Q1 (a) Answer the Following

1. Three Types of Design Patterns in Java:

- Creational Patterns: Deal with object creation mechanisms. Examples include Singleton, Factory Method, and Builder patterns.

- Structural Patterns: Focus on how classes and objects are composed to form larger structures. Examples are Adapter, Composite, and Bridge patterns.

- Behavioral Patterns: Concerned with algorithms and the assignment of responsibilities between objects. Examples include Observer, Strategy, and Command patterns.

2. When to Use Design Patterns:

- Use design patterns when facing recurring design problems in software development.

- They provide proven solutions that enhance code reusability, maintainability, and scalability.

3. Smalltalk MVC:

- The Smalltalk Model-View-Controller (MVC) pattern separates an application into three interconnected components:

  - Model: Manages the data and business logic.

  - View: Displays the data to the user.

  - Controller: Handles user input and updates the model.

4. Factory Method Pattern is also known as:

- Virtual Constructor.

5. Bridge Pattern is also known as:

- Handle/Body Pattern.

6. Early Instantiation:

- Early instantiation refers to creating an instance of a class at the time of its declaration rather than at runtime.

7. Advantage of Prototype Pattern:

- The Prototype Pattern allows for creating new objects by copying an existing object (prototype), which can be more efficient than creating new instances from scratch.

Q.1(b) What is a Design Pattern?

A design pattern is a general reusable solution to a commonly occurring problem in software design. It represents a best practice that can be applied to solve specific design challenges in software development. Design patterns are not finished designs that can be transformed directly into code; rather, they are templates or descriptions of how to solve problems in various contexts.

- How Design Patterns Solve Design Problems

Design patterns address common issues faced during software development by providing established solutions that have been tested and refined over time. They help developers:

1. Standardize Solutions: By using recognized patterns, developers can apply proven solutions to problems, ensuring consistency and reliability in their designs.

2. Facilitate Communication: Design patterns provide a common vocabulary for developers, making it easier to discuss complex design issues and solutions.

3. Promote Reusability: Patterns encourage code reuse, allowing developers to implement solutions without having to reinvent the wheel for every new project.

4. Enhance Maintainability: By structuring code around design patterns, applications become easier to maintain and extend. This modularity allows changes to be made with minimal impact on the overall system.

5. Improve Flexibility: Patterns allow for the decoupling of components, making it easier to modify or replace parts of a system without affecting others.

Advantages of Design Patterns

1. Improved Code Quality: Utilizing design patterns leads to cleaner, more organized code that adheres to best practices, which enhances readability and maintainability.

2. Reduced Complexity: Patterns simplify complex systems by breaking them down into manageable components, making it easier for developers to understand and work with the codebase.

3. Faster Development: By leveraging existing patterns, developers can accelerate the development process, as they do not need to create solutions from scratch.

4. Easier Maintenance: Code structured around design patterns is typically easier to modify and extend, reducing the effort required for future updates or enhancements.

5. Enhanced Collaboration: A shared understanding of design patterns fosters better collaboration among team members, as they can communicate effectively about system architecture and design decisions.

6. Adaptability: Design patterns allow systems to adapt more easily to changing requirements or new technologies by providing flexible structures that can accommodate modifications without significant rework.

Q2 (a) Builder Design Pattern

The Builder Design Pattern is a creational design pattern that allows for constructing complex objects step by step. It separates the construction of a complex object from its representation, enabling the same construction process to create different representations.

- Key Components of the Builder Pattern

1. Product: This is the complex object that is created by the builder. It typically consists of multiple parts that need to be assembled.

2. Builder Interface: This defines the methods for creating different parts of the product. It abstracts the construction process.

3. Concrete Builder: This implements the builder interface and provides specific implementations for building parts of the product.

4. Director: This class manages the construction process. It uses a builder instance to construct the product step by step.

Example Implementation

Let's consider an example of building a Car object:

java

```
// Product
class Car {
    private String engine;
    private String wheels;

    public void setEngine(String engine) { this.engine = engine; }
    public void setWheels(String wheels) { this.wheels = wheels; }

    @Override
    public String toString() {
        return "Car with engine: " + engine + " and wheels: " + wheels;
    }
}

// Builder Interface
```

```
interface CarBuilder {
    void buildEngine();
    void buildWheels();
    Car getCar();
}

// Concrete Builder
class SportsCarBuilder implements CarBuilder {
    private Car car;

    public SportsCarBuilder() {
        car = new Car();
    }

    @Override
    public void buildEngine() {
        car.setEngine("V8 Engine");
    }

    @Override
    public void buildWheels() {
        car.setWheels("Sport Wheels");
    }

    @Override
    public Car getCar() {
        return car;
    }
}

// Director
```

```
class CarDirector {
    private CarBuilder builder;

    public CarDirector(CarBuilder builder) {
        this.builder = builder;
    }

    public Car constructCar() {
        builder.buildEngine();
        builder.buildWheels();
        return builder.getCar();
    }
}

// Client Code
public class BuilderPatternExample {
    public static void main(String[] args) {
        CarBuilder builder = new SportsCarBuilder();
        CarDirector director = new CarDirector(builder);
        Car car = director.constructCar();
        System.out.println(car);
    }
}
```

Explanation of Example

- Product (Car): The Car class represents the complex object being constructed.

- Builder Interface (CarBuilder): Defines methods for building parts of a car.

- Concrete Builder (SportsCarBuilder): Implements the CarBuilder interface and specifies how to build a sports car.

- Director (CarDirector): Manages the building process, ensuring that all parts are built in the correct order.

This pattern is particularly useful when an object needs to be created with many optional parameters or when there are multiple representations of an object.

Q2 (b) Prototype Design Pattern

The Prototype Design Pattern is a creational design pattern that enables cloning existing objects without making the code dependent on their classes. This pattern is especially useful when creating new instances of an object is costly or complex.

Advantages of Prototype Pattern

1. Efficiency: Cloning an existing object can be more efficient than creating a new instance from scratch, particularly for objects that are resource-intensive to create.

2. Decoupling: The prototype pattern reduces dependencies on specific classes, allowing for more flexible code. New classes can be created by copying existing ones rather than requiring knowledge of their implementation details.

3. Dynamic Configuration: It allows for dynamic configuration of objects at runtime, as new prototypes can be created without altering existing code.

Usage Scenarios

- The Prototype Pattern is often used in scenarios where:

  - Object creation is expensive or involves complex setup.

  - You need to create multiple instances of an object with similar properties.

  - You want to avoid subclassing in favor of composition.

Example Implementation

java

```
// Prototype Interface
interface Prototype {
    Prototype clone();
}

// Concrete Prototype
class ConcretePrototype implements Prototype {
    private String property;

    public ConcretePrototype(String property) {
```

```
        this.property = property;
    }

    @Override
    public Prototype clone() {
        return new ConcretePrototype(property);
    }

    @Override
    public String toString() {
        return "ConcretePrototype with property: " + property;
    }
}

// Client Code
public class PrototypePatternExample {
    public static void main(String[] args) {
        ConcretePrototype prototype1 = new ConcretePrototype("Property1");

        // Cloning prototype1 to create prototype2
        ConcretePrototype prototype2 = (ConcretePrototype) prototype1.clone();

        System.out.println(prototype1); // Output: ConcretePrototype with property: Property1
        System.out.println(prototype2); // Output: ConcretePrototype with property: Property1

        // Verify that they are different objects
        System.out.println(prototype1 != prototype2); // Output: true
    }
}
```

In this example:

- The Prototype interface defines a method for cloning.

- ConcretePrototype implements this interface and provides a specific implementation for cloning itself.

- The client demonstrates how to create a clone of an existing object, showcasing both efficiency and decoupling.

The Prototype Pattern is particularly beneficial in applications like graphical editors, where many similar shapes or components need to be instantiated dynamically based on user actions or configurations.

Q3 (a) Singleton Design Pattern

The Singleton Design Pattern is a creational design pattern that restricts the instantiation of a class to one single instance and provides a global point of access to that instance. This pattern is useful when exactly one object is needed to coordinate actions across the system.

- Characteristics of Singleton Pattern

1. Single Instance: Ensures that only one instance of the class exists throughout the application.

2. Global Access Point: Provides a way to access the instance globally, usually through a static method.

3. Lazy Initialization: Often employs lazy initialization, meaning the instance is created only when it is needed.

- How to Create a Singleton Design

To implement the Singleton pattern, follow these steps:

1. Private Constructor: Prevents instantiation from outside the class.

2. Static Variable: Holds the single instance of the class.

3. Static Method: Provides access to the instance, creating it if it does not already exist.

Example Implementation

java

```
class Singleton {
   // Static variable to hold the single instance
   private static Singleton instance;

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

   // Static method to provide access to the instance
   public static Singleton getInstance() {
      if (instance == null) { // Lazy initialization
         instance = new Singleton();
```

```
        }
        return instance;
    }
}

// Client Code
public class SingletonPatternExample {
    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();

        System.out.println(singleton1 == singleton2); // Output: true
    }
}
```

- Structure of Singleton Pattern

The structure can be represented in a UML diagram as follows:

```
+-----------------+
|   Singleton     |
+-----------------+
| - instance      |
+-----------------+
| + getInstance() |
+-----------------+
```

Q3 (b) Adapter Pattern

The Adapter Pattern is a structural design pattern that allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, enabling classes to interact that otherwise could not due to interface differences.

- Advantages of Adapter Pattern

1. Reusability: Enables existing classes to be reused without modification.

2. Flexibility: New functionalities can be introduced without altering existing code.

3. Decoupling: Reduces dependencies between components, allowing for easier maintenance and scalability.

- Usage of Adapter Pattern

The Adapter Pattern is commonly used in scenarios where:

- Integrating new features into legacy systems where existing interfaces cannot be changed.

- Working with third-party libraries that have different interfaces from your application.

Specification for Adapter Pattern

1. Target Interface: Defines the interface that clients expect.

2. Adapter Class: Implements the target interface and translates requests into calls to the adaptee's interface.

3. Adaptee Class: The existing class with an incompatible interface that needs adaptation.

Example Implementation

java

```
// Target Interface
interface Target {
    void request();
}

// Adaptee Class
class Adaptee {
    public void specificRequest() {
```

```
        System.out.println("Specific request from Adaptee.");
    }
}

// Adapter Class
class Adapter implements Target {
    private Adaptee adaptee;

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

    @Override
    public void request() {
        adaptee.specificRequest(); // Adapting the call
    }
}

// Client Code
public class AdapterPatternExample {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target target = new Adapter(adaptee);

        target.request(); // Output: Specific request from Adaptee.
    }
}
```

In this example:

- The Target interface defines what the client expects.

- The Adaptee class has an incompatible interface that needs adaptation.

- The Adapter class implements the Target interface and translates requests into calls to the Adaptee.

This pattern effectively allows systems with incompatible interfaces to work together seamlessly, enhancing code flexibility and reusability.

Q4 (a) Behavioral Design Patterns

Behavioral design patterns are concerned with the interaction and responsibility of objects. They focus on how objects communicate with each other and how they delegate responsibilities. These patterns help define clear communication and control flow between objects, promoting loose coupling and enhancing flexibility.

Key Behavioral Design Patterns

1. Observer Pattern:

- Definition: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Use Case: Commonly used in event handling systems, such as GUI frameworks where multiple components need to respond to user actions.

2. Strategy Pattern:

- Definition: Enables selecting an algorithm's behavior at runtime. It encapsulates algorithms within a family of interchangeable classes.

- Use Case: Useful in scenarios where multiple algorithms can be applied to a problem, such as sorting or filtering data.

3. Command Pattern:

- Definition: Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

- Use Case: Often used in GUI applications for implementing undo/redo functionality.

4. Chain of Responsibility Pattern:

- Definition: Allows multiple handlers to process a request without knowing which handler will ultimately handle it.

- Use Case: Useful in scenarios where a request needs to be handled by multiple objects in a chain, such as logging or event handling.

5. Mediator Pattern:

- Definition: Defines an object that encapsulates how a set of objects interact. It promotes loose coupling by preventing objects from referring to each other explicitly.

- Use Case: Often used in complex UIs where many components need to communicate without tight coupling.

6. Memento Pattern:

- Definition: Captures and externalizes an object's internal state so that the object can be restored to this state later without violating encapsulation.

- Use Case: Useful for implementing undo functionality in applications.

7. State Pattern:

- Definition: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

- Use Case: Commonly used in scenarios where an object's behavior depends on its state, such as in game development or workflow management.

8. Visitor Pattern:

- Definition: Represents an operation to be performed on the elements of an object structure without changing the classes of the elements on which it operates.

- Use Case: Useful when you need to perform operations on a set of objects with different types but want to avoid modifying their classes.

Q4 (b) Chain of Responsibility with Example

The Chain of Responsibility pattern is a behavioral design pattern that allows multiple objects to handle a request without knowing which object will handle it. Each handler decides either to process the request or pass it along the chain.

Key Components

1. Handler Interface: Defines a method for handling requests and a reference to the next handler in the chain.

2. Concrete Handlers: Implement the handler interface and provide specific handling logic.

3. Client Code: Initiates the request and sets up the chain of handlers.

Example Implementation

java

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

    public void setNext(Handler next) {
        this.next = next;
    }

    public abstract void handleRequest(int request);
}

// Concrete Handlers
class ConcreteHandlerA extends Handler {
    public void handleRequest(int request) {
        if (request < 10) {
            System.out.println("Handler A processed request " + request);
        } else if (next != null) {
            next.handleRequest(request);
        }
```

```
    }
}

class ConcreteHandlerB extends Handler {
    public void handleRequest(int request) {
        if (request >= 10 && request < 20) {
            System.out.println("Handler B processed request " + request);
        } else if (next != null) {
            next.handleRequest(request);
        }
    }
}

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

        // Set up the chain
        handlerA.setNext(handlerB);

        // Test requests
        handlerA.handleRequest(5);   // Output: Handler A processed request 5
        handlerA.handleRequest(15);  // Output: Handler B processed request 15
        handlerA.handleRequest(25);  // No handler processes this request
    }
}
```

Explanation of Example

- The Handler class defines the interface for handling requests and maintains a reference to the next handler in the chain.

- ConcreteHandlerA and ConcreteHandlerB implement specific handling logic based on the value of the request.

- The client code sets up the chain by linking handlers together and then initiates requests, demonstrating how requests are processed by different handlers based on their conditions.

This pattern is particularly useful in scenarios where multiple potential handlers exist for a single request, allowing for greater flexibility and separation of concerns within the codebase.

Example:

java

```
abstract class Handler {
    protected Handler next;

    public void setNext(Handler next) {
        this.next = next;
    }

    public abstract void handleRequest(int request);
}

class ConcreteHandlerA extends Handler {
    public void handleRequest(int request) {
        if (request < 10) {
            System.out.println("Handler A processed request " + request);
        } else if (next != null) {
            next.handleRequest(request);
        }
    }
}
```

Q5 (a) State Pattern in Detail

The State Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. The object will appear to change its class, enabling it to exhibit different behaviors based on its current state. This pattern is particularly useful in scenarios where an object's behavior is dependent on its state, such as in user interfaces, game development, and workflow management systems.

Key Components of the State Pattern

1. Context: The class that maintains a reference to one of the concrete states and delegates state-specific behavior to that state.

2. State Interface: An interface that defines the methods that concrete states must implement.

3. Concrete States: Classes that implement the state interface and provide specific behaviors for each state.

How the State Pattern Works

- The context holds a reference to a state object that represents its current state.

- When an operation is called on the context, it delegates the request to the current state object.

- The current state can change the context's state by setting a new state object.

Example Implementation

java

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

// Concrete State A
class ConcreteStateA implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Handling request in State A");
        context.setState(new ConcreteStateB()); // Transition to State B
```

```
    }
}

// Concrete State B
class ConcreteStateB implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Handling request in State B");
        context.setState(new ConcreteStateA()); // Transition back to State A
    }
}

// Context Class
class Context {
    private State state;

    public Context(State state) {
        this.state = state;
    }

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

    public void request() {
        state.handle(this); // Delegate the request to the current state
    }
}

// Client Code
public class StatePatternExample {
```

```
    public static void main(String[] args) {
        Context context = new Context(new ConcreteStateA()); // Initial state A

        context.request(); // Output: Handling request in State A
        context.request(); // Output: Handling request in State B
        context.request(); // Output: Handling request in State A again...
    }
}
```

Explanation of Example

- Context: The Context class maintains a reference to a State object and delegates requests to it.

- Concrete States: ConcreteStateA and ConcreteStateB implement the State interface and define specific behaviors. They can change the context's current state by invoking context.setState().

- Client Code: The client initializes the context with a specific state and calls request(), which delegates the call to the current state's handle() method.

This pattern is beneficial for managing complex states and transitions in applications, allowing for cleaner code and better separation of concerns.

Q5 (b) Template Pattern Structure and Implementation

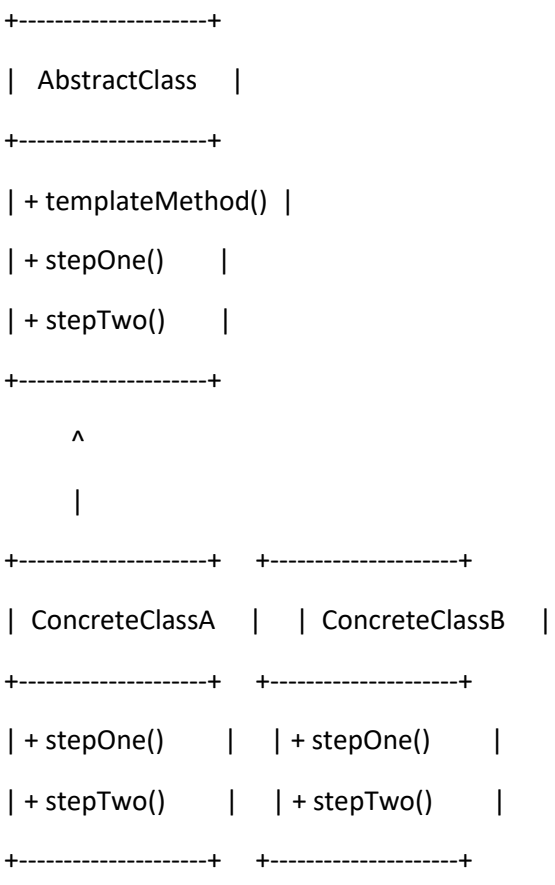
The Template Pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure. This pattern promotes code reuse and consistency by providing a common framework for related operations.

Common Structure of Template Pattern

1. Abstract Class: Contains a template method that defines the algorithm's structure and calls abstract methods for specific steps.

2. Concrete Classes: Implement specific steps defined in the abstract class.

UML Diagram of Template Pattern



Example Implementation

java

```
abstract class AbstractClass {

    // Template Method
    public final void templateMethod() {
        stepOne();
        stepTwo();
        stepThree(); // Final method, cannot be overridden
    }

    protected abstract void stepOne();
    protected abstract void stepTwo();

    private void stepThree() {
        System.out.println("Step Three executed");
    }
}

class ConcreteClassA extends AbstractClass {

    @Override
    protected void stepOne() {
        System.out.println("Step One from Class A executed");
    }

    @Override
    protected void stepTwo() {
        System.out.println("Step Two from Class A executed");
    }
}
```

```
class ConcreteClassB extends AbstractClass {

  @Override
  protected void stepOne() {
    System.out.println("Step One from Class B executed");
  }

  @Override
  protected void stepTwo() {
    System.out.println("Step Two from Class B executed");
  }
}

// Client Code
public class TemplatePatternExample {

  public static void main(String[] args) {

    AbstractClass classA = new ConcreteClassA();
    classA.templateMethod();

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

Explanation of Example

- Abstract Class: The AbstractClass defines the templateMethod() which outlines the algorithm's structure. It calls abstract methods (stepOne() and stepTwo()) that must be implemented by subclasses.

- Concrete Classes: ConcreteClassA and ConcreteClassB provide specific implementations for the abstract methods.

- Client Code: The client creates instances of concrete classes and calls templateMethod(), which executes the defined sequence of steps while allowing subclasses to customize certain parts.

The Template Pattern is particularly useful when you have multiple classes that follow a similar process but require variations at certain steps, promoting code reuse and reducing duplication.

State Pattern

Overview

The State Design Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. This pattern encapsulates state-specific behavior in separate classes, enabling the object to appear as if it has changed its class. It is particularly useful for managing complex state transitions without relying on extensive conditional logic, such as if-else statements or switch-case constructs.

Structure

The State pattern typically involves the following components:

1. Context: This class maintains a reference to the current state and delegates state-specific behavior to the state object.

2. State Interface: This defines the methods that all concrete states must implement.

3. Concrete States: These classes implement the state interface, providing specific behaviors for each state.

UML Representation

The UML diagram for the State pattern includes:

- A `Context` class that holds a reference to a `State`.

- A `State` interface that declares methods for state-specific behavior.

- Concrete classes implementing the `State` interface, each representing a specific state.

Implementation Steps

To implement the State pattern, follow these steps:

1. Define the Context Class: Create a context class that will use the state objects.

```java
public class Context {
    private State currentState;

    public void setState(State state) {
```

```
        this.currentState = state;
    }

    public void request() {
        currentState.handle(this);
    }
}
```

2. Create the State Interface: Define an interface for states.

```java
public interface State {
    void handle(Context context);
}
```

3. Implement Concrete States: Create classes for each specific state.

```java
public class ConcreteStateA implements State {
    public void handle(Context context) {
        // Behavior specific to State A
        context.setState(new ConcreteStateB());
    }
}

public class ConcreteStateB implements State {
    public void handle(Context context) {
        // Behavior specific to State B
        context.setState(new ConcreteStateA());
    }
}
```

```

4. Use the Context and States: Instantiate the context and set an initial state.

```java
Context context = new Context();
context.setState(new ConcreteStateA());
context.request(); // Triggers behavior based on current state
```

Advantages

- Encapsulation of State-Specific Behavior: Each state is encapsulated in its own class, making it easier to manage and extend.

- Reduced Complexity: The need for conditional statements is minimized, leading to cleaner code.

- Flexibility in Adding New States: New states can be added with minimal changes to existing code, promoting maintainability.
```

Template Pattern

Overview

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

Structure

The Template Method pattern consists of:

1. Abstract Class: Contains the template method and defines abstract methods for subclasses to implement.

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

UML Representation

The UML diagram illustrates:

- An abstract class with a template method calling abstract operations.

- Concrete subclasses implementing these operations.

Implementation Steps

To implement the Template Method pattern:

1. Define an Abstract Class: Create an abstract class with a template method.

```java
public abstract class AbstractClass {
    // Template method
    public final void templateMethod() {
        stepOne();
        stepTwo();
        stepThree();
    }
```

```
    protected abstract void stepOne();
    protected abstract void stepTwo();

    protected void stepThree() {
        // Default implementation (optional)
    }
}
```

2. Create Concrete Classes: Implement the abstract methods in concrete classes.

```java
public class ConcreteClassA extends AbstractClass {
    protected void stepOne() {
        // Implementation for Class A
    }

    protected void stepTwo() {
        // Implementation for Class A
    }
}

public class ConcreteClassB extends AbstractClass {
    protected void stepOne() {
        // Implementation for Class B
    }

    protected void stepTwo() {
        // Implementation for Class B
    }
}
```

3. Use the Template Method: Call the template method on instances of concrete classes.

```java
AbstractClass objA = new ConcreteClassA();
objA.templateMethod(); // Executes steps defined in Class A

AbstractClass objB = new ConcreteClassB();
objB.templateMethod(); // Executes steps defined in Class B
```

Advantages

- Code Reusability: Common code can be reused in the template method while allowing customization through subclass implementations.

- Control Over Algorithm Structure: The template method controls the overall structure of an algorithm while allowing variations at specific points.