设计模式报告

1. 文档引言
   1.1. 编写目的
   1.2. 文档范围
   1.3. 读者对象
   1.4. 术语表
2. 概述
   2.1. 设计模式定义
   2.2. 设计模式起源
   2.3. 设计模式分类
3. 经典设计模式
   3.1. 创建型模式
      3.1.1. 单例模式
      3.1.2. 工厂方法模式
      3.1.3. 抽象工厂模式
      3.1.4. 建造者模式
      3.1.5. 原型模式
   3.2. 结构型模式
      3.2.1. 适配器模式
      3.2.2. 桥接模式
      3.2.3. 组合模式
      3.2.4. 装饰器模式
      3.2.5. 外观模式
      3.2.6. 享元模式
      3.2.7. 代理模式
   3.3. 行为型模式
      3.3.1. 责任链模式
      3.3.2. 命令模式
      3.3.3. 解释器模式

3.3.4. 迭代器模式

3.3.5. 中介者模式

3.3.6. 备忘录模式

3.3.7. 观察者模式

3.3.8. 状态模式

3.3.9. 策略模式

3.3.10. 模板方法模式

3.3.11. 访问者模式

4. 项目设计模式

OpenCourse 官方图标

1. 文档引言

1.1. 编写目的

本文档为设计模式报告（DPR），基于《总体设计报告》中提出的设计方案补充软件体系的设计模式，为系统的实际软件开发提供指导

1.2. 文档范围

OpenCourse 设计模式报告主要包含了该软件系统整体设计模式分析与规划

1.3. 读者对象

OpenCourse 开发团队成员，验收者

1.4. 术语表

术语	定义
设计模式 (Design Pattern)	针对软件设计中常见问题的可复用解决方案，是一种最佳实践模板
模式名称 (Pattern Name)	设计模式的标准术语，用于开发者交流
问题 (Problem)	描述设计模式适用的常见设计问题
解决方案 (Solution)	提供代码结构和实现思路，不涉及具体语言
效果 (Consequences)	分析设计模式的优缺点
四人帮 (Gang of Four, GoF)	提出23种经典设计模式的四位作者
单例模式 (Singleton)	确保一个类只有一个实例，并提供全局访问点
工厂方法模式 (Factory Method)	定义一个创建对象的接口，由子类决定实例化哪个类
抽象工厂模式 (Abstract Factory)	提供接口创建相关或依赖对象的家族，无需指定具体类
建造者模式 (Builder)	分步骤构造复杂对象，允许相同的构建过程创建不同的表示形式
原型模式 (Prototype)	通过复制现有对象（原型）来创建新对象，避免重复初始化
适配器模式 (Adapter)	允许不兼容接口的类协同工作，通过转换接口实现适配
桥接模式 (Bridge)	将抽象部分与实现部分分离，使它们可以独立变化
组合模式 (Composite)	将对象组合成树形结构，表示部分–整体层次关系
装饰器模式 (Decorator)	动态地向对象添加额外职责，不改变其结构
外观模式 (Facade)	提供一个统一的简化接口来访问复杂子系统中的多个接口
享元模式 (Flyweight)	通过共享对象高效支持大量细粒度对象的复用，减少内存占用

代理模式 (Proxy)	为其他对象提供代理以控制对其访问，充当客户端和目标对象中介
责任链模式 (Chain of Responsibility)	允许请求沿处理链传递，直到有一个处理者能够处理
命令模式 (Command)	将请求封装为独立对象，支持参数化、撤销和重做操作
解释器模式 (Interpreter)	定义语言的文法表示，并提供解释器处理该语言的句子
迭代器模式 (Iterator)	提供一种方式顺序访问聚合对象中的元素，隐藏内部实现细节
中介者模式 (Mediator)	封装一组对象间的交互，降低对象间的直接耦合
备忘录模式 (Memento)	在不破坏封装性的前提下捕获并保存对象的内部状态以便后续恢复
观察者模式 (Observer)	定义对象间的一对多依赖关系，当一个对象状态改变时，所有依赖者自动更新
状态模式 (State)	允许对象在其内部状态改变时改变行为
策略模式 (Strategy)	定义一系列算法，封装每个算法，使它们可以互相替换
模板方法模式 (Template Method)	在父类中定义算法骨架，将某些步骤延迟到子类实现
访问者模式 (Visitor)	将算法与对象结构分离，在不修改元素类的前提下定义新操作

---

2. 概述

2.1. 设计模式定义

在软件工程中，设计模式（ Design Pattern ）是针对软件设计中常见问题的可复用解决方案，它并不是可以直接转换成代码的完整实现，而是一种最佳实践模板，帮助开发者设计更灵活、可维护和可扩展的软件系统

设计模式类似于建筑领域的蓝图，提供了一套经过验证的结构，使开发者能够避免重复造轮子，而是利用已有的经验来优化代码设计，增强代码的复用性和可维护性，提升系统的扩展性，让开发者

高效沟通

每个设计模式通常包含以下几个关键部分：

1. 模式名称：提供一个标准术语，方便开发者交流
2. 问题：描述设计模式适用于哪些常见设计问题
3. 解决方案：提供代码结构和关键实现思路，但不涉及具体编程语言
4. 效果：分析该设计模式的优缺点

2.2. 设计模式起源

设计模式的概念最早由“四人帮”（Gang of Four, GoF）在 1994 年的经典书籍 《Design Patterns: Elements of Reusable Object-Oriented Software》 中系统化提出，该书归纳了 23 种经典设计模式，并分为**创建型**、**结构型**和**行为型**三大类

2.3. 设计模式分类

GoF 的 23 种模式按用途分为三类：

1. 创建型模式（Creational Patterns）：关注**对象创建机制**（如 `Singleton` 确保唯一实例，`Factory` 解耦实例化过程）
2. 结构型模式（Structural Patterns）：关注**类和对象的组合**（如 `Adapter` 兼容接口，`Decorator` 动态扩展功能）
3. 行为型模式（Behavioral Patterns）：关注**对象间通信**（如 `Observer` 实现事件通知，`Strategy` 封装算法族）

---

3. 经典设计模式

3.1. 创建型模式

创建型模式（Creational Patterns）主要提供对象的创建机制，增加灵活性和复用性

3.1.1. 单例模式

单例模式（Singleton）确保一个类只有一个实例，并提供全局访问点来获取该实例，其核心思想是**控制实例数量，防止外部通过 `new` 创建多个对象**，并提供静态方法（如 `getInstance()`）以访问唯一实例

单例模式的适用场景通常为

- 数据库连接池（避免重复连接）
- 日志系统（全局共享日志文件）
- 配置管理器（统一读取配置文件）

对于需要多实例的类（如多线程任务）和需要频繁创建销毁对象的情况，单例模式通常情况啊不适用

示例：单例模式 Java

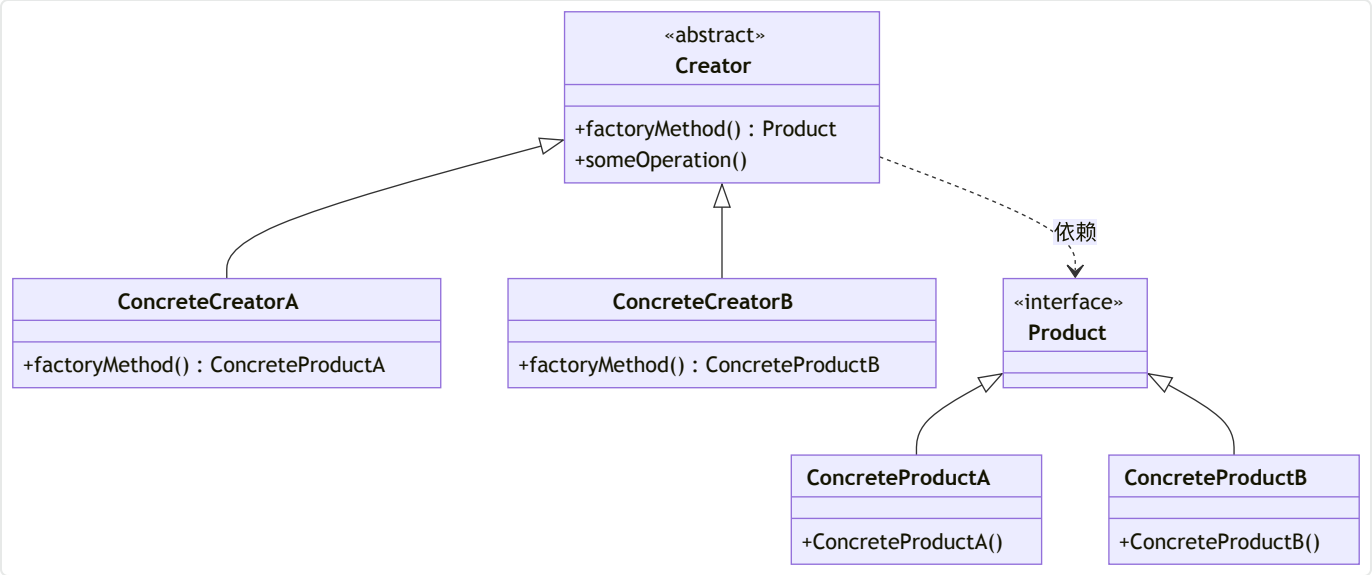
```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {} // 私有构造方法

    // 加锁保证线程安全
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

3.1.2. 工厂方法模式

工厂方法模式（Factory Method Pattern）负责定义一个创建对象的接口，由子类决定实例化哪个类，即让类的实例化推迟到子类，其核心思想是**将对象创建于使用分离以实现解耦**，通过子类实现具体对象的多态创建，同时使用工厂方法模式可以在新增产品类时无需修改客户端代码而实现良好扩展性

工厂方法模式的适用场景通常为

- 无法预知需要创建哪种具体对象
- 扩展系统时不影响现有代码
- 需要将对象创建逻辑集中管理
- 框架需要为不同实现提供扩展点


«abstract»
Creator
+factoryMethod() : Product
+someOperation()
依赖
ConcreteCreatorA
+factoryMethod() : ConcreteProductA
ConcreteCreatorB
+factoryMethod() : ConcreteProductB
«interface»
Product
ConcreteProductA
+ConcreteProductA()
ConcreteProductB
+ConcreteProductB()

示例：工厂方法模式 Java

```java
// 抽象产品负责定义产品接口
interface Button {
    void render();
    void onClick();
}

// 具体产品负责实现产品接口
class WindowsButton implements Button {
    public void render() { System.out.println("Render Windows button"); }
    public void onClick() { System.out.println("Windows button clicked");
}
}

class HTMLButton implements Button {
    public void render() { System.out.println("Render HTML button"); }
    public void onClick() { System.out.println("HTML button clicked"); }
}

// 抽象工厂用于声明工厂方法
abstract class Dialog {
    abstract Button createButton(); // 工厂方法

    void renderWindow() {
        Button button = createButton();
        button.render();
        button.onClick();
    }
}

// 具体工厂用于实现工厂方法，返回具体产品
class WindowsDialog extends Dialog {
    Button createButton() {
        return new WindowsButton();
    }
}

class WebDialog extends Dialog {
    Button createButton() {
        return new HTMLButton();
    }
}

// 客户端调用工厂方法
public class Client {
    public static void main(String[] args) {
```

```
        Dialog dialog;

        // 根据配置或环境选择具体工厂
        if (System.getProperty("os.name").contains("Windows")) {
            dialog = new WindowsDialog();
        } else {
            dialog = new WebDialog();
        }

        dialog.renderWindow();
    }
}
```

3.1.3. 抽象工厂模式

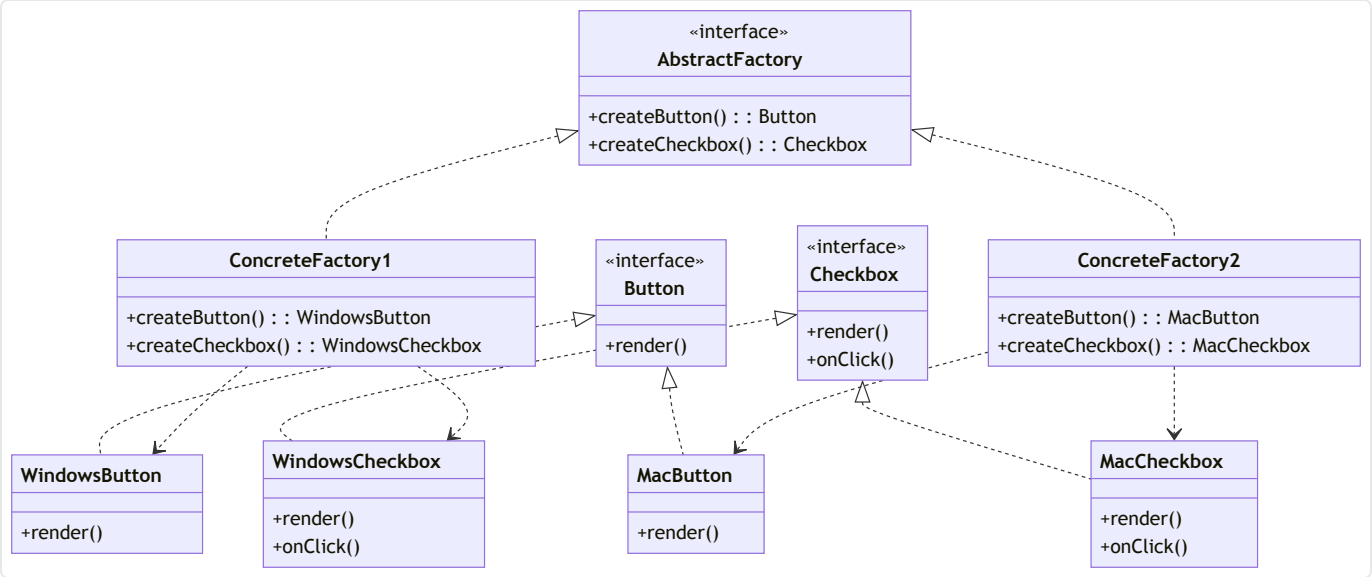
抽象工厂模式（Abstract Factory Pattern）提供接口用于创建相关或依赖对象的家族，而无需指定具体类，其核心思想是**构建一个产品家族以创建一组相互关联和依赖的对象**（如 UI 组件：按钮+文本框+复选框），客户端代码只通过抽象工厂的抽象接口和抽象产品交互

抽象工厂模式适用于以下场景：

- 系统需要创建多个产品系列（如不同操作系统风格的 UI 控件）
- 需要确保产品兼容性
- 希望隐藏具体产品创建细节，客户端只接触抽象接口

工厂方法模式 VS 抽象工厂模式

- 工厂方法模式是创建单一产品，通过子类决定实例化哪个类
- 抽象工厂模式是创建产品家族，通过组合多个工厂方法实现



示例：抽象工厂模式 Java

```java
// 抽象产品：按钮
interface Button {
    void render();
}

// 具体产品：Windows按钮
class WindowsButton implements Button {
    public void render() {
        System.out.println("Render a Windows-style button");
    }
}

// 具体产品：Mac按钮
class MacButton implements Button {
    public void render() {
        System.out.println("Render a Mac-style button");
    }
}

// 抽象产品：复选框
interface Checkbox {
    void render();
    void onClick();
}

// 具体产品：Windows复选框
class WindowsCheckbox implements Checkbox {
    public void render() {
        System.out.println("Render a Windows-style checkbox");
    }
    public void onClick() {
        System.out.println("Windows checkbox clicked");
    }
}

// 具体产品：Mac复选框
class MacCheckbox implements Checkbox {
    public void render() {
        System.out.println("Render a Mac-style checkbox");
    }
    public void onClick() {
        System.out.println("Mac checkbox clicked");
    }
}

```

```java
// 抽象工厂
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

// 具体工厂：Windows风格
class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

// 具体工厂：Mac风格
class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }
    public Checkbox createCheckbox() {
        return new MacCheckbox();
    }
}

// 客户端
public class Application {
    private Button button;
    private Checkbox checkbox;

    public Application(GUIFactory factory) {
        button = factory.createButton();
        checkbox = factory.createCheckbox();
    }

    public void renderUI() {
        button.render();
        checkbox.render();
        checkbox.onClick();
    }

    public static void main(String[] args) {
        // 根据配置选择工厂
        GUIFactory factory;
        String osName = System.getProperty("os.name").toLowerCase();

        if (osName.contains("win")) {
```

```
            factory = new WindowsFactory();
        } else {
            factory = new MacFactory();
        }

        Application app = new Application(factory);
        app.renderUI();
    }
}
```

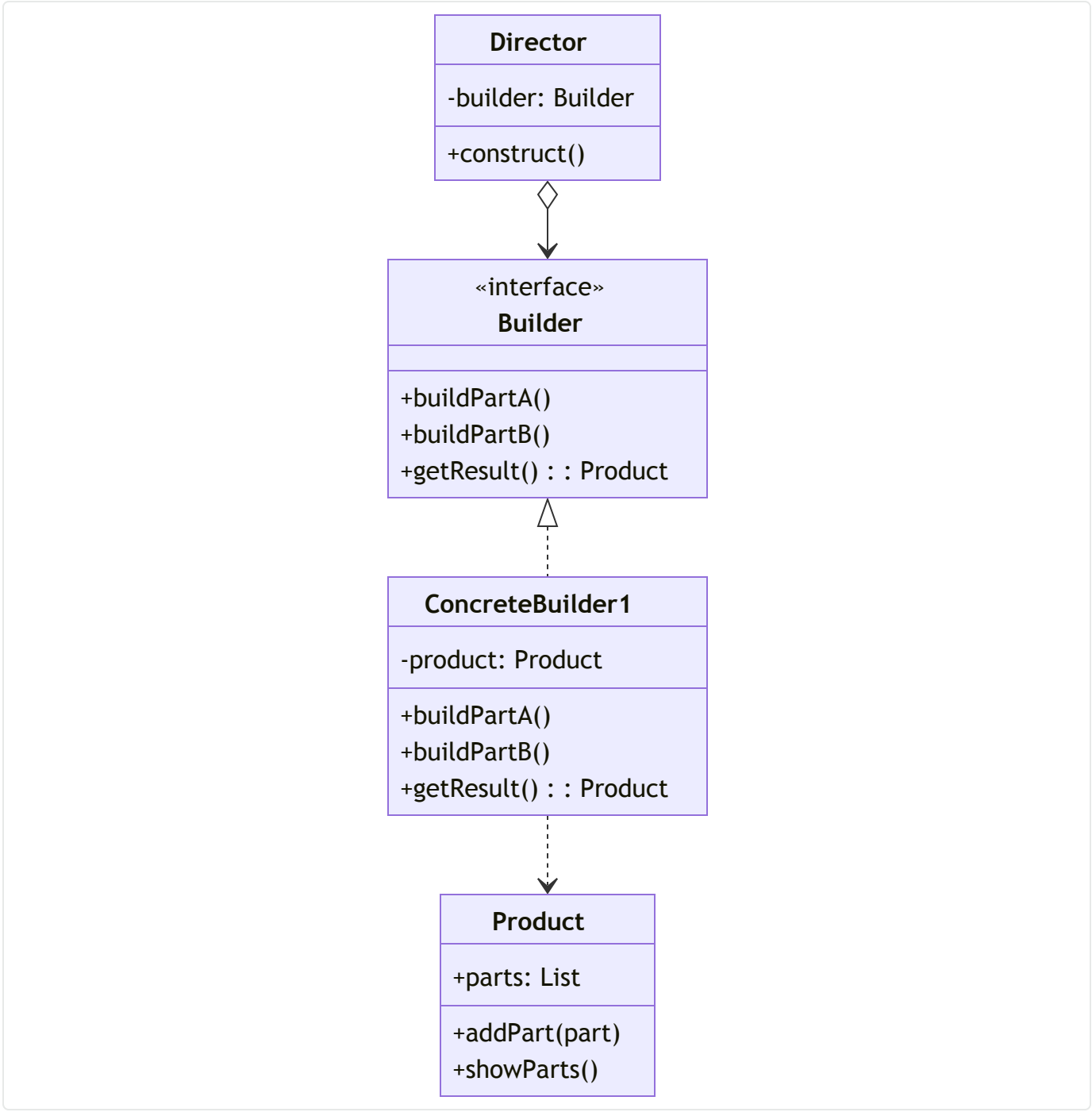
3.1.4. 建造者模式

建造者模式（Builder Pattern）负责分步骤构造复杂对象，允许相同的构建过程创建不同的表示形式，其核心思想是**将对象的构造过程与对象本身分离**，通过链式调用逐步设置属性，使用者可以创建不同风格的对象而无需修改构造代码

建造者模式适用于以下场景：

- 需要创建包含多个组成部分的复杂对象
- 对象的构造过程需要不同的实现
- 希望创建不可变对象

对于对象属性很少的情况或构造过程不发生变化的情况，建造者模式会造成冗余


Director
-builder: Builder
+construct()
«interface»
Builder
+buildPartA()
+buildPartB()
+getResult() : : Product
ConcreteBuilder1
-product: Product
+buildPartA()
+buildPartB()
+getResult() : : Product
Product
+parts: List
+addPart(part)
+showParts()

示例：建造者模式 Java

```java
// 产品：电脑
class Computer {
    private String cpu;
    private String ram;
    private String ssd;

    // 私有构造，只能通过Builder创建
    private Computer(Builder builder) {
        this.cpu = builder.cpu;
        this.ram = builder.ram;
        this.ssd = builder.ssd;
    }

    // 静态Builder类
    public static class Builder {
        private String cpu; // 必选
        private String ram; // 必选
        private String ssd = "512GB"; // 可选（默认值）

        public Builder(String cpu, String ram) {
            this.cpu = cpu;
            this.ram = ram;
        }

        public Builder setSSD(String ssd) {
            this.ssd = ssd;
            return this; // 返回this实现链式调用
        }

        public Computer build() {
            return new Computer(this);
        }
    }

    @Override
    public String toString() {
        return String.format("Computer[CPU=%s, RAM=%s, SSD=%s]", cpu, ram,
 ssd);
    }
}

// 客户端使用
public class Client {
    public static void main(String[] args) {
        Computer gamingPC = new Computer.Builder("Intel i9", "32GB")
```

```
                                .setSSD("1TB") // 可选配置
                                .build();

        System.out.println(gamingPC);
        // 输出: Computer[CPU=Intel i9, RAM=32GB, SSD=1TB]
    }
}
```

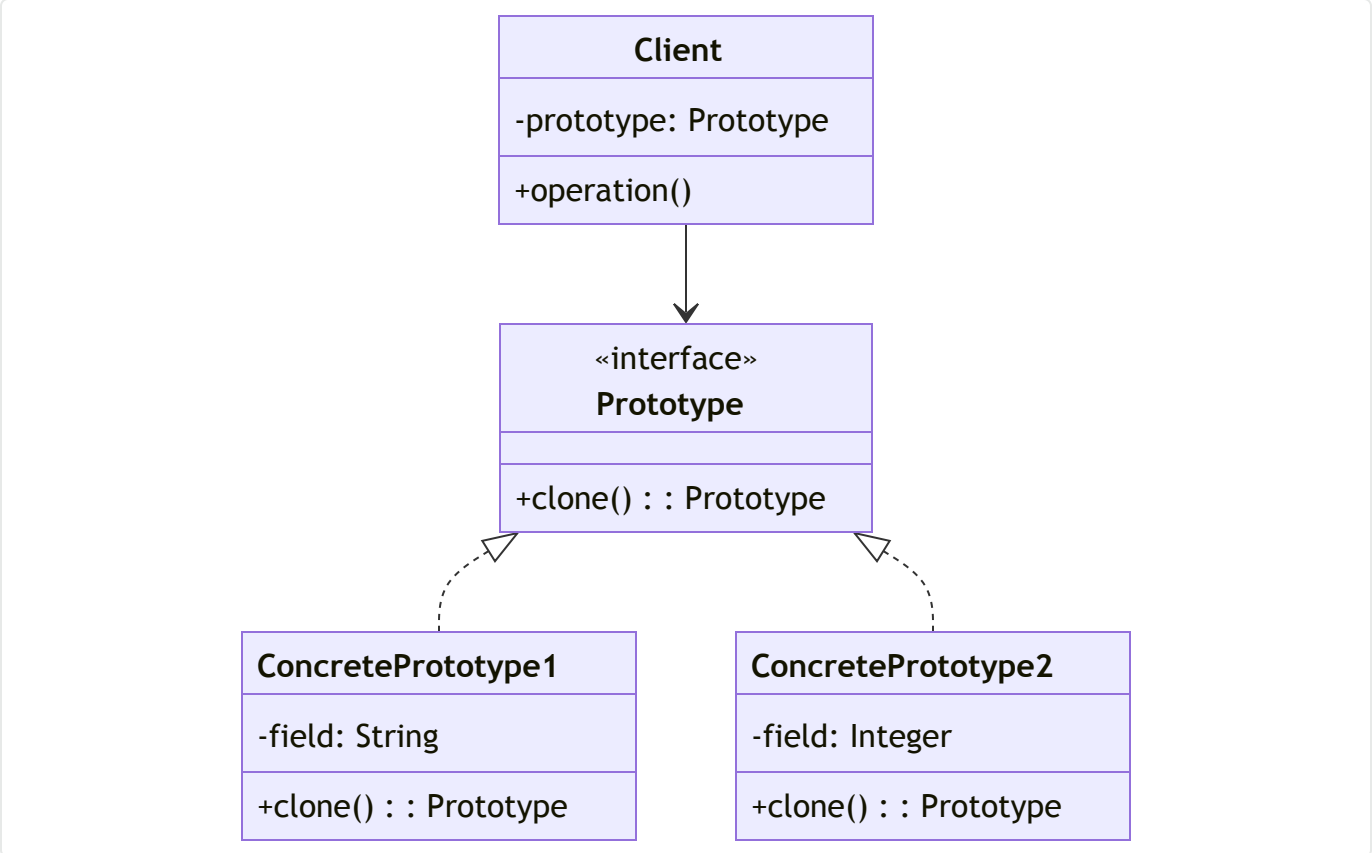
3.1.5. 原型模式

原型模式（Prototype Pattern）通过复制现有对象（原型）来创建新对象，而不是通过 `new` 实例化，其核心思想是**通过克隆代替新建以避免重复初始化操作**，提高性能，由运行时决定克隆对象类型

原型模式的适用场景如下：

- 对象创建成本高（如数据库查询结果缓存）
- 系统需要动态配置对象（如游戏中敌人生成）
- 需要避免构造函数约束（绕过私有构造函数）
- 对象状态频繁变化，但初始状态相同

原型模式的优化对象创建的利器，适合高性能复制和动态对象生成的场景



示例：原型模式 Java

```java
import java.util.HashMap;
import java.util.Map;

// 原型注册表（管理常用原型）
class PrototypeRegistry {
    private static Map<String, Prototype> prototypes = new HashMap<>();

    static {
        prototypes.put("default", new ConcretePrototype("Default"));
        prototypes.put("custom", new ConcretePrototype("Custom"));
    }

    public static Prototype getPrototype(String type) {
        return prototypes.get(type).clone();
    }
}

// 客户端通过注册表获取克隆对象
Prototype p = PrototypeRegistry.getPrototype("custom");
```

3.2. 结构型模式

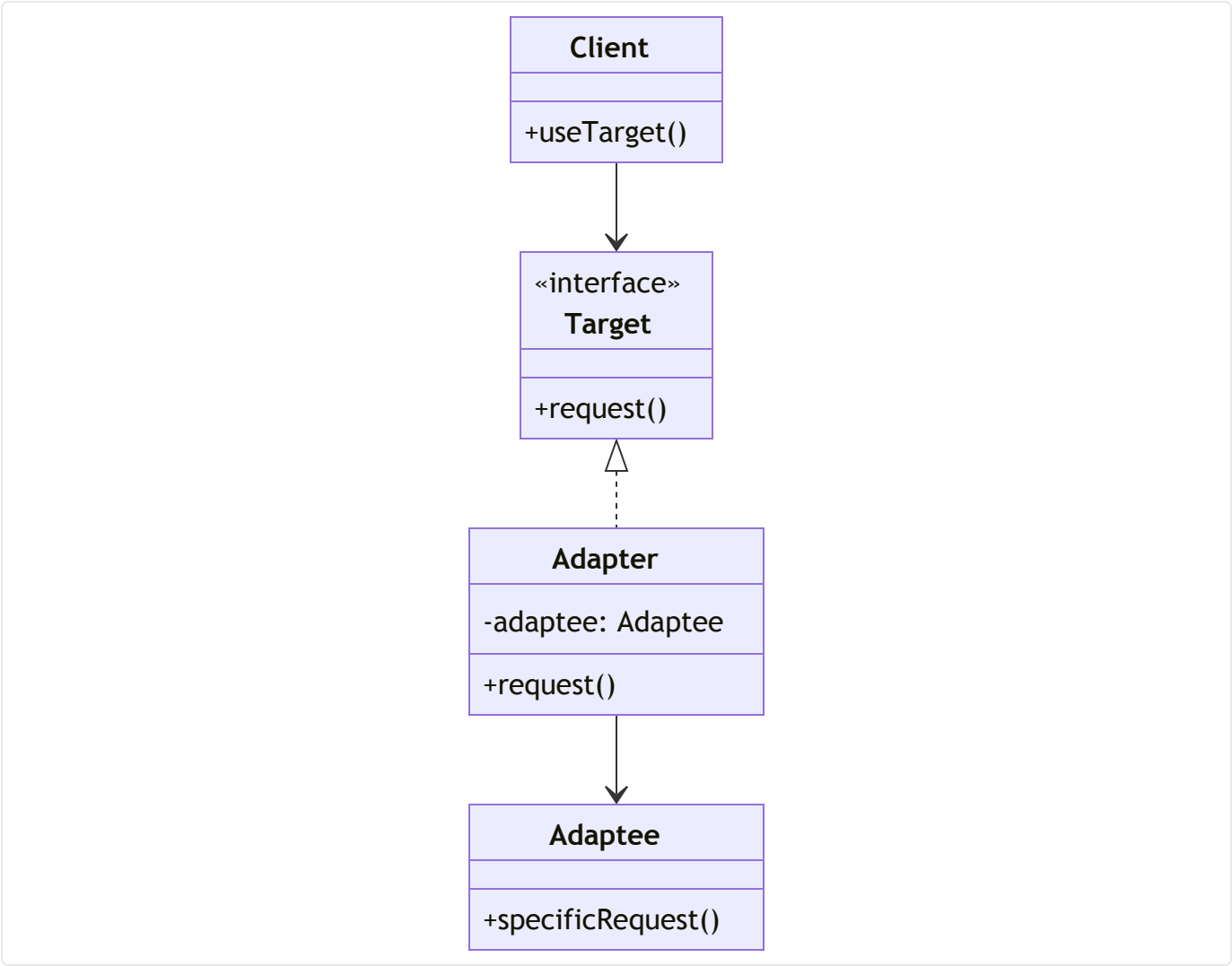
结构型模式（Structural Patterns）主要处理类或对象的组合，形成更大的结构

3.2.1. 适配器模式

适配器模式（Adaptor Pattern）允许不兼容接口的类能够协同工作，通过将一个类的接口转换成客户端期望的另一个接口实现不同接口适配，其核心思想是**充当两个不兼容接口间的接口转换**，通过复用现有类而无需修改原有代码即可整合新功能，客户端可以只与目标接口交互而不直接依赖被适配者

适配器模式的适用场景包括：

- 需要使用现有的接口不兼容的类
- 需要统一多个不同子系统的接口
- 希望创建一个可复用的类用于与未来类协作



示例：适配器模式 Java

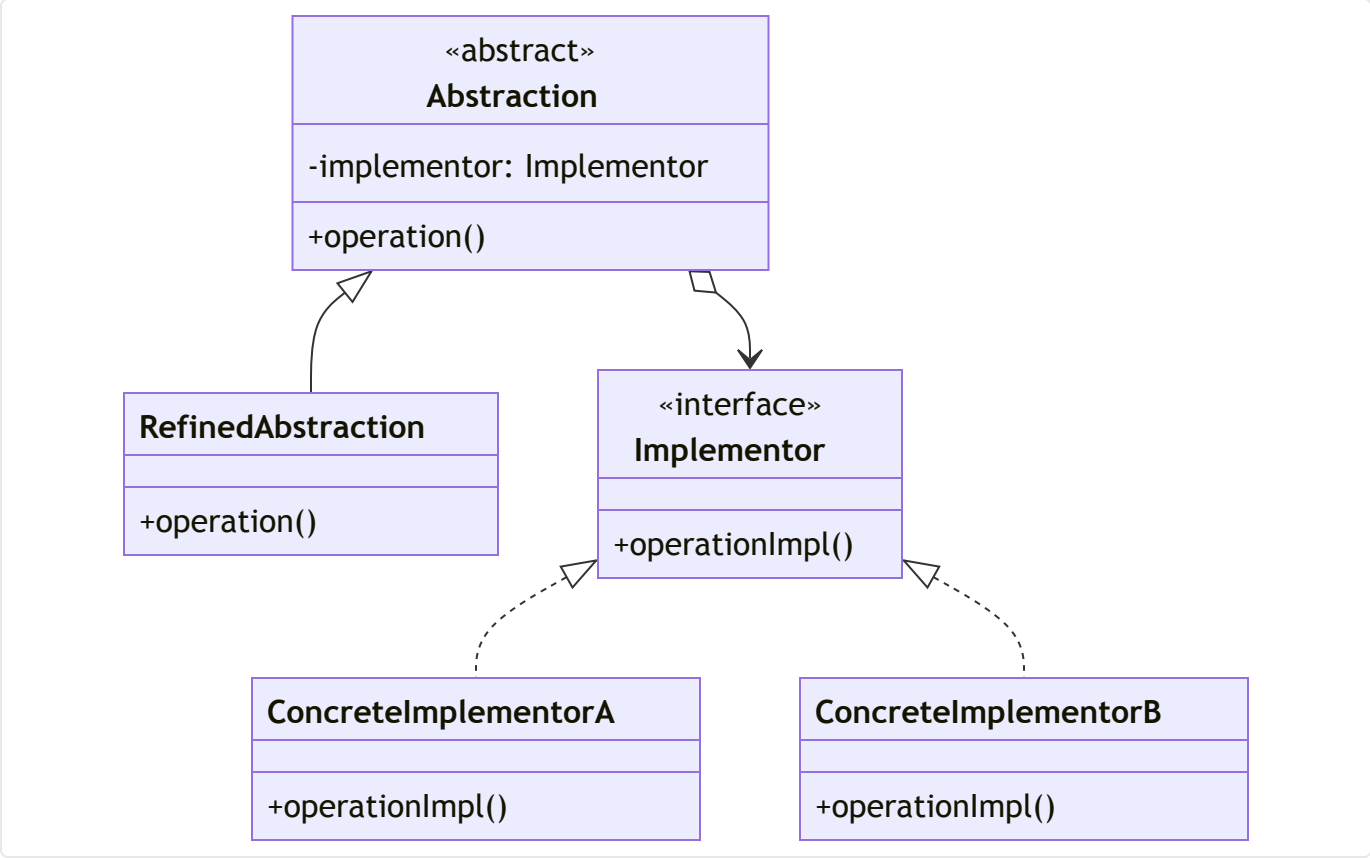
```java
// 将旧版Enumeration适配为新版Iterator
List<String> list = Arrays.asList("A", "B", "C");
Enumeration<String> en = Collections.enumeration(list);
Iterator<String> it = new EnumerationIteratorAdapter(en);

// 适配器实现
class EnumerationIteratorAdapter implements Iterator<String> {
    private Enumeration<String> en;

    public boolean hasNext() { return en.hasMoreElements(); }
    public String next() { return en.nextElement(); }
}
```

3.2.2. 桥接模式

桥接模式（Bridge Pattern）将抽象部分与实现部分分离，使它们可以独立变化，通过组合关系代替继承关系，其核心思想是**解耦抽象与实现，组合优于继承和运行时绑定**

桥接模式的适用场景包括：

- 需要避免抽象与实现之间的永久绑定（如支持多平台渲染的 UI 框架）
- 抽象和实现都应能通过子类化独立扩展
- 需要运行时切换实现（如动态更换数据库驱动）



示例：桥接模式 Java

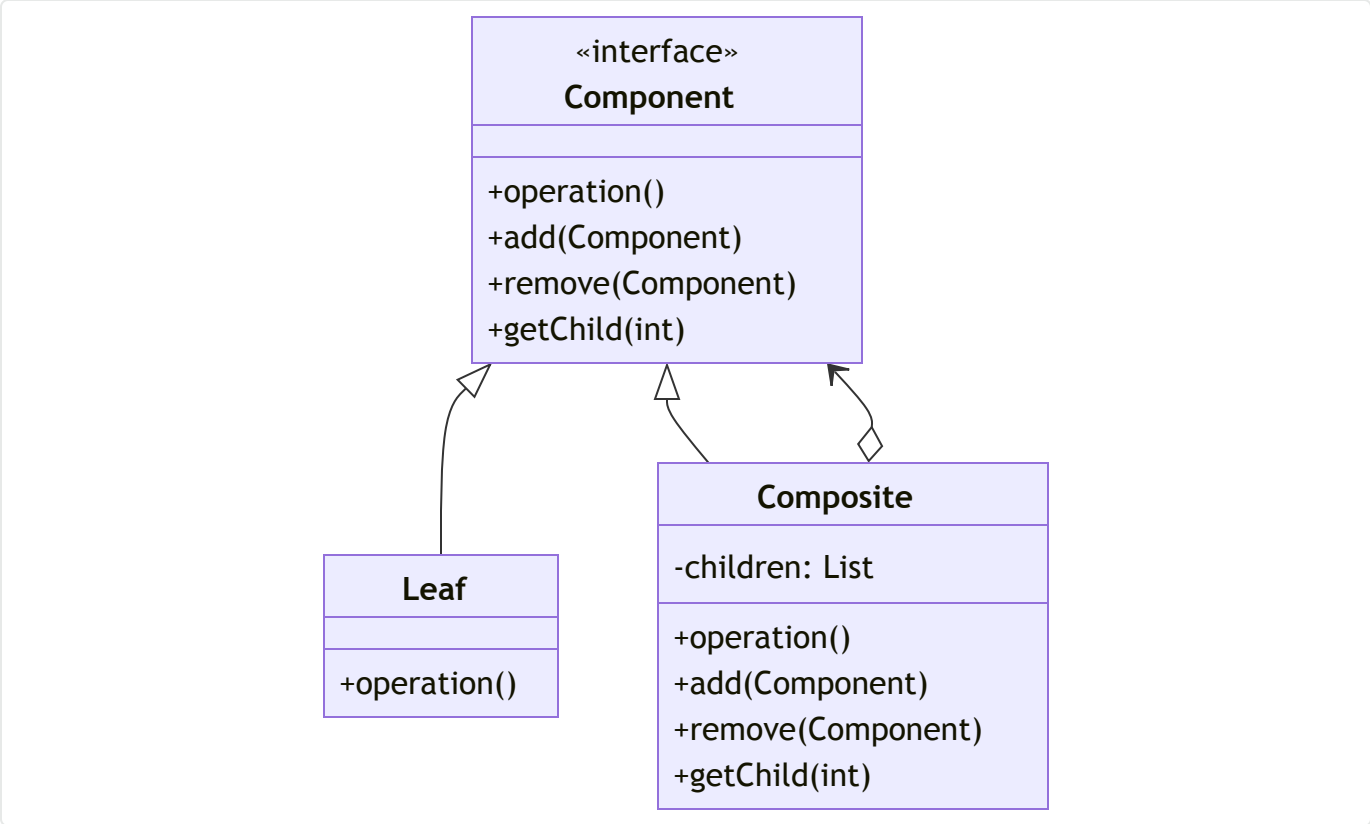
```java
// 抽象部分：数据库连接
Connection conn = DriverManager.getConnection(url);
// 实现部分：com.mysql.jdbc.Driver 或 oracle.jdbc.driver.OracleDriver
```

3.2.3. 组合模式

组合模式（Composite Pattern）允许你将对象组合成树形结构来表示部分–整体的层次关系，使得客户端可以统一处理单个对象和组合对象，其核心思想是**通过树形结构来表示层次关系**，对单个对象和组合对象使用统一接口

组合模式的适用场景包括：

- 需要表示部分–整体的层次结构
- 客户端希望忽略组合与单个对象的差异
- 需要递归处理树形结构的所有节点



示例：组合模式 Java

```java
interface Employee {
    void showDetails();
}

class Developer implements Employee { ... } // Leaf
class Manager implements Employee {      // Composite
    private List<Employee> subordinates;
    public void add(Employee e) { ... }
}
```

3.2.4. 装饰器模式

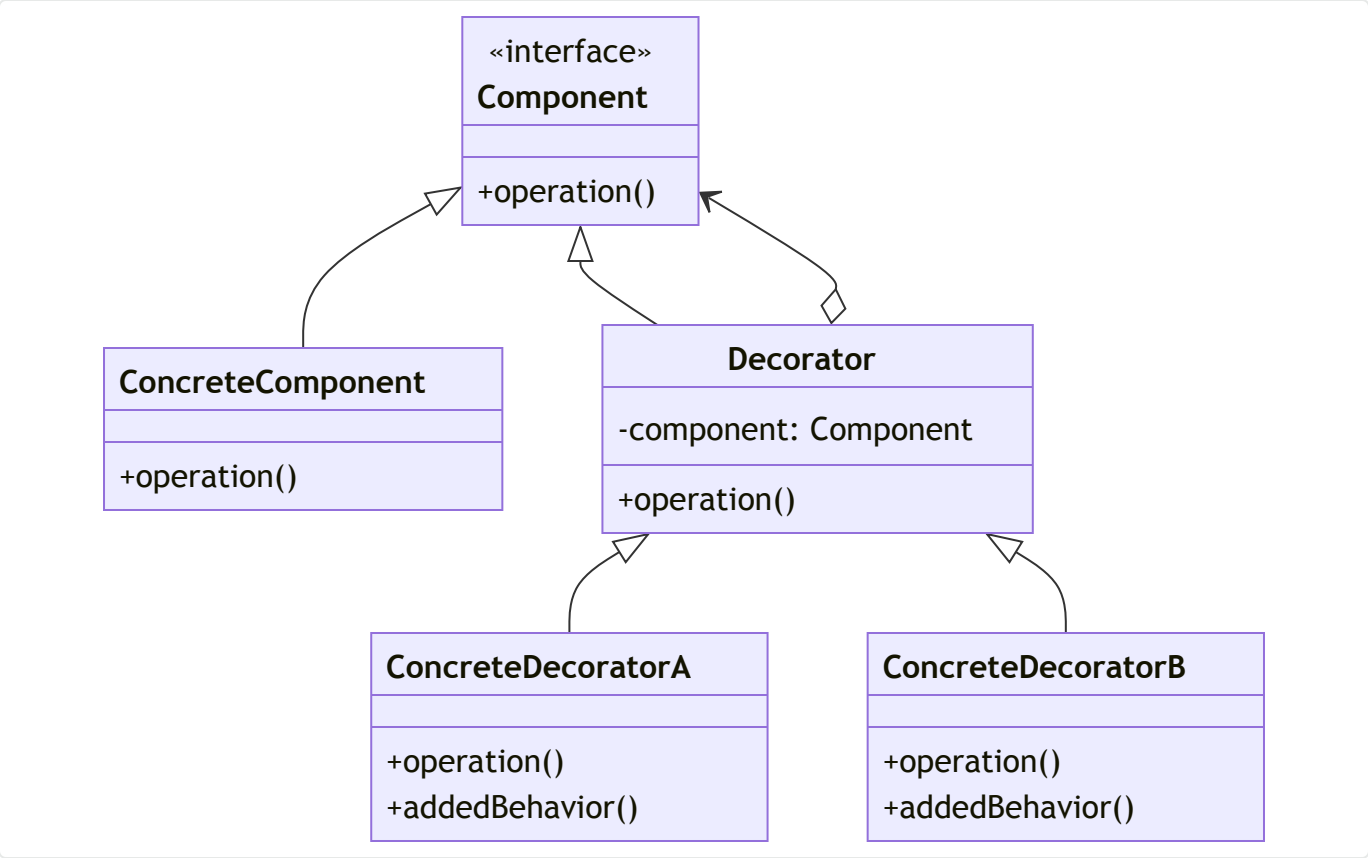
装饰器模式（Decorator Pattern）允许动态地向一个对象添加额外职责同时又不改变其结构，其核心思想是**通过创建包装对象的方式提供比继承更灵活的功能扩展**

装饰器模式的适用场景如下：

- 需要动态地给对象添加或撤销功能

- 当继承不适合扩展功能
- 需要不影响其他对象的情况下扩展单个对象的功能

需要注意的是，大量装饰器可能导致系统复杂，且对于最终类（`final` 类）或不可继承类无法进行不建议进行装饰



示例：装饰器模式 Java

```java
// 多层装饰的I/O流
InputStream in = new FileInputStream("data.txt");
in = new BufferedInputStream(in); // 添加缓冲功能
in = new DataInputStream(in);    // 添加基本类型读取功能
```

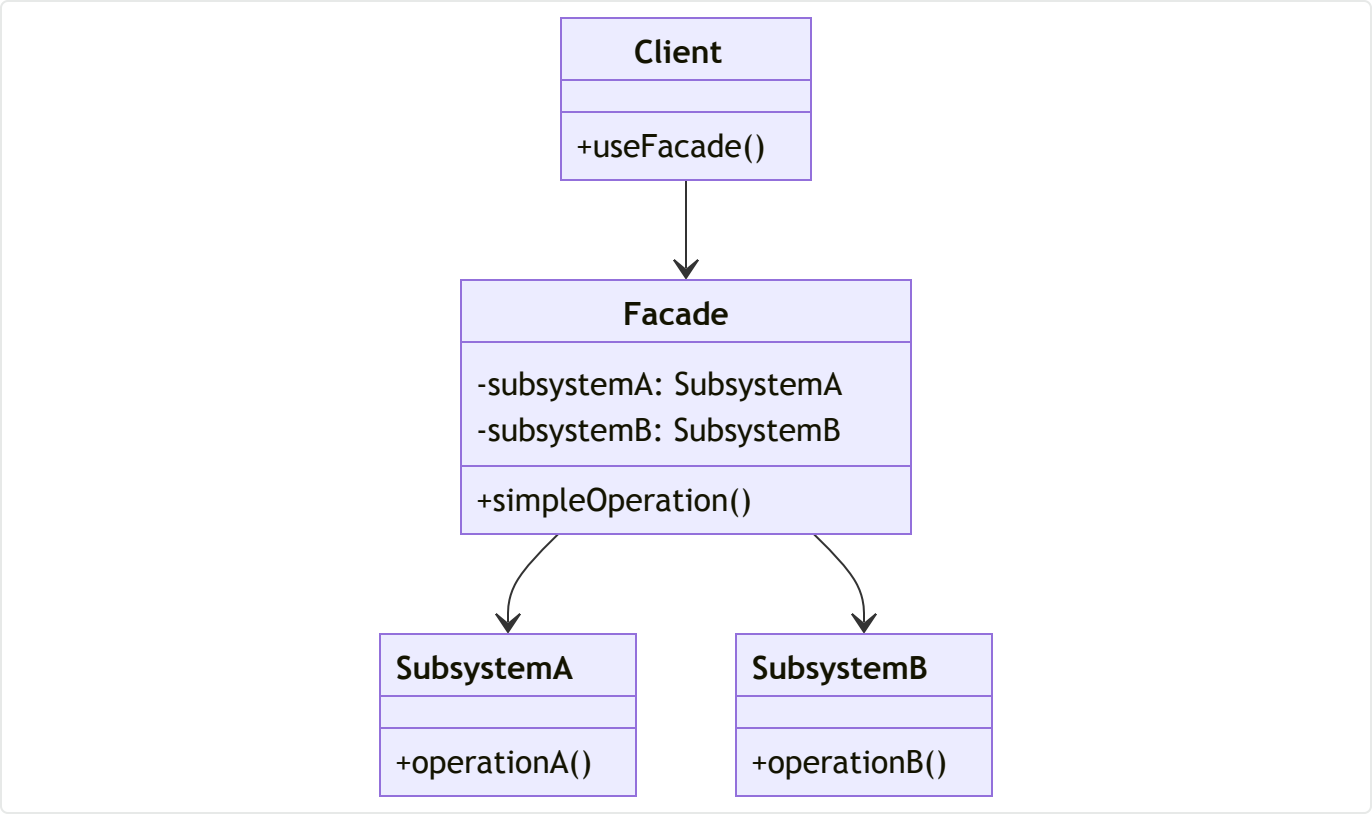
3.2.5. 外观模式

外观模式（Facade Pattern）提供了一个统一的简化接口来访问复杂子系统中的多个接口，其核心思想是**为复杂系统提供单一的高层接口**，解耦客户端与子系统的直接交互，将多个步骤的操作封装为简单调用

外观模式的适用场景如下：

- 需要为复杂子系统提供简单入口

- 客户端与多个子系统存在强耦合需要解耦
- 需要分层构建系统



示例：外观模式 Java

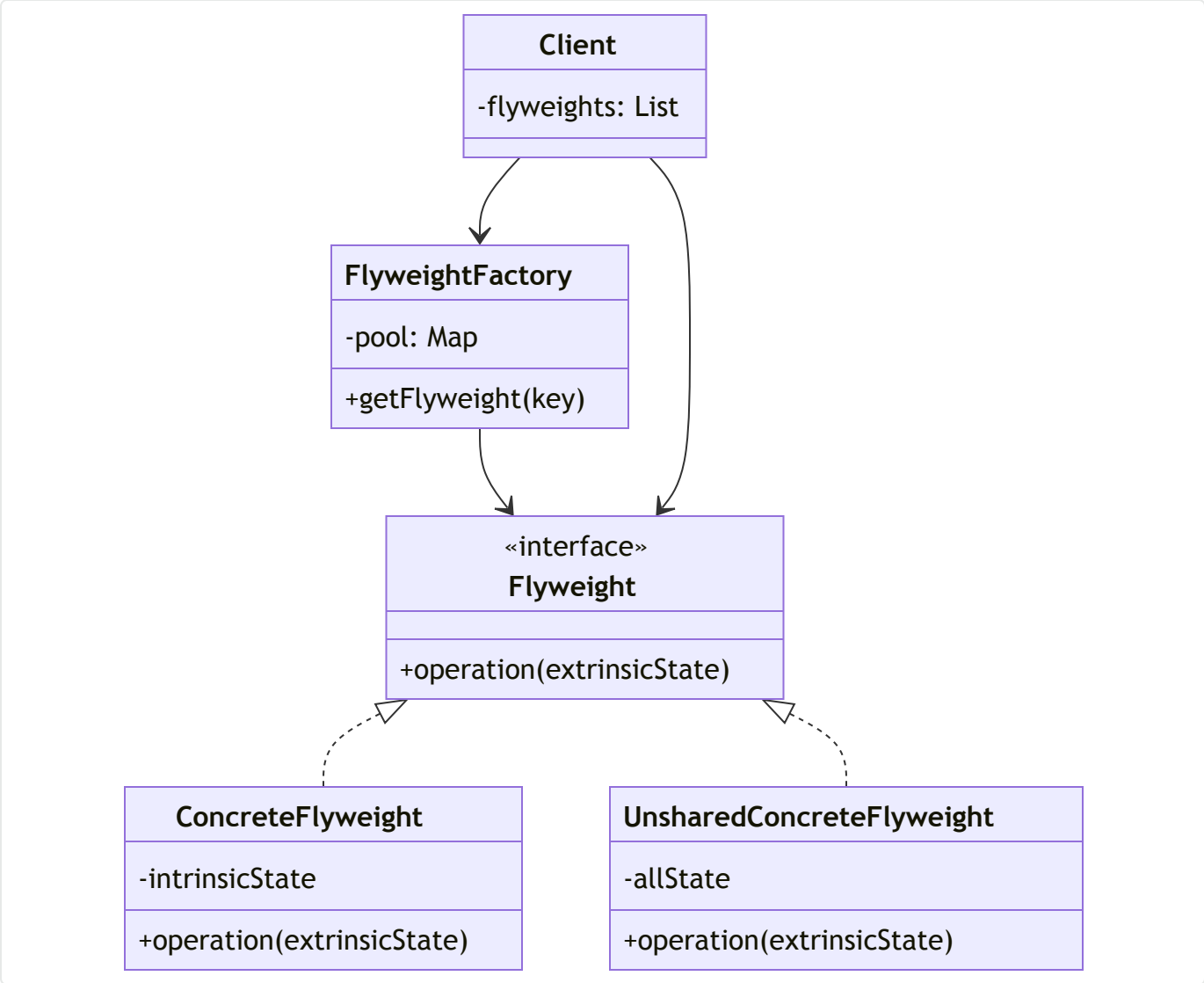
```java
// JDBC 操作封装
class JdbcFacade {
    public void executeQuery(String sql) {
        try (Connection conn = DriverManager.getConnection(url);
             Statement stmt = conn.createStatement()) {
            stmt.execute(sql);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

3.2.6. 享元模式

享元模式（Flyweight Pattern） 通过共享对象高效支持大量细粒度对象的复用，减少内存占用，其核心思想是**将对象的内部状态和外部状态分离**，维护共享对象的缓存池以利用共享对象来减少大量相似对象的内存占用，提高对象的复用性

享元模式的适用场景包括：

- 系统中存在大量相似对象，这些对象的创建和销毁成本较高
- 对象的大部分状态可以外部化，从而减少对象的内存占用
- 需要减少内存占用，提高系统性能



示例：享元模式 Java

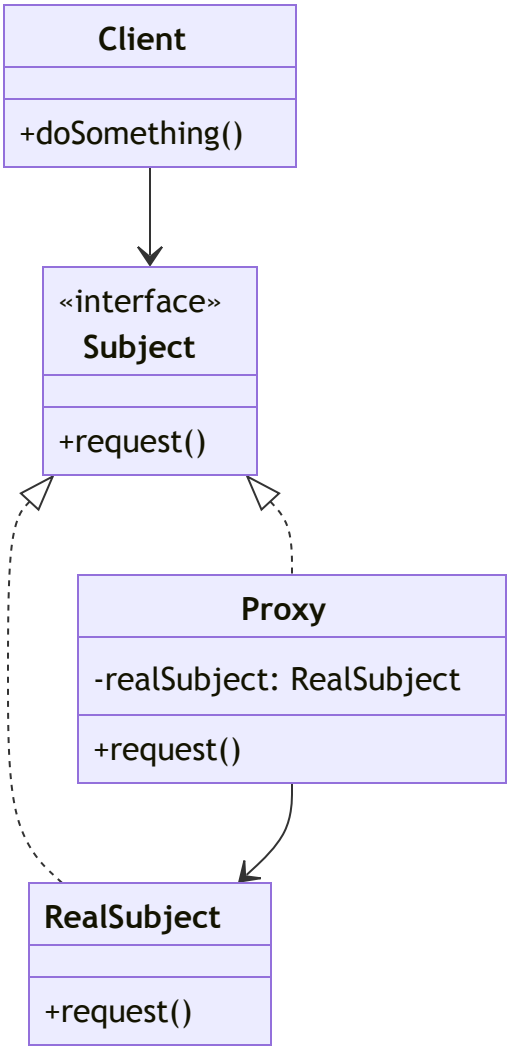
```java
// Integer.valueOf()使用享元池（-128~127）
Integer a = Integer.valueOf(127); // 从缓存获取
Integer b = Integer.valueOf(127);
System.out.println(a == b); // true
```

3.2.7. 代理模式

代理模式（Proxy Pattern）为其它对象提供一种代理以控制对这个对象的访问，代理对象在客户端和目标对象之间起到中介作用，其核心思想是**通过代理控制对目标对象的访问**

代理模式的适用场景包括：

- 远程代理：为远程对象提供本地代表（如 RPC 调用）
- 虚拟代理：延迟初始化大资源对象（如图片懒加载）
- 保护代理：控制对敏感对象的访问权限
- 智能应用：添加对象使用时的额外操作（如引用计数等）



示例：代理模式 Java

```java
// Collections.unmodifiableList()返回保护代理
List<String> list = new ArrayList<>();
List<String> proxyList = Collections.unmodifiableList(list);
proxyList.add("test"); // 抛出UnsupportedOperationException
```

3.3. 行为型模式

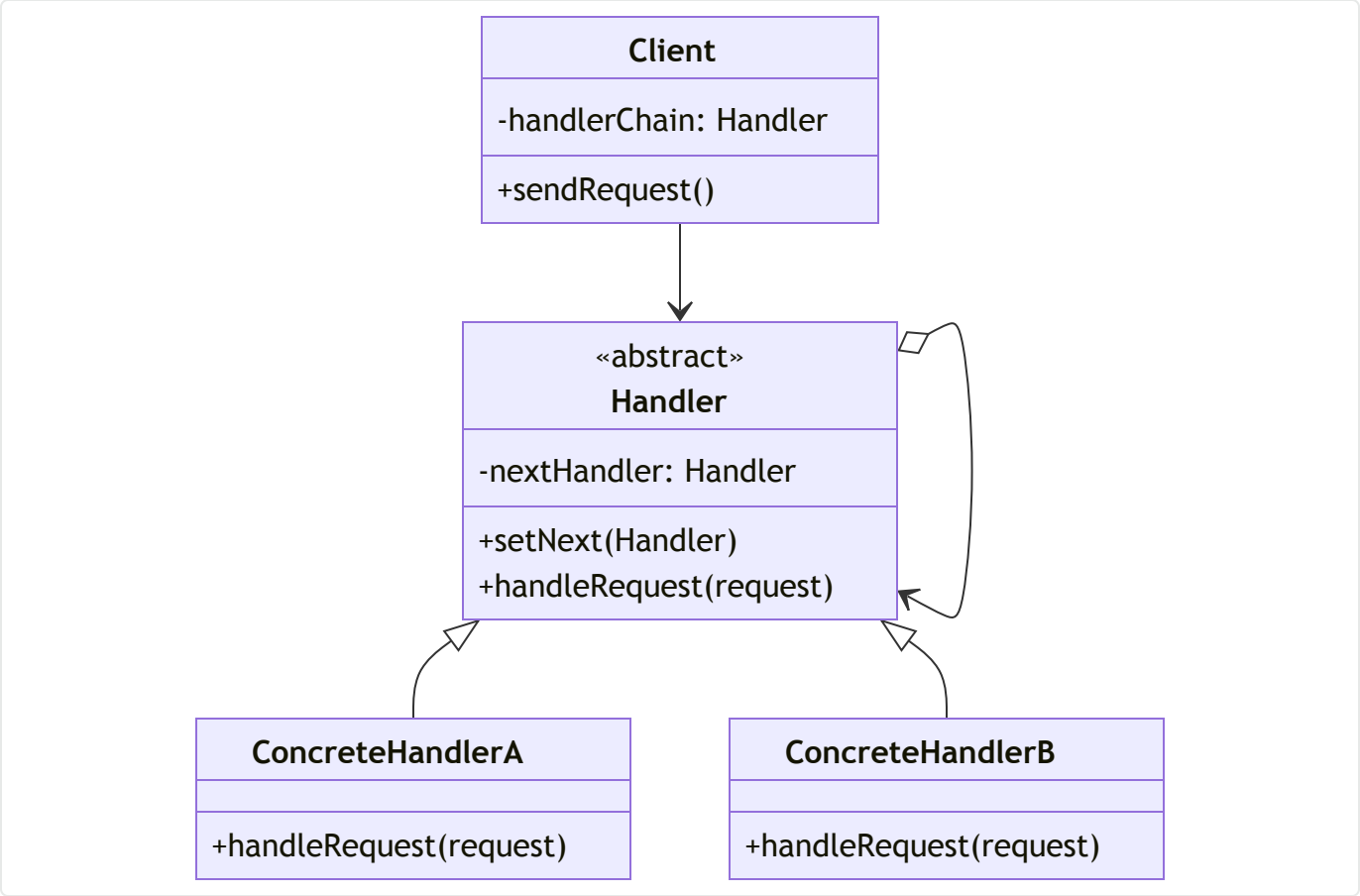
行为型模式（Behavioral Patterns）主要管理对象间的交互和职责分配

3.3.1. 责任链模式

责任链模式（Chain of Responsibility Pattern）允许你将请求沿着处理链传递，直到有一个处理者能够处理，每个处理这决定自己处理请求或将其传递给链中的下一个处理者，其核心思想是**解耦发送者与接收者使得发送者无需知道具体由谁处理**

责任链模式的适用场景包括：

- 需要多个对象处理同意请求，但具体处理者在运行时确定
- 希望动态指定一组对象处理请求
- 需要按顺序尝试不同的处理方式（如异常处理）



示例：责任链模式 Java

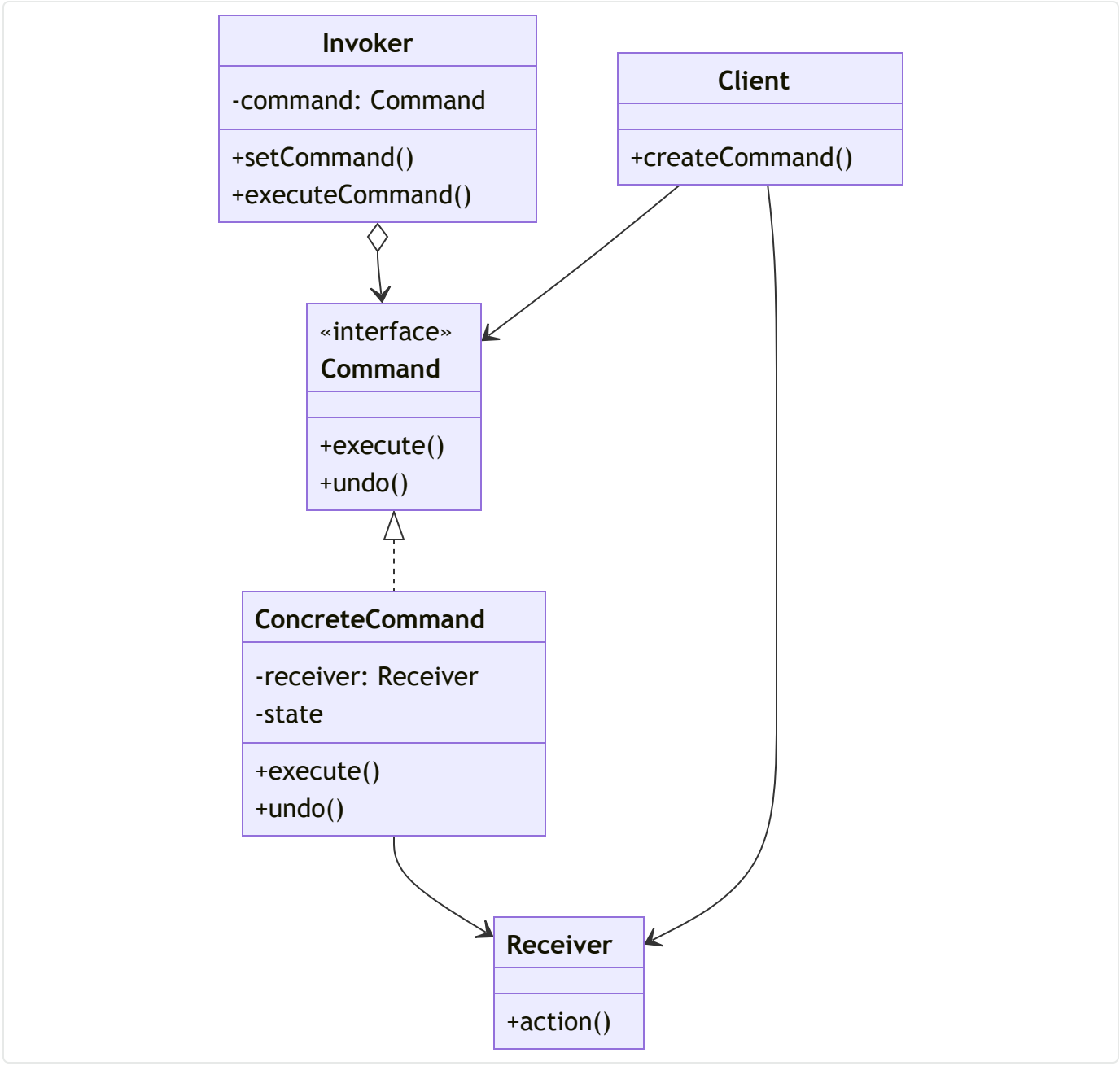
```java
try {
    // 业务代码
} catch (NullPointerException e) {
    // 处理空指针
} catch (IllegalArgumentException e) {
    // 处理参数错误
} catch (Exception e) {
    // 兜底处理
}
```

3.3.2. 命令模式

命令模式（Command Pattern）将请求封装为独立对象，使操作者可以参数化客户端使用的不同请求，其核心思想是**解耦调用者与执行者使调用者无需知道具体执行细节**，同时把请求操作封装为包含具体信息的独立对象

命令模式适用场景包括：

- 需要参数化对象
- 需要支持撤销和重做功能
- 需要队列请求或记录请求日志
- 需要支持事务


Invoker
-command: Command
+setCommand()
+executeCommand()
Client
+createCommand()
«interface»
Command
+execute()
+undo()
ConcreteCommand
-receiver: Receiver
-state
+execute()
+undo()
Receiver
+action()

示例：命令模式 Java

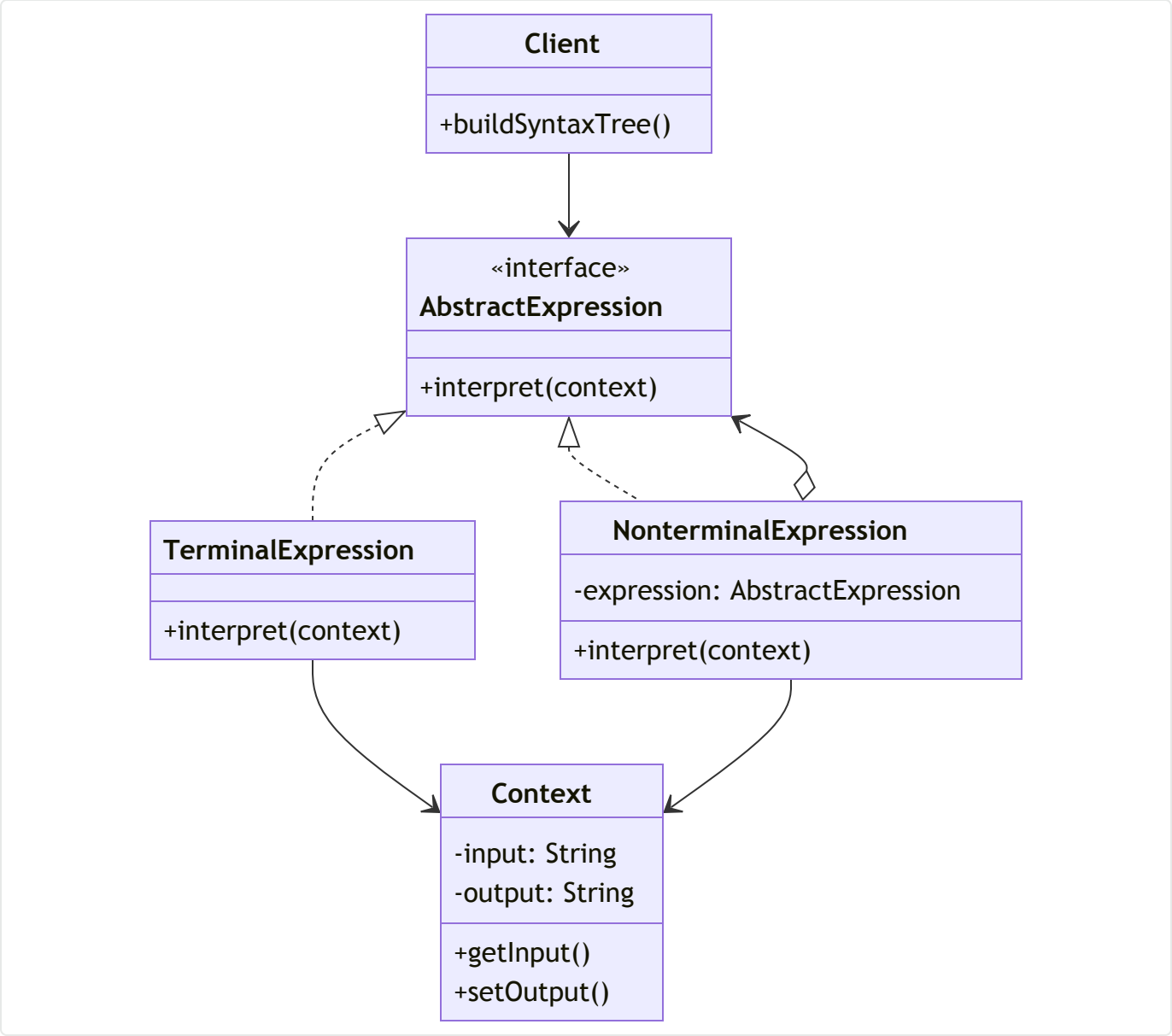
```java
// 事务管理器
class TransactionManager {
    private List<Command> commands = new ArrayList<>();

    public void addCommand(Command cmd) {
        commands.add(cmd);
    }

    public void commit() {
        try {
            for (Command cmd : commands) {
                cmd.execute();
            }
        } catch (Exception e) {
            for (int i = commands.size()-1; i >= 0; i--) {
                commands.get(i).undo();
            }
        }
    }
}
```

3.3.3. 解释器模式

解释器模式（Interpreter Pattern）定义了一个语言的文法表示，并提供一个解释器来处理这个语句中的句子，主要用于解析特定语法规则的文本输入，其核心思想是**语法解析将特定语言表示为语法树**，通过组合多个表达式对象完成复杂解析

解释器模式的适用场景包括：

- 需要解释执行特定语法规则的语言
- 语法相对简单且固定
- 需要频繁解析和执行表达式



示例：解释器模式 Java

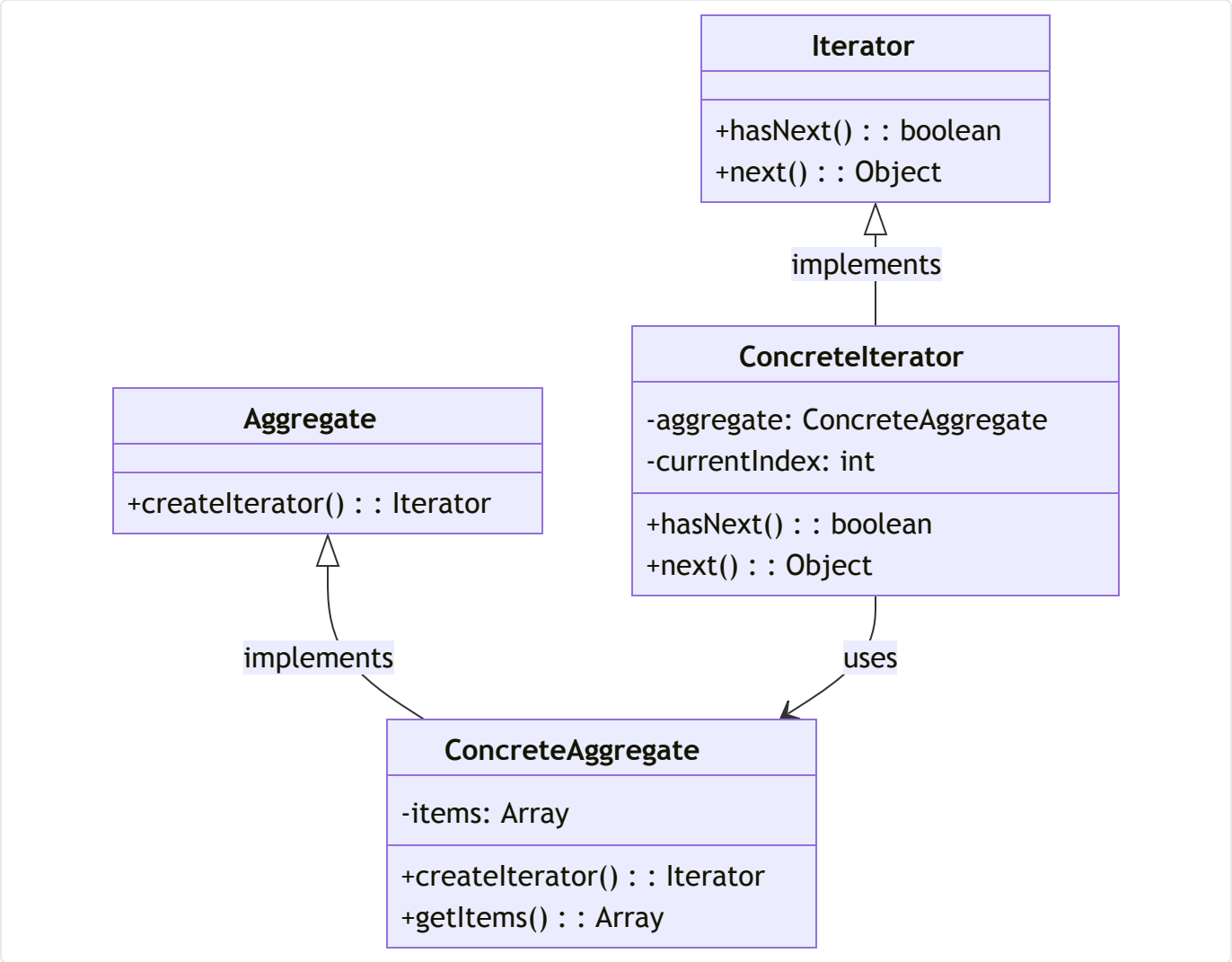
```java
// SQL 解析：WHERE age > 18 AND name = 'John'
Expression condition = new And(
    new GreaterThan(new Column("age"), new Value(18)),
    new Equals(new Column("name"), new Value("John"))
);
```

3.3.4. 迭代器模式

迭代器模式（Iterator Pattern）用于顺序访问一个聚合对象中的各个元素而不暴露其内部的表示，为遍历不同的数据结构提供了一种统一的接口，使得客户端代码可以独立于聚合对象的内部结构进行遍历操作，其核心思想是**提供一种方式来访问聚合对象中的元素同时隐藏内部的实现细节**

迭代器模式的适用场景包括：

- 遍历复杂数据结构
- 隐藏内部实现细节
- 解耦聚合对象与客户端



示例：迭代器模式 Java

```java
// 节点类
class Node {
    Object data;
    Node next;

    public Node(Object data) {
        this.data = data;
        this.next = null;
    }
}

// 迭代器接口
interface Iterator {
    boolean hasNext();
    Object next();
}

// 聚合接口
interface Aggregate {
    Iterator iterator();
}

// 具体聚合类（链表）
class LinkedListAggregate implements Aggregate {
    private Node head;

    public LinkedListAggregate() {
        this.head = null;
    }

    public void add(Object data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }

    @Override
    public Iterator iterator() {
```

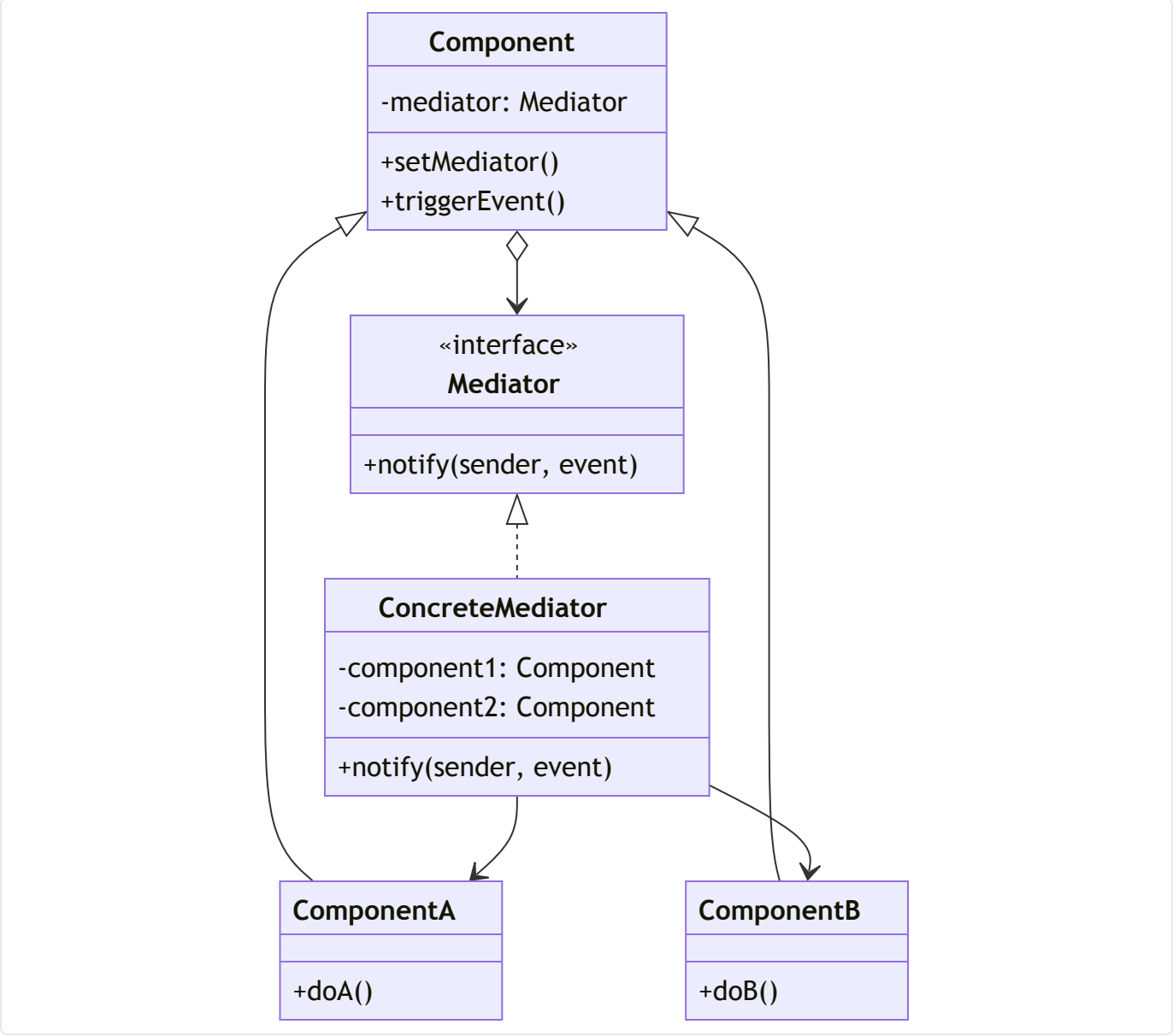
```
            return new LinkedListIterator(head);
        }
    }

    // 具体迭代器类
    class LinkedListIterator implements Iterator {
        private Node current;

        public LinkedListIterator(Node head) {
            this.current = head;
        }

        @Override
        public boolean hasNext() {
            return current != null;
        }

        @Override
        public Object next() {
            if (hasNext()) {
                Object data = current.data;
                current = current.next;
                return data;
            } else {
                throw new NoSuchElementException("No more elements");
            }
        }
    }

    // 客户端代码
    public class LinkedListIteratorExample {
        public static void main(String[] args) {
            LinkedListAggregate list = new LinkedListAggregate();
            list.add("Node 1");
            list.add("Node 2");
            list.add("Node 3");

            Iterator iterator = list.iterator();
            while (iterator.hasNext()) {
                System.out.println(iterator.next());
            }
        }
    }
```

3.3.5. 中介者模式

中介者模式（Mediator Pattern）通过封装一组对象间的交互来降低对象间的直接耦合，使得这些对象不必显式相互引用，其核心思想是**集中控制交互的对象使得各对象只需与中介者通信**，复杂交互逻辑统一由中介者管理

中介者模式的适用场景包括：

- 对象间存在复杂的网状引用关系（如 GUI 组件交互）
- 需要集中管理多个对象间的交互逻辑
- 希望复用组件但组件间依赖过多



示例：中介者模式 Java

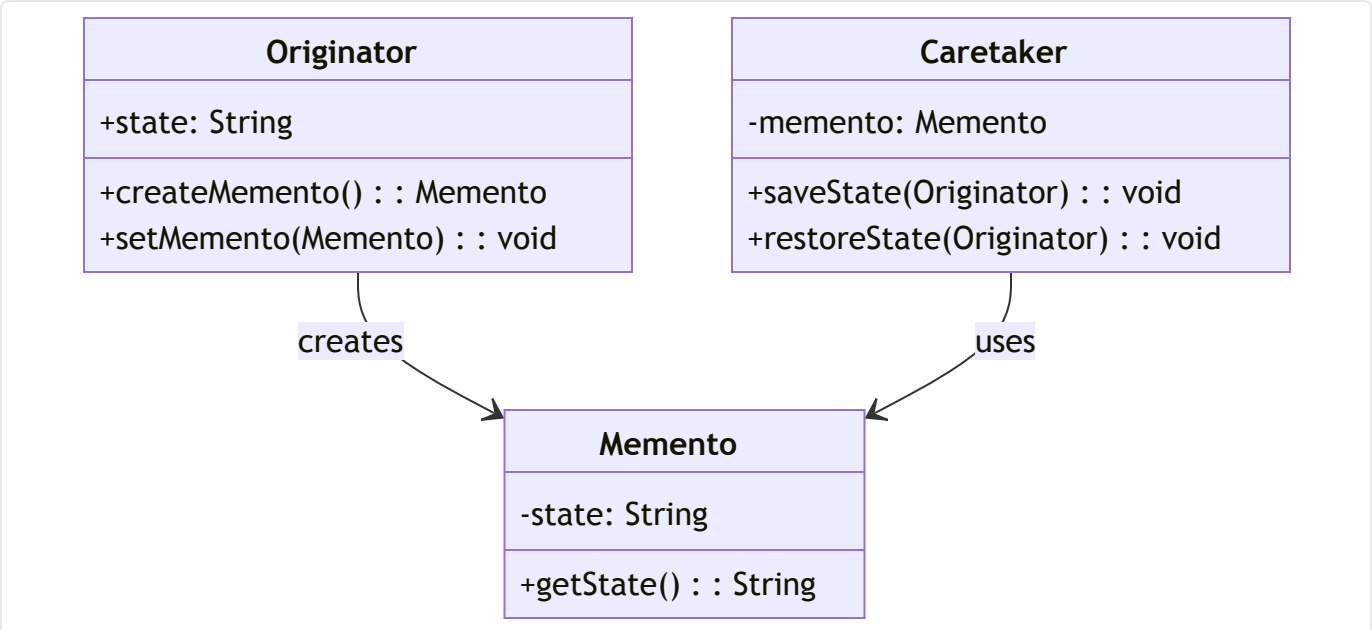
```java
// JMS中的Topic/Queue就是中介者
ConnectionFactory factory = new ActiveMQConnectionFactory();
Connection connection = factory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic topic = session.createTopic("news");
MessageProducer producer = session.createProducer(topic);
```

3.3.6. 备忘录模式

备忘录模式（Memento Pattern）用于在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态允许对象在状态变化后恢复到某个特定的“快照”状态，其核心思想是**将对象的状态封装在备忘录对象中，确保状态的保存和恢复操作不会影响对象的封装性**

备忘录模式的适用场景包括：

- 需要撤销操作
- 对象的状态可能会频繁变化且需要在某个时间点恢复到之前的状态时
- 需要对对象的状态进行备份



示例：备忘录模式 Java

```java
class MementoPatternExample {
    public static void main(String[] args) {
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        // 设置初始状态
        originator.setState("State 1");
        System.out.println("Initial state: " + originator.getState());

        // 保存状态
        caretaker.saveState(originator);

        // 修改状态
        originator.setState("State 2");
        System.out.println("Modified state: " + originator.getState());

        // 恢复状态
        caretaker.restoreState(originator);
        System.out.println("Restored state: " + originator.getState());
    }
}

// 发起人角色，负责创建备忘录对象并从备忘录对象中恢复状态
class Originator {
    private String state;

    public String getState() {
        return state;
    }

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

    // 创建备忘录对象，保存当前状态
    public Memento createMemento() {
        return new Memento(state);
    }

    // 从备忘录对象中恢复状态
    public void setMemento(Memento memento) {
        this.state = memento.getState();
    }
}

```

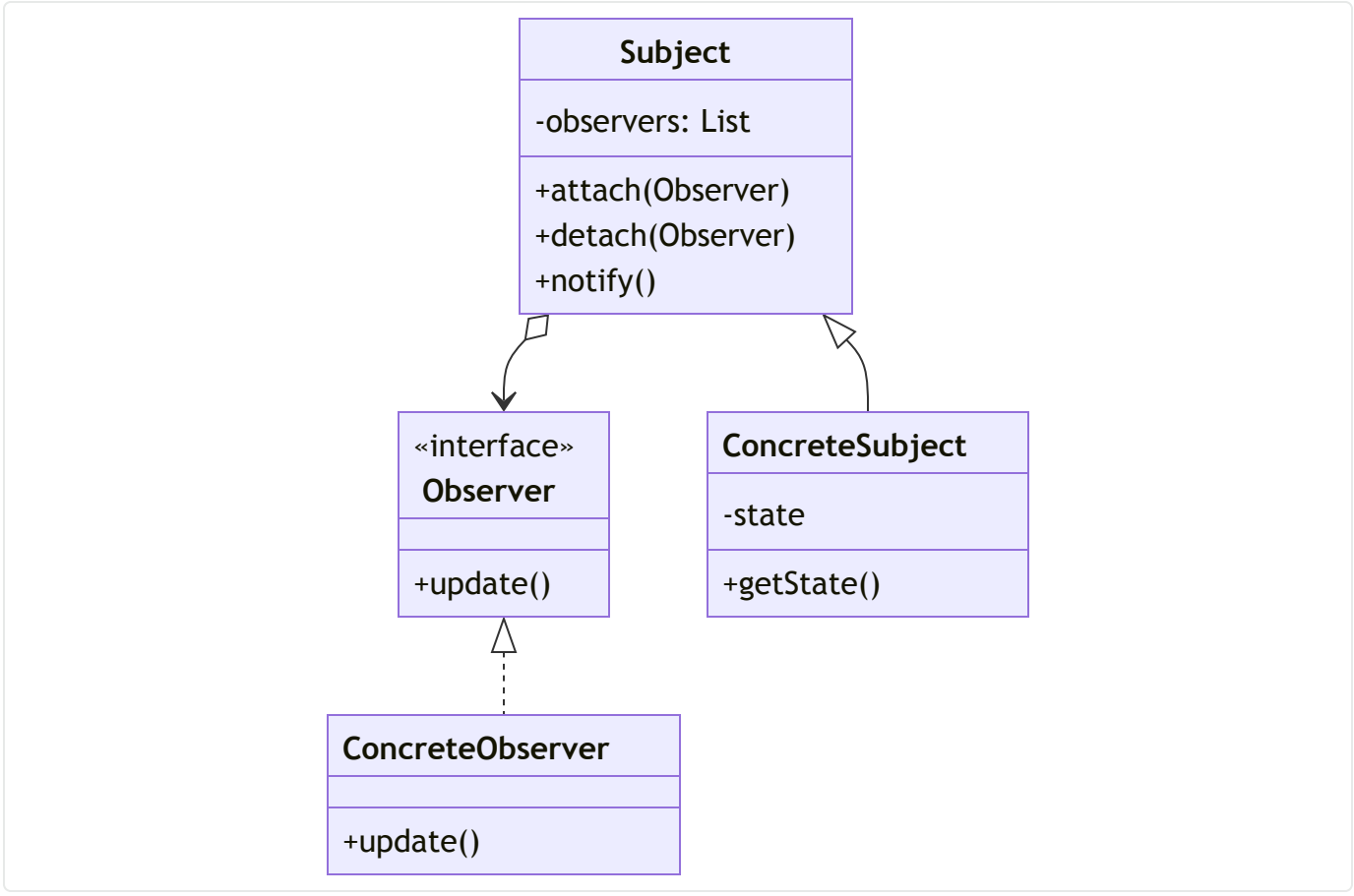
```java
// 备忘录角色，用于存储发起人的内部状态
class Memento {
    private final String state;

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

    public String getState() {
        return state;
    }
}

// 管理者角色，负责保存和恢复备忘录对象
class Caretaker {
    private Memento memento;

    // 保存状态
    public void saveState(Originator originator) {
        this.memento = originator.createMemento();
    }

    // 恢复状态
    public void restoreState(Originator originator) {
        originator.setMemento(memento);
    }
}
```

3.3.7. 观察者模式

观察者模式（Observer Pattern）定义对象间的一对多依赖关系，当一个对象状态改变时，所有依赖它的对象都自动收到通知并更新，其核心思想是**发布–订阅机制**，由发布者维护订阅者列表

观察者模式的适用场景包括：

- 事件驱动的交互（如 GUI 界面）
- 状态变化时需要广播
- 实时数据推送
- 多端同步场景
- 日志系统与监控反馈


Subject
-observers: List
+attach(Observer)
+detach(Observer)
+notify()
«interface»
Observer
+update()
ConcreteSubject
-state
+getState()
ConcreteObserver
+update()

示例：观察者模式 Java

```java
// 主题接口
interface Subject {
    void register(Observer o);
    void notifyObservers();
}

// 具体主题（气象站）
class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;

    public void setTemperature(float temp) {
        this.temperature = temp;
        notifyObservers();
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(temperature);
        }
    }
}

// 观察者接口
interface Observer {
    void update(float temperature);
}

// 具体观察者（手机显示）
class PhoneDisplay implements Observer {
    public void update(float temp) {
        System.out.println("手机显示温度: " + temp);
    }
}
```

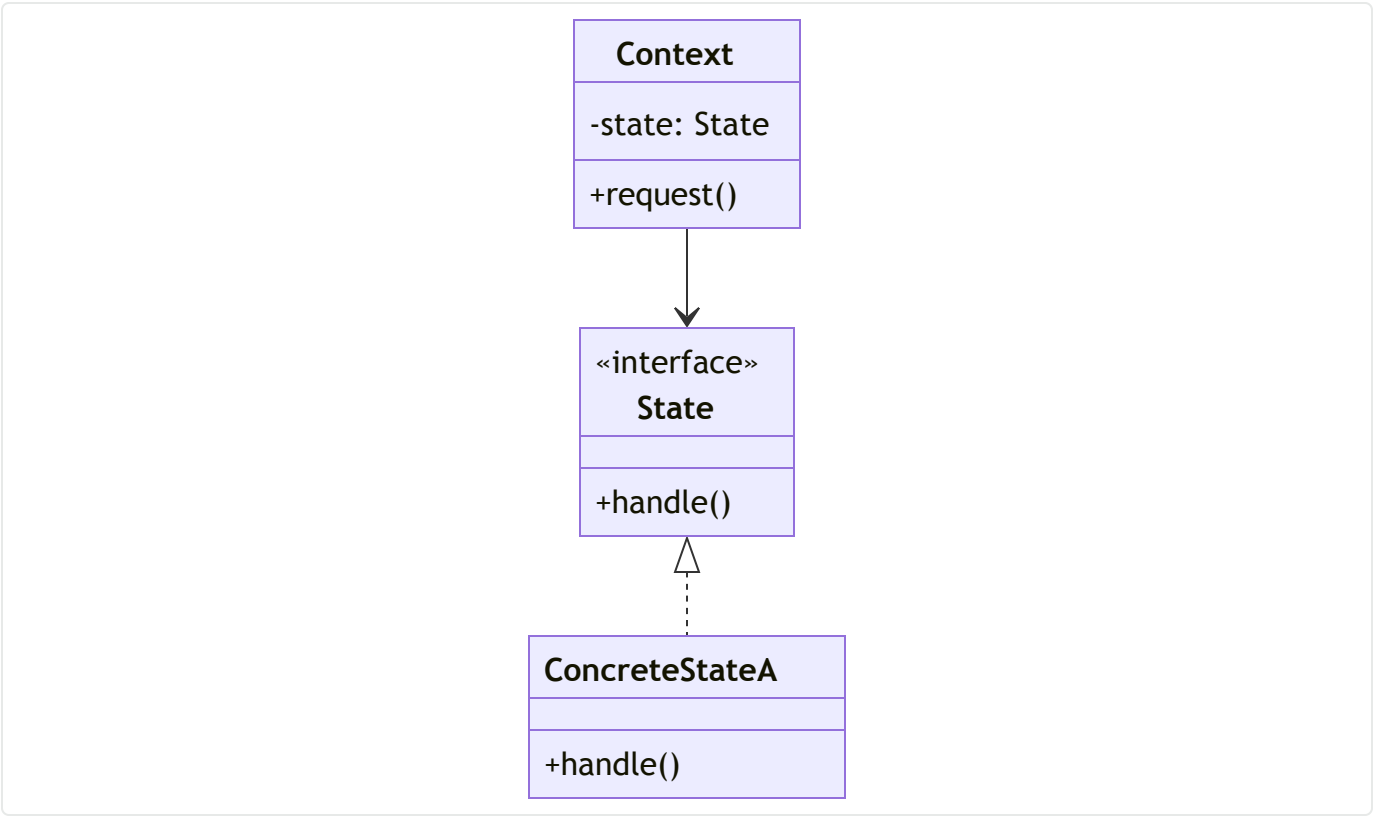
3.3.8. 状态模式

状态模式（State Pattern）允许对象在其内部状态改变行为时改变行为，其核心思想是**把每个状态的行为封装到独立的类中**，由上下文对象将请求委托给当前状态的对象

状态模式的适用场景包括：

- 对象行为依赖其内部状态（如游戏角色）

- 状态切换涉及复杂逻辑（如有限状态机）
- 状态行为需要动态切换（如设备状态管理）



示例：状态模式 Java

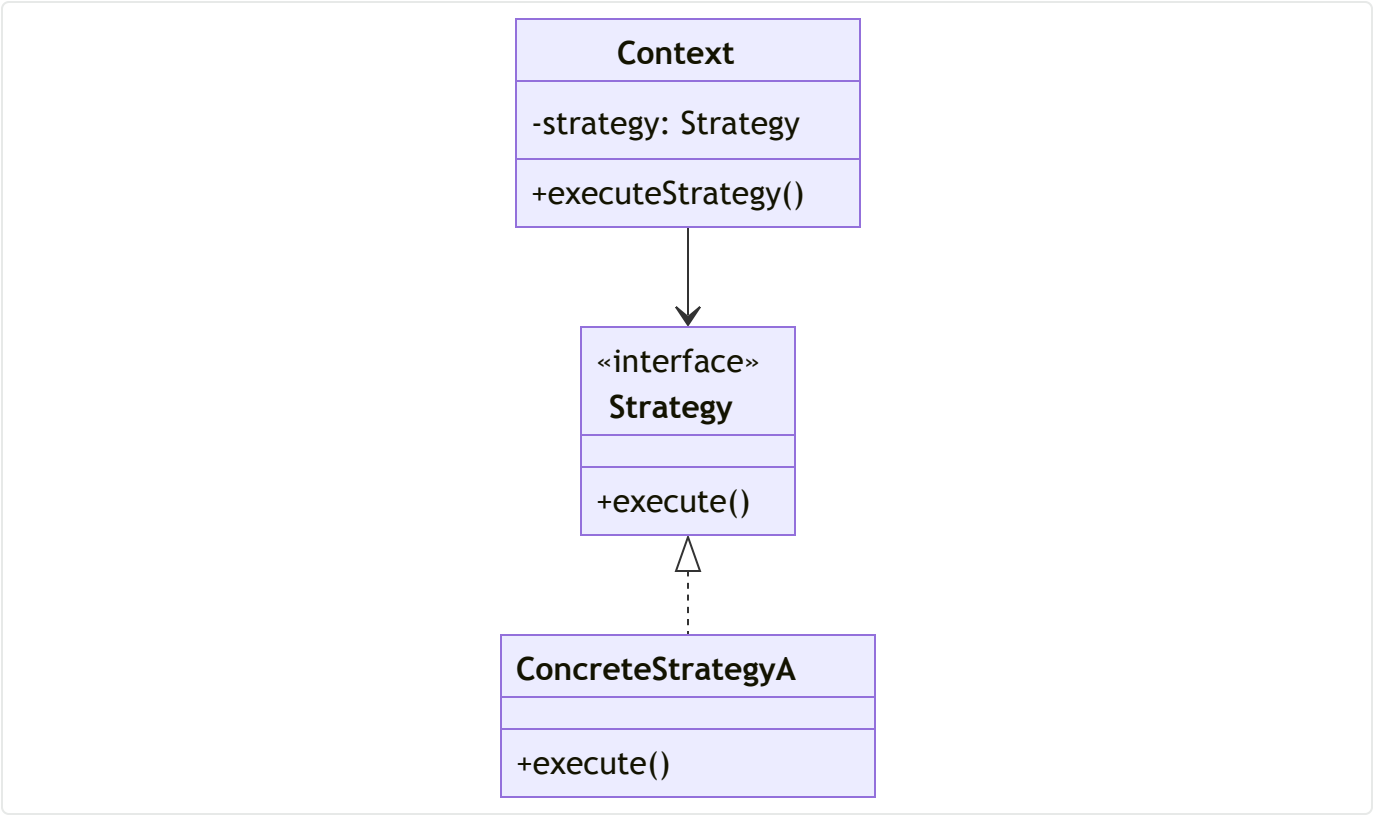
```java
// 状态接口
interface VendingMachineState {
    void insertCoin();
    void dispenseItem();
}

// 具体状态：待投币
class NoCoinState implements VendingMachineState {
    public void insertCoin() {
        System.out.println("硬币已投入");
    }
    public void dispenseItem() {
        System.out.println("请先投币");
    }
}

// 上下文（自动售货机）
class VendingMachine {
    private VendingMachineState currentState;

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

    public void insertCoin() {
        currentState.insertCoin();
    }
}
```

3.3.9. 策略模式

策略模式（Strategy Pattern）负责定义一系列算法，封装每个算法，使它们可以互相替换，其核心思想是**封装算法实现使之与使用解耦**，同时支持运行时的动态策略选择

策略模式的适用场景包括：

- 不同算法与行为的动态切换
- 多种策略与算法的组合使用
- 大量条件分支的替代



示例：策略模式 Java

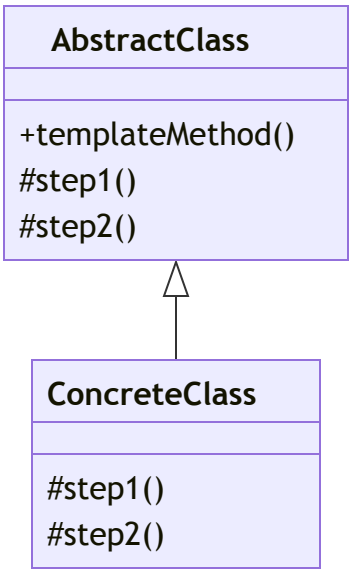
```java
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("信用卡支付: " + amount);
    }
}

class ShoppingCart {
    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void checkout(int amount) {
        strategy.pay(amount);
    }
}
```

3.3.10. 模板方法模式

模板方法模式（Template Method Pattern）在弗雷中定义算法骨架，将某些步骤延迟到子类实现，其核心思想是**由父类控制不可变流程而子类实现可变步骤**

模板方法模式的适用场景包括：

- 固定流程，可变步骤
- 代码复用与扩展
- 控制子类扩展



示例：模板方法模式 Java

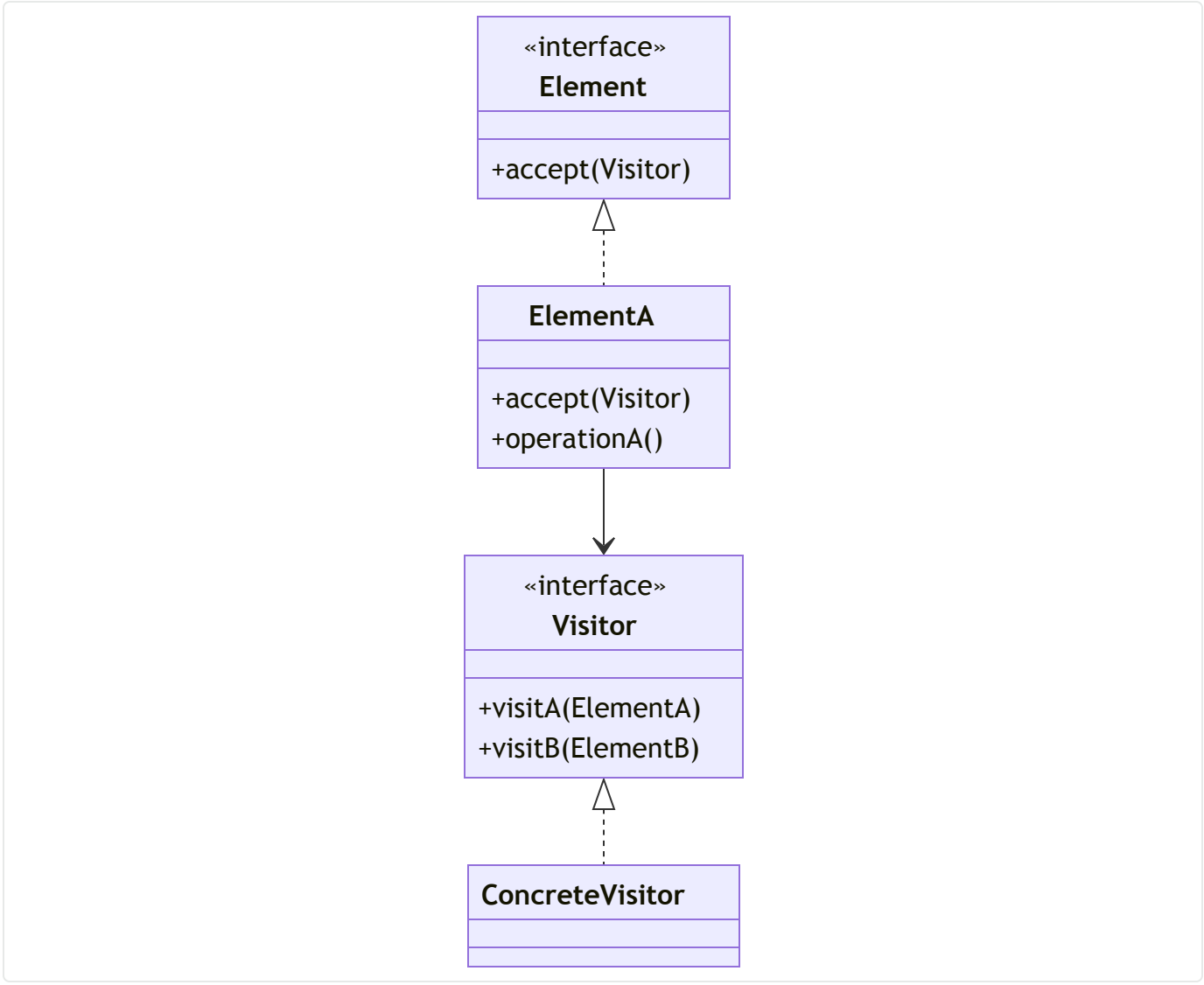
```java
abstract class Game {
    // 模板方法（final防止子类覆盖流程）
    final void play() {
        initialize();
        startPlay();
        endPlay();
    }

    abstract void initialize();
    abstract void startPlay();
}

class Cricket extends Game {
    void initialize() {
        System.out.println("初始化板球游戏");
    }
    void startPlay() {
        System.out.println("开始玩板球");
    }
}
```

3.3.11. 访问者模式

访问者模式（Visitor Pattern）负责将算法与对象结构分离，在不修改元素类的前提下定义新操作，其核心思想是**通过双重分发实现动态绑定**，将数据结构与操作逻辑分离

访问者模式的适用场景包括：

- 对象结构相对稳定但操作频繁变化
- 对象操作依赖其具体类型
- 操作需要跨多对象结构


«interface»
Element
+accept(Visitor)
ElementA
+accept(Visitor)
+operationA()
«interface»
Visitor
+visitA(ElementA)
+visitB(ElementB)
ConcreteVisitor

示例：访问者模式 Java

```java
interface ComputerPartVisitor {
    void visit(Keyboard keyboard);
    void visit(Monitor monitor);
}

class ComputerPartDisplayVisitor implements ComputerPartVisitor {
    public void visit(Keyboard keyboard) {
        System.out.println("显示键盘");
    }
    public void visit(Monitor monitor) {
        System.out.println("显示显示器");
    }
}

interface ComputerPart {
    void accept(ComputerPartVisitor visitor);
}

class Keyboard implements ComputerPart {
    public void accept(ComputerPartVisitor visitor) {
        visitor.visit(this);
    }
}
```

4. 项目设计模式

本项目是以文件上传与下载为主功能的共享资源平台，项目主要围绕文件的上传与下载展开，也支持用户系统、评论区等功能，因此可以使用**面向对象**的思想进行建模，辅以上述的设计模式进行设计开发

1. 系统的数据库连接池、配置管理采用**单例模式**，确保系统运行时只有一个单一的配置实例和连接对象，减少重复冗余的开销
2. 系统主要围绕文件操作展开，文件类型有多种多样，对于 PDF 类型文件可以直接进行 Web 端展示，对于纯文本类型也可以直接展示，而对于其它类型的文件则需要做其它处理，这里可以采用**工厂方法模式**来实例化每种资源类型的文件
3. 系统的 UI 设计采用**抽象工厂模式**，制定一系列跨平台 UI 组件
4. 数据库查询的 SQL 语句可以采用**建造者模式**构建，逐步加入参数并适配业务需求

5. 用户系统支持用户订阅相关资源，不同资源嵌套订阅在用户对象下，采用**组合模式**
6. 将数据层的数据持久化操作统一封装，其操作全部封装在单独的类中，向服务层提供访问接口以实现解耦，采用**外观模式**
7. 采用连接池、文件对象缓冲池等策略实现**享元模式**，减少服务器运行时开销
8. MVC 架构的视图层和服务层之间由控制层负责处理二者转换，体现**中介者模式**
9. 资源文件保存其每个历史状态，便于查看历史版本和修改记录，体现**备忘录模式**
10. 用户订阅相关课程资源发生更新时可以即使通知订阅者，体现**观察者模式**
11. 服务器支持后台管理人员进入，不同身份权限的用户登录可以有不同的服务，体现**策略模式**