:** Creates a memento containing a snapshot of its current state and can use it to restore its state.
- **Caretaker:** Maintains the memento and can restore the originator to its previous state.

Example Code:

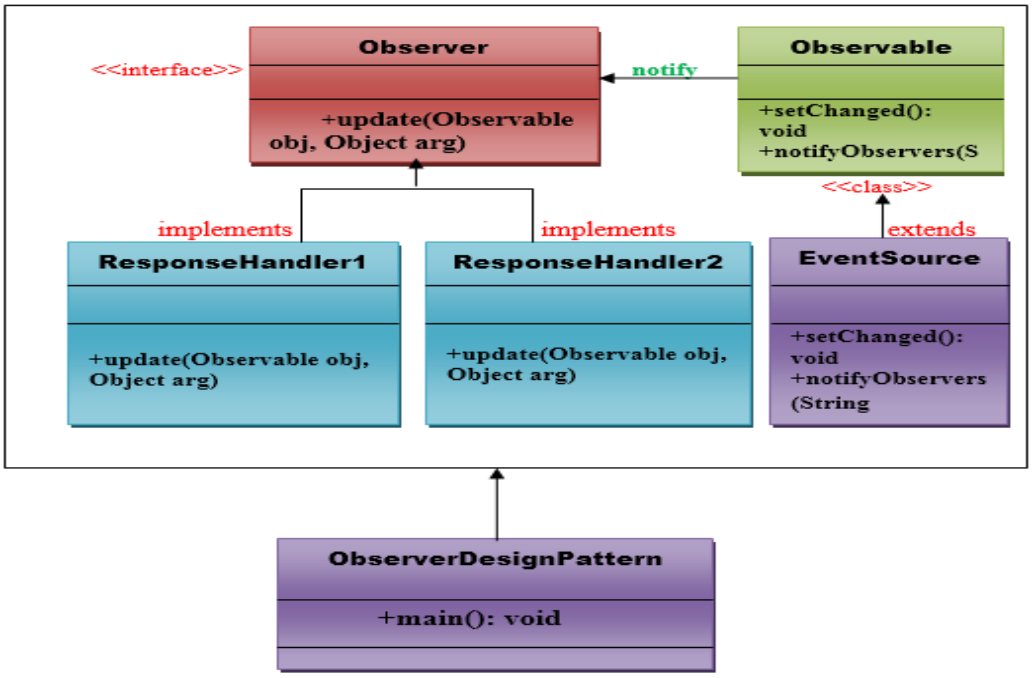
```
// Memento Class
class Memento {
    private String state;

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
```

```
        return state;
    }
}

// Originator Class
class Originator {
    private String state;

    public void setState(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }

    public Memento saveStateToMemento() {
        return new Memento(state);
    }

    public void getStateFromMemento(Memento memento) {
        state = memento.getState();
    }
}

// Caretaker Class
class Caretaker {
    private Memento memento;

    public void saveState(Originator originator) {
        memento = originator.saveStateToMemento();
    }

    public void restoreState(Originator originator) {
        originator.getStateFromMemento(memento);
    }
}

// Client Code
public class MementoPatternDemo {
    public static void main(String[] args) {
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        originator.setState("State1");
```

```
        caretaker.saveState(originator);

        originator.setState("State2");
        System.out.println("Current State: " + originator.getState()); // Output:
State2

        caretaker.restoreState(originator);
        System.out.println("Restored State: " + originator.getState()); //
Output: State1
    }
}
```

7. Observer Pattern

Intent: The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Benefits:

- **Loose Coupling:** Decouples the subject from its observers.
- **Dynamic Relationships:** Observers can be added or removed at runtime.

Properties:

- **Subject:** Maintains a list of observers and facilitates adding/removing observers.
- **Observer:** Defines an interface for receiving updates from the subject.

- **ConcreteObserver:** Implements the observer interface and updates its state based on the subject's state.

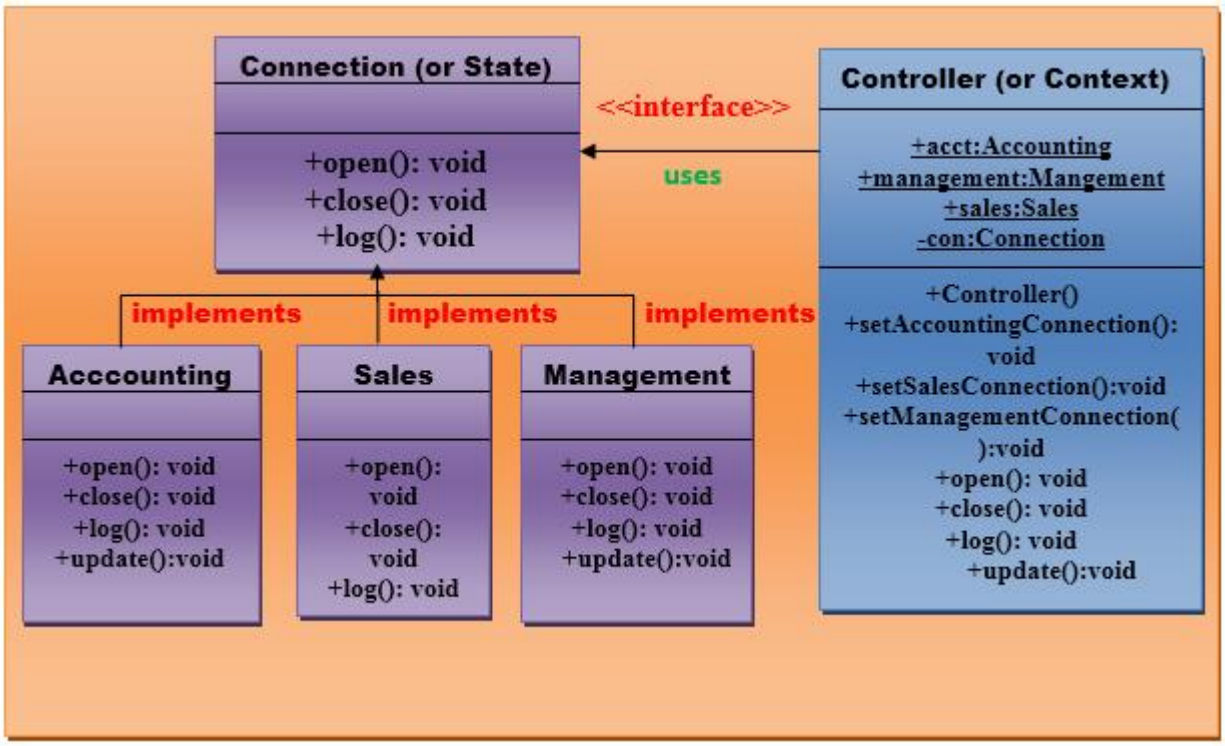
Example Code:

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

// Observer Interface
interface Observer {
    void update(String message);
}

// Subject Class
class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}

// Concrete Observer
class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " received message: " + message);
    }
}
```

```
// Client Code
public class ObserverPatternDemo {
    public static void main(String[] args) {
        Subject subject = new Subject();

        Observer observer1 = new ConcreteObserver("Observer 1");
        Observer observer2 = new ConcreteObserver("Observer 2");

        subject.addObserver(observer1);
        subject.addObserver(observer2);

        subject.notifyObservers("Hello Observers!");
    }
}
```

8. State Pattern

Intent: The State pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



Benefits:

- **State-Specific Behavior:** Simplifies the addition of new states and behaviors.
- **Cleaner Code:** Eliminates complex conditionals and state-related logic.

Properties:

- **Context:** Maintains an instance of a ConcreteState subclass that defines the current state.
- **State:** Defines an interface for encapsulating the behavior associated with a particular state.
- **ConcreteState:** Implements the state-specific behavior.

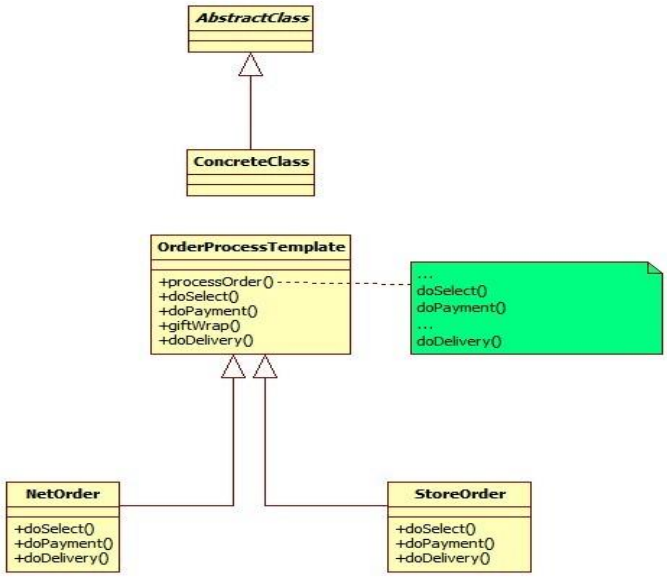
Example Code:

```
// State Interface
interface State {
    void handle();
}

// Concrete State A
class ConcreteStateA implements State {
    @Override
    public void handle() {
        System.out.println("Handling request in State A");
    }
}

// Concrete State B
class ConcreteStateB implements State {
    @Override
    public void handle() {
        System.out.println("Handling request in State B");
    }
}

// Context Class
class Context {
    private State state;

    public void setState(State state) {
        this.state = state;
    }

    public void request() {
        state.handle();
```

```
    }
}

// Client Code
public class StatePatternDemo {
    public static void main(String[] args) {
        Context context = new Context();

        State stateA = new ConcreteStateA();
        State stateB = new ConcreteStateB();

        context.setState(stateA);
        context.request();  // Output: Handling request in State A

        context.setState(stateB);
        context.request();  // Output: Handling request in State B
    }
}
```

10. Template Method Pattern

Intent: The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Benefits:

- **Code Reuse:** Promotes code reuse by defining common behavior in a base class.
- **Extensibility:** Allows subclasses to customize parts of the algorithm.

Properties:

- **AbstractClass:** Defines a template method that contains the skeleton of an algorithm.
- **ConcreteClass:** Implements the steps of the algorithm defined in the abstract class.

Example Code:

```
// Abstract Class
abstract class AbstractClass {
    // Template Method
    public final void templateMethod() {
        step1();
        step2();
        step3();
    }

    protected abstract void step1();
```

```
    protected abstract void step2();
    protected void step3() {
        System.out.println("Step 3 is common for all subclasses");
    }
}

// Concrete Class A
class ConcreteClassA extends AbstractClass {
    @Override
    protected void step1() {
        System.out.println("ConcreteClassA: Step 1");
    }

    @Override
    protected void step2() {
        System.out.println("ConcreteClassA: Step 2");
    }
}

// Concrete Class B
class ConcreteClassB extends AbstractClass {
    @Override
    protected void step1() {
        System.out.println("ConcreteClassB: Step 1");
    }

    @Override
    protected void step2() {
        System.out.println("ConcreteClassB: Step 2");
    }
}

// Client Code
public class TemplateMethodPatternDemo {
    public static void main(String[] args) {
        AbstractClass classA = new ConcreteClassA();
        classA.templateMethod();

        AbstractClass classB = new ConcreteClassB();
        classB.templateMethod();
    }
}
```