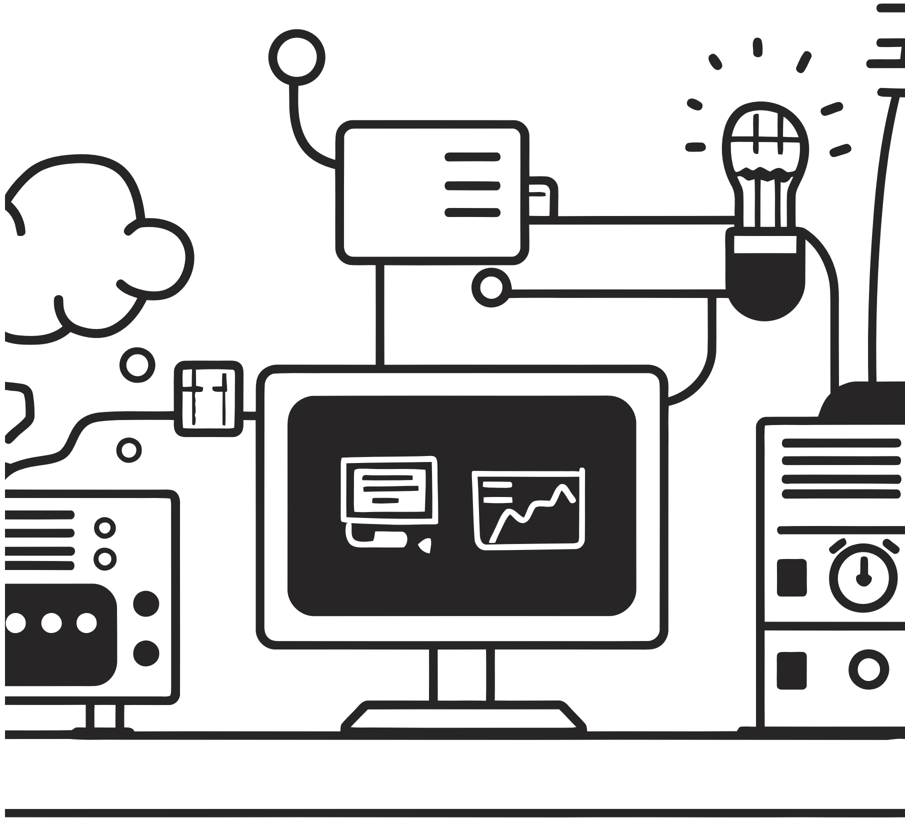
Advanced Programming Techniques - Week 4

Behavioral Design Patterns

Introduction to Behavioral Design Patterns

Behavioral design patterns define **how objects communicate** and interact with each other to achieve a specific behavior in a loosely coupled manner. These patterns help in designing **flexible, reusable, and maintainable** software architectures by encapsulating behaviors and delegating responsibilities efficiently.



Overview of Behavioral Patterns

1. **Strategy Pattern**

   Defines a family of algorithms and lets them be interchangeable.

2. **Observer Pattern**

   Establishes a one-to-many dependency between objects.

3. **Command Pattern**

   Encapsulates a request as an object to parameterize clients.

4. **Chain of Responsibility Pattern**

   Passes a request along a chain of handlers.

5. **State Pattern**

   Allows an object to alter its behavior when its internal state changes.

6. **Template Method Pattern**

   Defines the skeleton of an algorithm with customizable steps.

7. **Visitor Pattern**

   Encapsulates operations to be performed on object structures.

1. Strategy Pattern

Definition:

The **Strategy Pattern** defines a family of algorithms, encapsulates them, and makes them interchangeable.

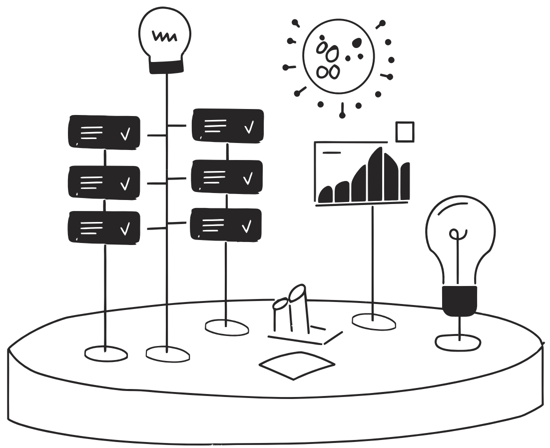
The strategy is chosen at runtime, **decoupling** the algorithm from the client that uses it.

Use Cases:

- When multiple algorithms can be applied to a problem.
- When algorithms need to be selected dynamically at runtime.
- When avoiding multiple conditional statements in code is desired.

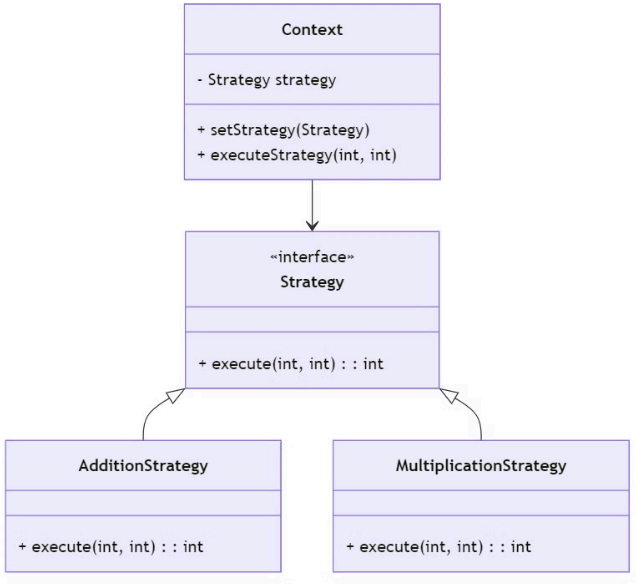
Common Incorrect Usage:

- Hardcoding the algorithm inside the client class.
- Violating the **Open/Closed Principle** by adding conditions to switch strategies.



Strategy Pattern Implementation

Diagram:



Java Implementation:

```java
interface Strategy { int execute(int a, int b); }
class AdditionStrategy implements Strategy {
  public int execute(int a, int b) {
    return a + b;
  }
}
class MultiplicationStrategy implements Strategy {
  public int execute(int a, int b) {
    return a * b;
  }
}
class Context {
  private Strategy strategy;
  public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }
  public int executeStrategy(int a, int b) {
    return strategy.execute(a, b);
  }
}
public class StrategyPatternDemo {
  public static void main(String[] args) {
    Context context = new Context();
    context.setStrategy(new AdditionStrategy());
    System.out.println("Addition: " + context.executeStrategy(5, 3));
    context.setStrategy(new MultiplicationStrategy());
    System.out.println("Multiplication: " + context.executeStrategy(5, 3));
  }
}
```

2. Observer Pattern

Definition:

The **Observer Pattern** defines a **one-to-many dependency** between objects.

When one object (subject) changes state, **all dependent observers are notified automatically**.

Use Cases:

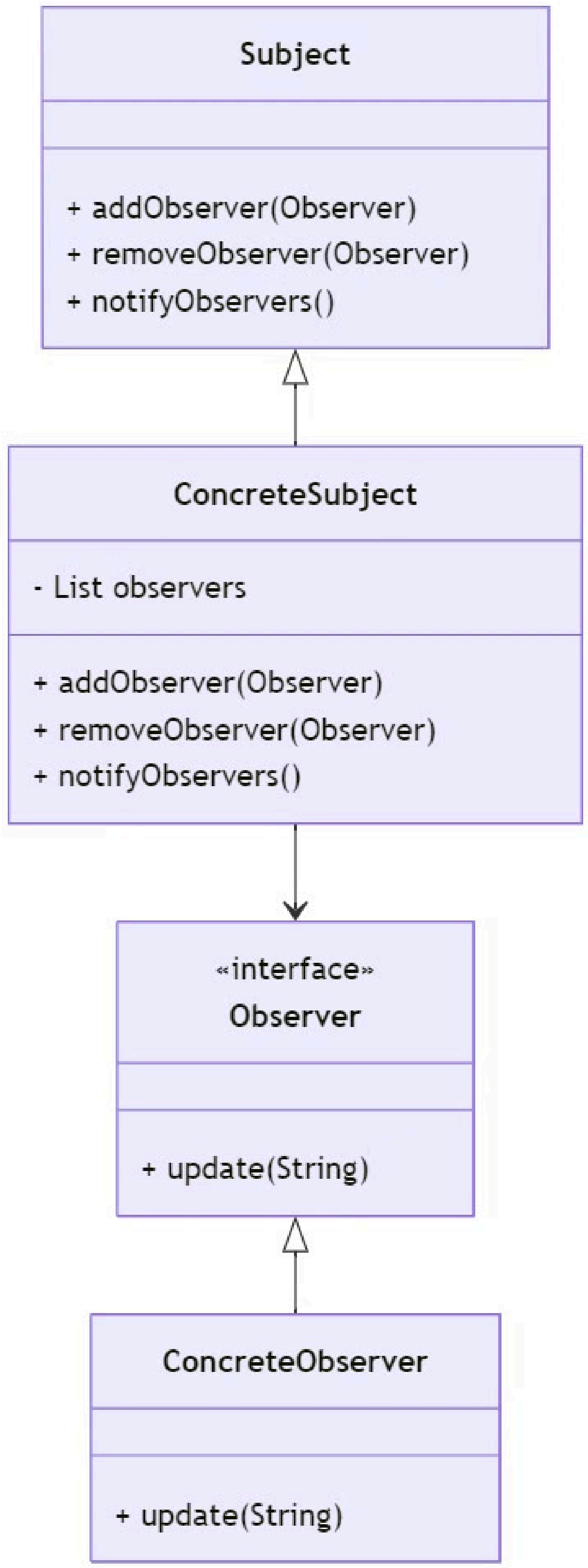
- Implementing event-driven systems (e.g., GUI event listeners).
- Real-time data synchronization (e.g., stock market updates).
- Reducing tight coupling between components.

Common Incorrect Usage:

- Tight coupling between the observer and subject classes.
- Forgetting to remove observers, leading to memory leaks.

Observer Pattern Implementation

Mermaid Diagram:



Java Implementation:

```
interface Observer { void update(String message); }

class ConcreteObserver implements Observer {
  private String name;
  public ConcreteObserver(String name) {
    this.name = name;
  }
  public void update(String message) {
    System.out.println(name + " received update: " + message);
  }
}

interface Subject {
  void addObserver(Observer observer);
  void removeObserver(Observer observer);
  void notifyObservers(String message);
}

class ConcreteSubject implements Subject {
  private List 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);
    }
  }
}

public class ObserverPatternDemo {
  public static void main(String[] args) {
    ConcreteSubject subject = new ConcreteSubject();
    Observer observer1 = new ConcreteObserver("Observer 1");
    Observer observer2 = new ConcreteObserver("Observer 2");
    subject.addObserver(observer1);
    subject.addObserver(observer2);
    subject.notifyObservers("New Update Available!");
  }
}
```

3. Command Pattern

The **Command Pattern** encapsulates a request as an object, thereby allowing users to parameterize clients with different requests, queue requests, and log the history of executed operations.

Definition

Encapsulates a request as an object

Use Cases

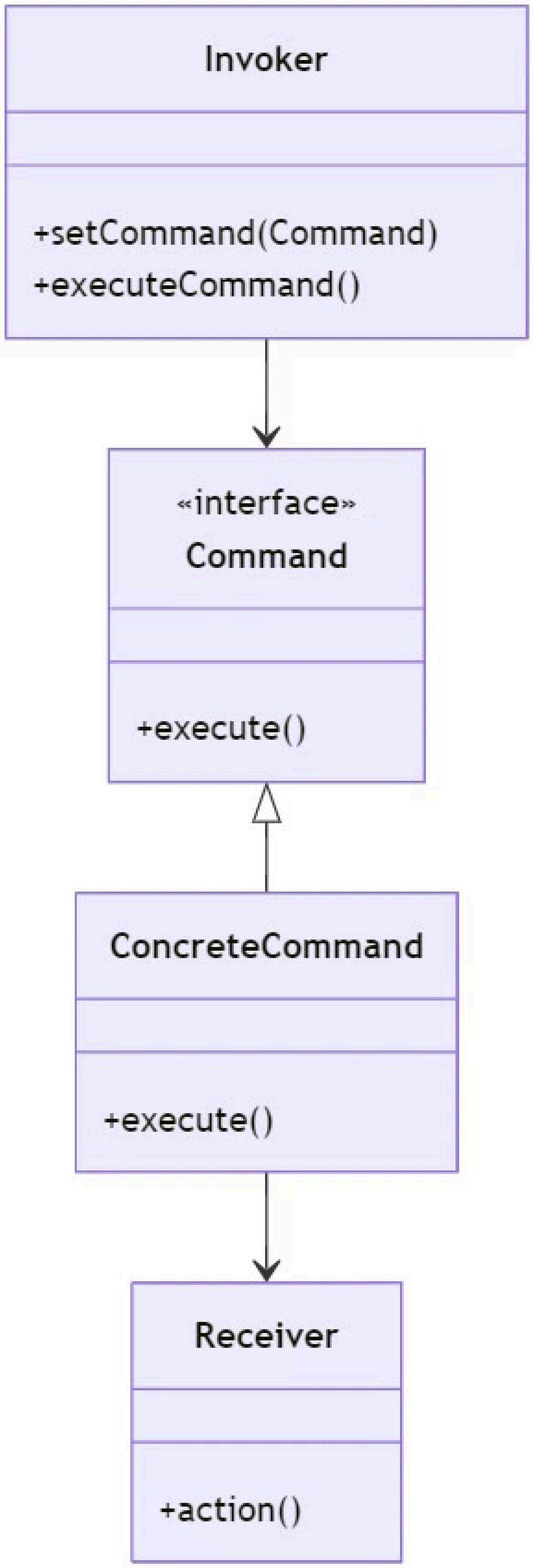
- Implementing undo/redo functionality in applications
- Creating task schedulers and job queues
- Decoupling senders and receivers of requests

Common Incorrect Usage

- Tightly coupling the invoker and receiver, reducing flexibility
- Not implementing a proper command history when needed

Command Pattern Implementation

Diagram:



Java Implementation:

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

Receiver {
  void action() {
    System.out.println("Receiver performing action.");
  }
}

class ConcreteCommand implements Command {
  private Receiver receiver;
  public ConcreteCommand(Receiver receiver) {
    this.receiver = receiver;
  }
  public void execute() {
    receiver.action();
  }
}

Invoker {
  private Command command;
  public void setCommand(Command command) {
    this.command = command;
  }
  public void executeCommand() {
    command.execute();
  }
}

public class CommandPatternDemo {
  public static void main(String[] args) {
    Receiver receiver = new Receiver();
    Command command = new ConcreteCommand(receiver);
    Invoker invoker = new Invoker();
    invoker.setCommand(command);
    invoker.executeCommand();
  }
}
```

4. Chain of Responsibility Pattern

The **Chain of Responsibility Pattern** allows a request to be passed along a chain of handlers until one of them processes it. Each handler in the chain decides **whether to handle the request or pass it along**.

Use Cases:

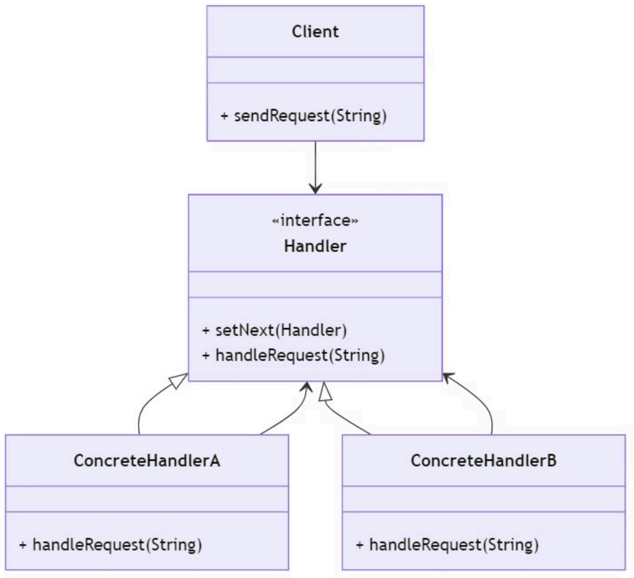
- Implementing **event propagation** in GUI applications.
- Handling **authorization and validation** checks in software systems.
- Creating **logging frameworks** with different log levels.

Common Incorrect Usage:

- Creating **tight coupling** between handlers and clients.
- Not terminating the chain when needed, leading to redundant processing.

Chain of Responsibility Implementation

Diagram:



Java Implementation:

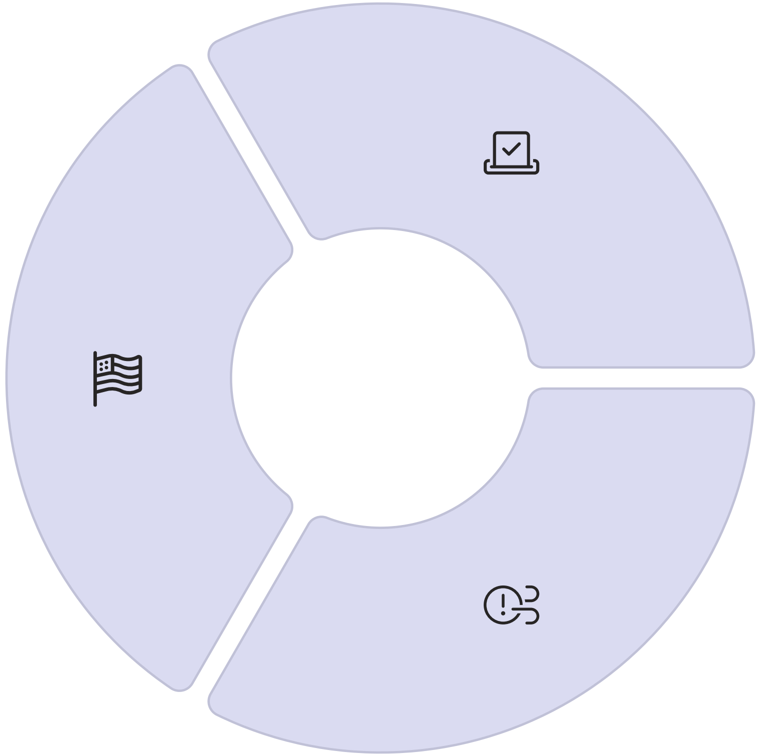
```
interface Handler {
  void setNext(Handler handler);
  void handleRequest(String request);
}

class ConcreteHandlerA implements Handler {
  private Handler next;
  public void setNext(Handler handler) {
    this.next = handler;
  }
  public void handleRequest(String request) {
    if (request.equals("A")) {
      System.out.println("Handler A processed the request.");
    } else if (next != null) {
      next.handleRequest(request);
    }
  }
}

class ConcreteHandlerB implements Handler {
  private Handler next;
  public void setNext(Handler handler) {
    this.next = handler;
  }
  public void handleRequest(String request) {
    if (request.equals("B")) {
      System.out.println("Handler B processed the request.");
    } else if (next != null) {
      next.handleRequest(request);
    }
  }
}

public class ChainOfResponsibilityDemo {
  public static void main(String[] args) {
    Handler handlerA = new ConcreteHandlerA();
    Handler handlerB = new ConcreteHandlerB();
    handlerA.setNext(handlerB);
    handlerA.handleRequest("A");
    handlerA.handleRequest("B");
    handlerA.handleRequest("C");
  }
}
```

5. State Pattern

Definition

The **State Pattern** allows an object to alter its behavior when its **internal state changes**.

The object appears to change its class by switching between different states.



Use Cases

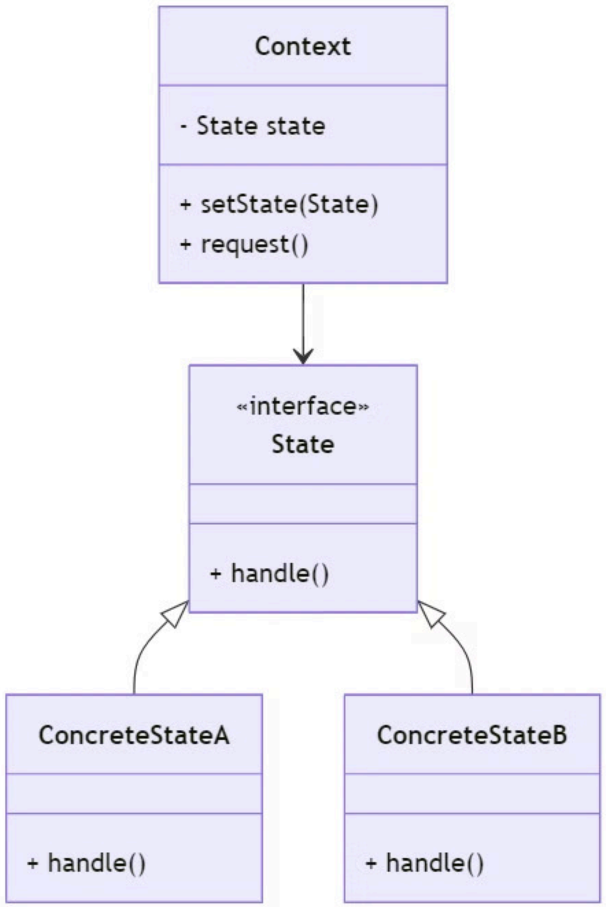
- Implementing **finite state machines** (e.g., vending machines, ATMs).
- Managing **workflow transitions** in applications.
- Avoiding complex if-else or switch statements for state management.

Common Incorrect Usage

- Using **multiple conditionals** instead of encapsulating state logic.
- Not maintaining a **single source of truth** for the object's state.

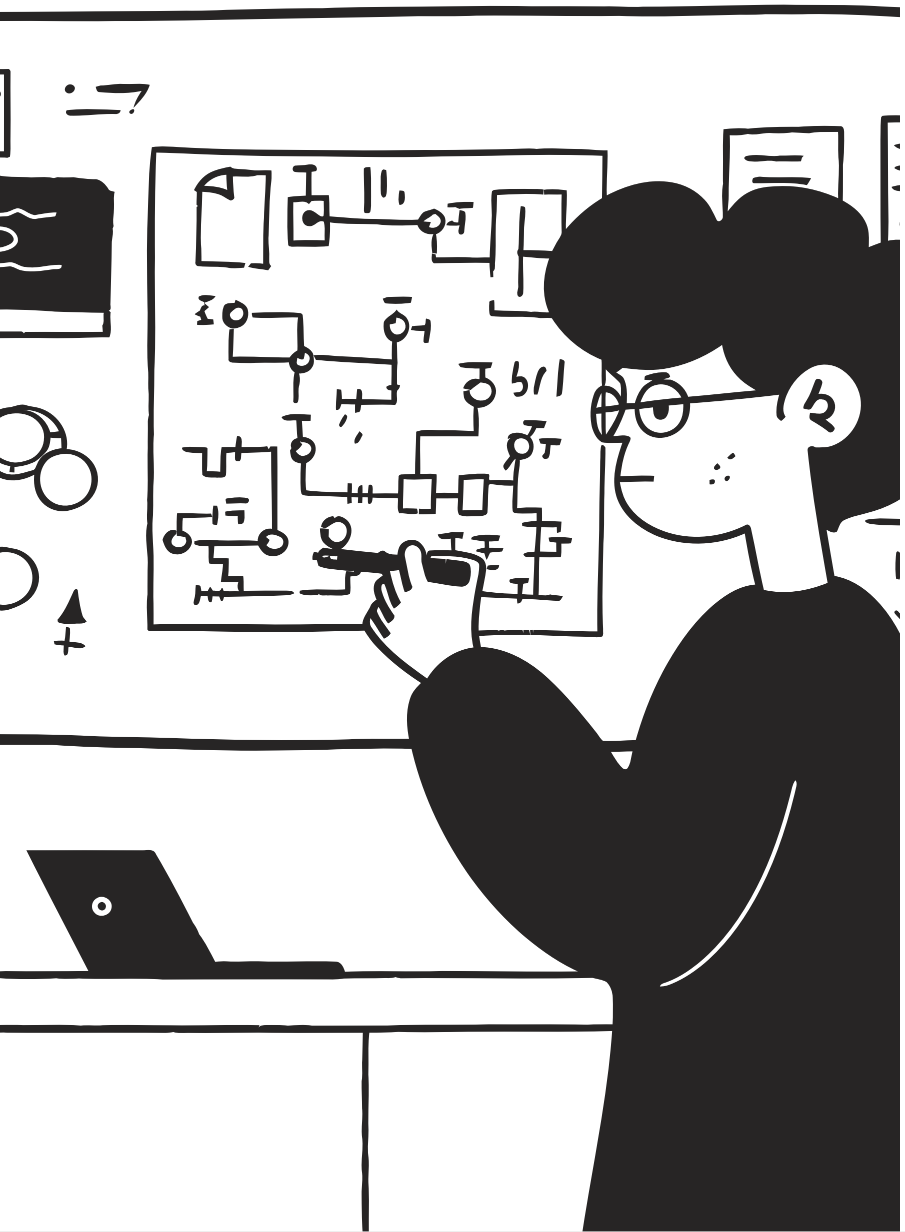
State Pattern Implementation

Diagram:



Java Implementation:

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

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

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

class Context {
  private State state;
  public void setState(State state) {
    this.state = state;
  }
  public void request() {
    state.handle();
  }
}

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();
    context.setState(stateB);
    context.request();
  }
}
```



6. Template Method Pattern

Definition

The Template Method Pattern defines the skeleton of an algorithm in a superclass but allows subclasses to override specific steps without changing its structure.

Use Cases

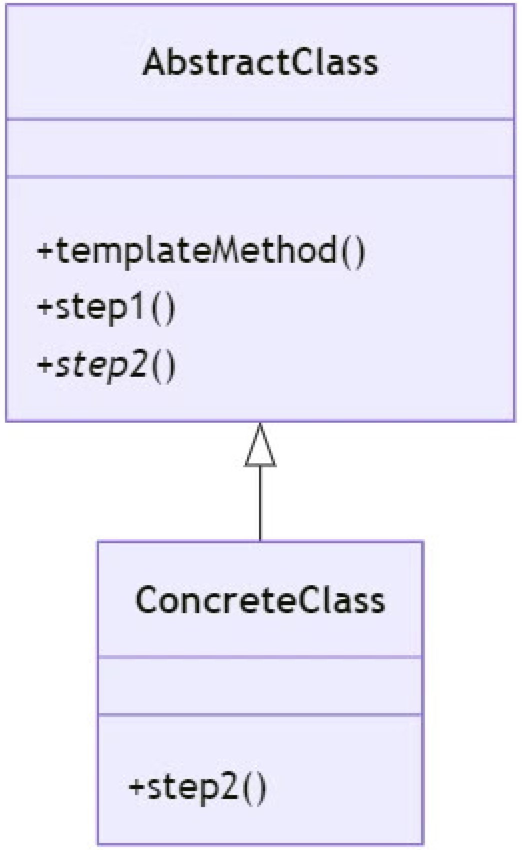
- Implementing workflow processes where some steps are predefined.
- Defining a generic algorithm while allowing extensions in subclasses.
- Enforcing coding standards across teams.

Common Incorrect Usage

- Placing all logic in the base class, limiting flexibility.
- Making the template method too rigid without extension points.

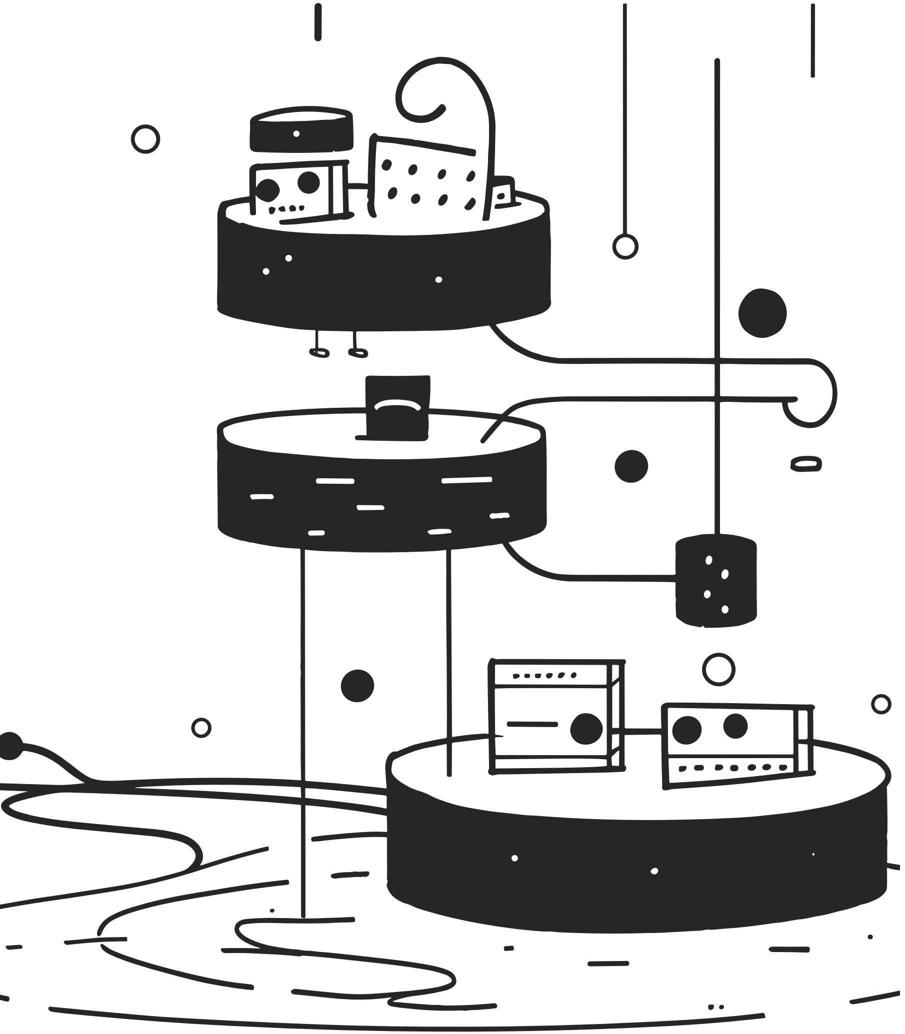
Template Method Pattern Implementation

Diagram:



Java Implementation:

```
abstract class AbstractClass {

  public final void templateMethod() {
    step1();
    step2();
  }

  void step1() {
    System.out.println("Executing step 1 (fixed behavior)");
  }

  abstract void step2();
}

class ConcreteClass extends AbstractClass {
  void step2() {
    System.out.println("Executing step 2 (custom behavior)");
  }
}

public class TemplateMethodDemo {
  public static void main(String[] args) {
    AbstractClass instance = new ConcreteClass();
    instance.templateMethod();
  }
}
```



7. Visitor Pattern

Definition

The Visitor Pattern allows adding new operations to existing object structures without modifying their classes. It separates algorithm logic from the object structure.

Use Cases

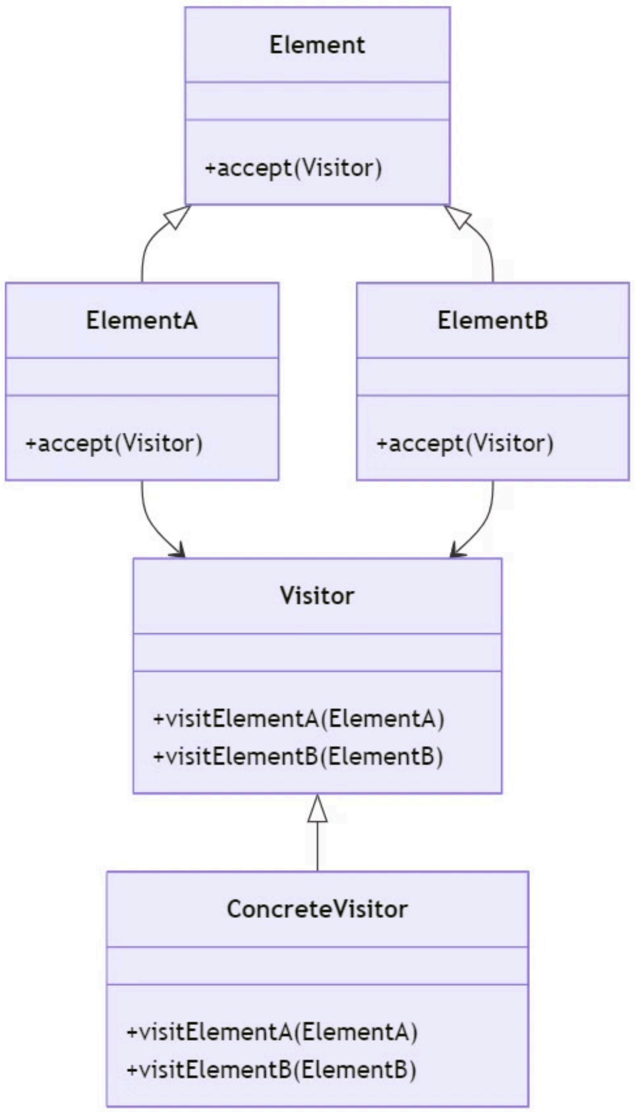
- Processing elements in hierarchical structures (e.g., parsing XML, AST).
- Adding new operations to a system without modifying existing classes.
- Implementing double dispatch in programming languages.

Common Incorrect Usage

- Using when class hierarchy rarely changes, making it unnecessary.
- Overcomplicating simple scenarios where methods in base classes would suffice.

Visitor Pattern Implementation

Diagram:



Java Implementation:

```
interface Visitor {
  void visitElementA(ElementA element);
  void visitElementB(ElementB element);
}

interface Element {
  void accept(Visitor visitor);
}

class ElementA implements Element {
  public void accept(Visitor visitor) {
    visitor.visitElementA(this);
  }
}

class ElementB implements Element {
  public void accept(Visitor visitor) {
    visitor.visitElementB(this);
  }
}

class ConcreteVisitor implements Visitor {
  public void visitElementA(ElementA element) {
    System.out.println("Processing Element A");
  }

  public void visitElementB(ElementB element) {
    System.out.println("Processing Element B");
  }
}

public class VisitorPatternDemo {
  public static void main(String[] args) {
    Element[] elements = {new ElementA(), new ElementB()};
    Visitor visitor = new ConcreteVisitor();

    for (Element e : elements) {
      e.accept(visitor);
    }
  }
}
```