

1.2 Observer Pattern

观察者模式

UML Relations Continued

继续上节课的UML关系讲解

UML Relationship

UML中的关系

~~[Animal] <|-- [Dog] // 继承~~

~~[Flyable] <|.. [Bird] // 接口实现~~

~~[Car] *-- [Engine] // 组合~~

[Car] o-- [Passenger] // 聚合

[Teacher] --> [Student] // 关联（成员变量）

[OrderService] ..> [Logger] // 依赖（临时使用）

Aggregation

聚合

聚合是一种“has-a”（有一个）的关系，用于表示一个对象（整体）拥有另一个或多个对象（部分），但这些部分对象在概念上可以独立存在。

- 聚合关系中的部分对象不依赖于整体对象的生命周期。
- 整体对象拥有对部分对象的引用，但并不负责创建或销毁这些部分对象。
- 聚合关系往往体现一种“弱拥有”的关系。

Aggregation

聚合

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

public class Main {
    public static void main(String[] args) {
        // 创建一些学生对象（这些对象可以在多个地方独立使用）
        Student student1 = new Student("Alice");
        Student student2 = new Student("Bob");
        Student student3 = new Student("Charlie");

        // 将学生对象加入一个列表，由外部传入 Classroom
        List<Student> studentList = new ArrayList<>();
        studentList.add(student1);
        studentList.add(student2);
        studentList.add(student3);

        // 创建班级对象（整体对象）并传入学生列表
        Classroom class101 = new Classroom(studentList);
        class101.printStudents(); // 输出班级中的学生名单
    }
}
```

```
// 学生类，表示聚合关系中的部分
class Student {
    private String name;

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

    public void study() {
        System.out.println(name + " 正在学习。");
    }

    public String getName() {
        return name;
    }
}

// 班级类，表示整体，聚合多个学生
class Classroom {
    // 采用集合来保存聚合的学生对象
    private List<Student> students;

    // 构造方法由外部传入学生列表
    public Classroom(List<Student> students) {
        this.students = students;
    }

    // 方法，打印班级中所有学生的名字
    public void printStudents() {
        System.out.println("班级里的学生有：");
        for (Student s : students) {
            System.out.println(s.getName());
        }
    }
}
```

Aggregation & Composition

聚合 与 组合

聚合通常通过成员变量实现

这和组合好像差不多？

组合也是通过Delegation（委托）行为给另一个类来执行，从而达到解耦合的目的？

究竟区别在哪？

与组合不同，聚合的成员变量通常由外部创建并传入整体对象，而不是在整体对象内部直接创建。

Aggregation & Composition

聚合 与 组合

聚合 (Aggregation)

整体可以拥有“部分”，但“部分”可以独立存在

“部分”对象与整体生命周期不绑定

“部分”对象由外部创建并且传入整体

组合通过构造或者setter传入已有对象

组合 (Composition)

整体强拥有“部分”，“部分”依附于整体存在，生命周期绑定

“部分”对象由整体内部创建、管理和销毁

Aggregation Example

聚合示例

Engine 是外部创建好的（可以被多个类复用）；

AggregatedCar 只是引用它；


如果 Car 销毁了，Engine 依然存在。

// 引擎类

```
class Engine {  
    public void start() {  
        System.out.println("引擎启动！");  
    }  
}
```

// 聚合关系：引擎由外部传入

```
class AggregatedCar {  
    private Engine engine; // 引擎是外部传入的  
  
    public AggregatedCar(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void run() {  
        System.out.println("汽车启动中...");  
        engine.start(); // 委托给引擎  
    }  
}
```



Composition Example

组合示例

Engine 由 Car 自己构造；

外部无法干涉引擎的存在；

如果 Car 销毁，engine 也一起消失。

// 引擎类

```
class Engine {  
    public void start() {  
        System.out.println("引擎启动！");  
    }  
}
```

// 组合关系：引擎由内部创建

```
class ComposedCar {  
    private Engine engine = new Engine(); //  
    Car 自己 new 出来
```

```
    public void run() {  
        System.out.println("组合式汽车启动  
中...");  
        engine.start(); // 委托给引擎  
    }  
}
```

Association & Dependency

关联 与 依赖

关联 Association

关联是类与类之间**最基础的关系**，表示一个类知道另一个类、并可以使用它。

- 通常通过**成员变量**表现
- 比如：老师有一个学生对象，学生也可能有一个老师引用（双向关联）
- 生命周期**不绑定**，只是“知道它”

Association Example

关联示例

Driver 是成员变量，“长期绑定”
的一对一、多对一等关系

适合表示“车有一个驾驶员”这
样的业务场景

不对，那这个成员变量哪里来的
呢？

```
class Driver {
    String name;
    public Driver(String name) {
        this.name = name;
    }

    void drive() {
        System.out.println(name + " 正在开车");
    }
}

class Car {
    private Driver driver; // 成员变量, Car 长期知道 Driver

    public void setDriver(Driver driver) {
        this.driver = driver;
    }

    public void run() {
        if (driver != null) driver.drive();
        System.out.println("汽车启动! ");
    }
}
```

Source of Associated Object

关联对象的来源

1. 构造函数传入

在创建对象时由外部注入

```
new Teacher(new Student("Tom"));
```

2. setter 注入

先创建对象，再设定

```
teacher.setStudent(student);
```

3. 关联注册

被动注册给主类

```
student.registerToTeacher(this);
```

4. 集合添加

将多个对象加入集合

```
teacher.addStudent(student);
```

5. IoC 容器注入

如 Spring 中由框架自动注入

```
@Autowired private Student s;
```

```
class Student {
    private String name;
    public Student(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}

class Teacher {
    private Student student; // 关联关系：知道 student，但不创建

    // 来源方式一：构造函数注入
    public Teacher(Student student) {
        this.student = student;
    }

    // 来源方式二：setter 注入
    public void setStudent(Student student) {
        this.student = student;
    }

    public void teach() {
        System.out.println("教导学生： " + student.getName());
    }
}
```

Association & Dependency

关联 与 依赖

依赖 Dependency

依赖是类 A 临时使用类 B 的某种功能，通常表现在方法中用作参数、局部变量或返回值。

- 生命周期完全不绑定，只是一种“用一下”的关系
- 属于最低耦合的关系

Dependency Example

依赖 示例

Logger 并不是 Car 的成员，只是临时在 run() 方法中使用一下

Car 和 Logger 并没有“关系”，只是“借用”一下

```
class Logger {  
    void log(String msg) {  
        System.out.println("[LOG] " + msg);  
    }  
}  
  
class Car {  
    public void run(Logger logger) {  
        logger.log("汽车即将启动");  
        System.out.println("汽车启动! ");  
    }  
}
```

UML Relationship

UML中的关系

~~[Animal] <|-- [Dog] // 继承~~

~~[Flyable] <|.. [Bird] // 接口实现~~

~~[Car] *-- [Engine] // 组合~~

[Car] o-- [Passenger] // 聚合

[Teacher] --> [Student] // 关联（成员变量）

[OrderService] ..> [Logger] // 依赖（临时使用）

UML Relationship Core Difference

UML中的关系关键差异

[Animal] <|-- [Dog] // 继承

[Flyable] <|.. [Bird] // 接口实现

[Car] *-- [Engine] // 组合

[Car] o-- [Passenger] // 聚合

[Teacher] --> [Student] // 关联（成员变量）

[OrderService] ..> [Logger] // 依赖（临时使用）

UML Relationship Core Difference

UML中的关系关键差异

关系类型	关键点	拥有成员变量	自建创建成员变量	生命周期绑定	使用时机
继承	extends	X	X	X	结构继承
接口实现	interface & implement	X	X	X	定义-实现-策略
组合	X	是	是-内部创建	是	强耦合，完全控制
聚合	X	是	否-外部传入	否	弱耦合，协作关系
关联	X	是	否	否	解耦合，长期使用
依赖	X	否	否	否	解耦合，临时调用

Quiz

辨别其所属关系

```
class Engine {  
    public void start() {  
        System.out.println("引擎启动! ");  
    }  
}  
  
class Car {  
    private Engine engine = new Engine();  
    public void run() {  
        engine.start();  
        System.out.println("汽车行驶中...");  
    }  
}
```

Quiz

辨别其所属关系 - 组合

```
class Engine {  
    public void start() {  
        System.out.println("引擎启动! ");  
    }  
}  
  
class Car {  
    private Engine engine = new Engine();  
    public void run() {  
        engine.start();  
        System.out.println("汽车行驶中...");  
    }  
}
```

Quiz

辨别其所属关系

```
class Employee {
    private String name;
    public Employee(String name) {
        this.name = name;
    }
}

class Company {
    private List<Employee> employees;

    public Company(List<Employee> employees) {
        this.employees = employees; // 外部传入
    }

    public void printEmployeeNames() {
        for (Employee e : employees) {
            System.out.println(e.name);
        }
    }
}
```

Quiz

辨别其所属关系-聚合

```
class Employee {
    private String name;
    public Employee(String name) {
        this.name = name;
    }
}

class Company {
    private List<Employee> employees;

    public Company(List<Employee> employees) {
        this.employees = employees; // 外部传入
    }

    public void printEmployeeNames() {
        for (Employee e : employees) {
            System.out.println(e.name);
        }
    }
}
```

Quiz

辨别其所属关系

```
class Student {
    String name;
    public Student(String name) {
        this.name = name;
    }
}

class Teacher {
    private Student student; // 拥有一个Student引用

    public void setStudent(Student s) {
        this.student = s;
    }

    public void teach() {
        System.out.println("老师正在教 " + student.name);
    }
}
```

Quiz

辨别其所属关系 - 关联

```
class Student {
    String name;
    public Student(String name) {
        this.name = name;
    }
}

class Teacher {
    private Student student; // 拥有一个Student引用

    public void setStudent(Student s) {
        this.student = s;
    }

    public void teach() {
        System.out.println("老师正在教 " + student.name);
    }
}
```

Quiz

辨别其所属关系

```
class Battery {  
    void charge() {  
        System.out.println("电池正在充电");  
    }  
}  
  
class Laptop {  
    private Battery battery = new Battery();  
  
    public void powerOn() {  
        battery.charge();  
        System.out.println("笔记本电脑开机");  
    }  
}
```


Quiz

辨别其所属关系 - 组合

```
class Battery {  
    void charge() {  
        System.out.println("电池正在充电");  
    }  
}  
  
class Laptop {  
    private Battery battery = new Battery();  
  
    public void powerOn() {  
        battery.charge();  
        System.out.println("笔记本电脑开机");  
    }  
}
```

Quiz

辨别其所属关系

```
class EmailValidator {
    public boolean isValid(String email) {
        return email != null && email.contains("@");
    }
}

class UserService {
    public void createUser(String email) {
        EmailValidator validator = new EmailValidator(); // 临时使用
        if (validator.isValid(email)) {
            System.out.println("创建用户成功: " + email);
        } else {
            System.out.println("邮箱地址不合法");
        }
    }
}
```

Quiz

辨别其所属关系 - 依赖, **UserService** 只是在方法内部临时创建

```
class EmailValidator {
    public boolean isValid(String email) {
        return email != null && email.contains("@");
    }
}

class UserService {
    public void createUser(String email) {
        EmailValidator validator = new EmailValidator(); // 临时使用
        if (validator.isValid(email)) {
            System.out.println("创建用户成功: " + email);
        } else {
            System.out.println("邮箱地址不合法");
        }
    }
}
```

Quiz

辨别其所属关系

```
class Animal {  
    public void eat() {  
        System.out.println("动物吃东西");  
    }  
}
```

```
class Dog extends Animal {  
    public void bark() {  
        System.out.println("狗汪汪叫");  
    }  
}
```

Quiz

辨别其所属关系 - 继承

```
class Animal {  
    public void eat() {  
        System.out.println("动物吃东西");  
    }  
}  
  
class Dog extends Animal {  
    public void bark() {  
        System.out.println("狗汪汪叫");  
    }  
}
```

Observer Pattern

观察者模式

Observer Pattern

观察者模式

观察者模式（Observer Pattern）

核心思想：当一个对象（**Subject**，被观察者）的状态或行为发生变化时，**自动通知**一组依赖它的对象（**Observer**，观察者）来更新自身，或者执行相应的行为。

观察者模式



发布-订阅者模式



Observer Pattern

观察者模式

生活中的例子

- **微信公众号**：你关注了一个公众号（观察者），当公众号（被观察者）发布新文章，所有订阅者都会收到推送。
- **消息系统**：你在某个聊天群里，当有人发消息时，群里所有成员（观察者）都能收到提醒。
- **事件监听**：GUI 编程中，按钮（被观察者）被点击后，会通知所有注册的监听器（观察者）。

Observer Pattern - Core Structure

观察者模式 - 核心结构

观察者模式通常包含以下几个角色：

1. Subject (主题/被观察者)

- 维护一个观察者列表（如：List<Observer>）。
- 提供方法：增加/移除观察者（addObserver() / removeObserver()）。
- 当自身状态改变时，负责调用 notifyObservers() 通知所有观察者。

2. Observer (观察者)

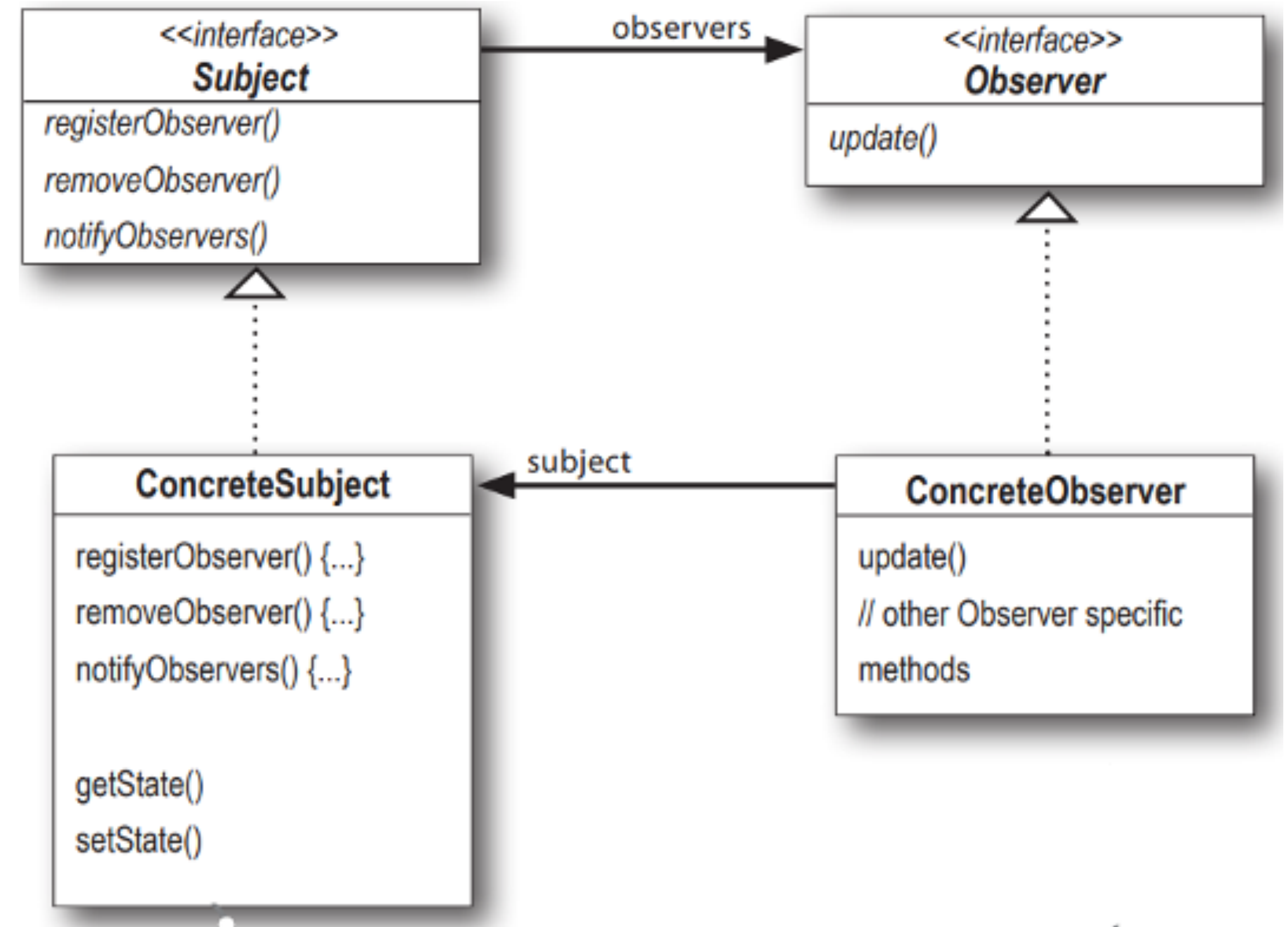
- 定义一个 update(...) 方法，表示当被观察者发生变化时，观察者如何做出反应或更新。

3. ConcreteSubject (具体被观察者)

- 实现 Subject 的接口/抽象类，内部包含自己的业务逻辑或状态。
- 在状态改变处调用 notifyObservers()。

4. ConcreteObserver (具体观察者)

- 实现 Observer 接口，在 update(...) 方法中编写如何响应被观察者的改变。



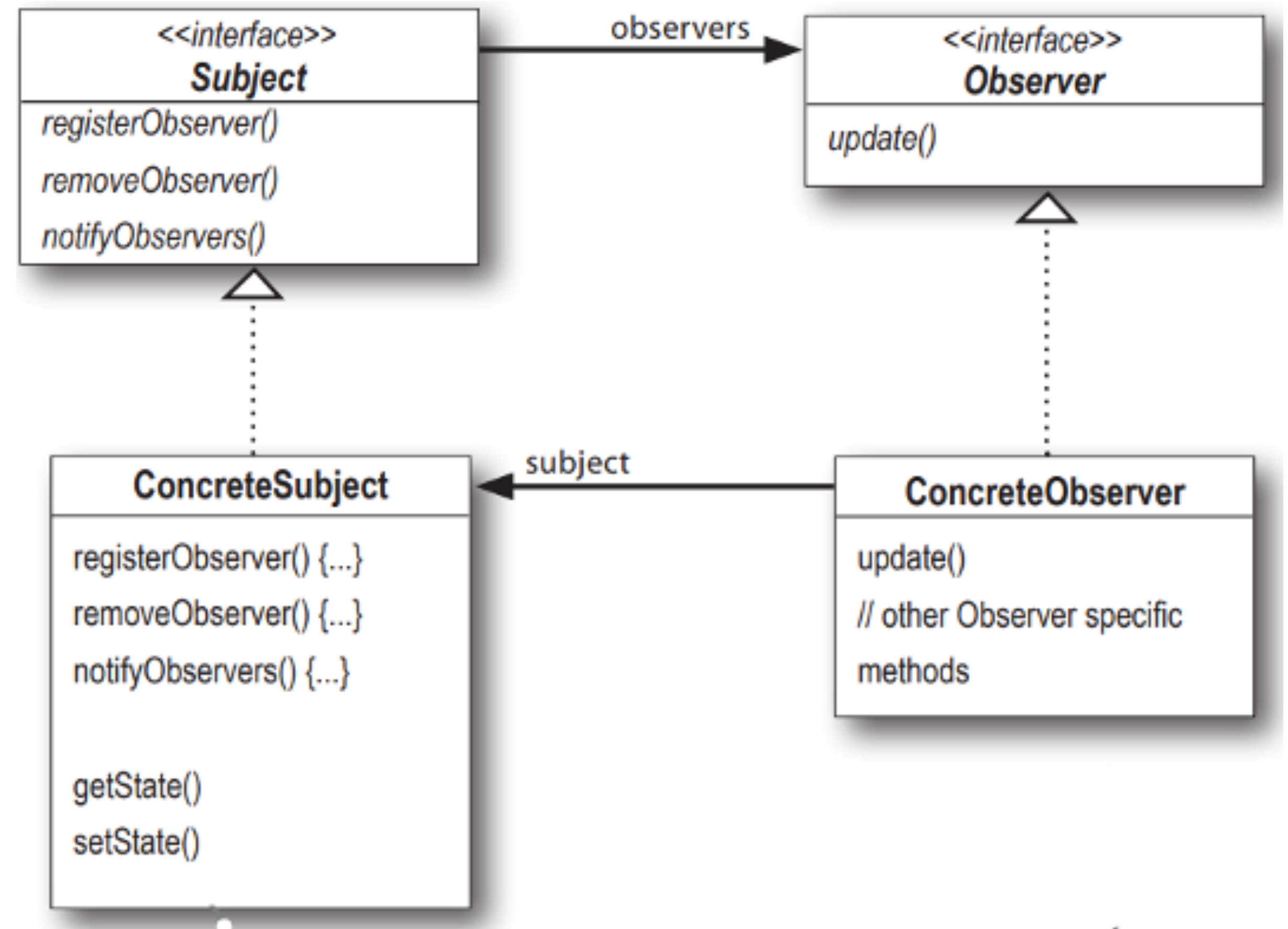
Observer Pattern - Core Structure

观察者模式 - 核心结构

Subject (主题/被观察者)

这是被观察者（也叫主题、发布者）的接口，定义了三大核心方法：

- registerObserver(): 添加观察者
- removeObserver(): 移除观察者
- notifyObservers(): 通知所有已注册的观察者



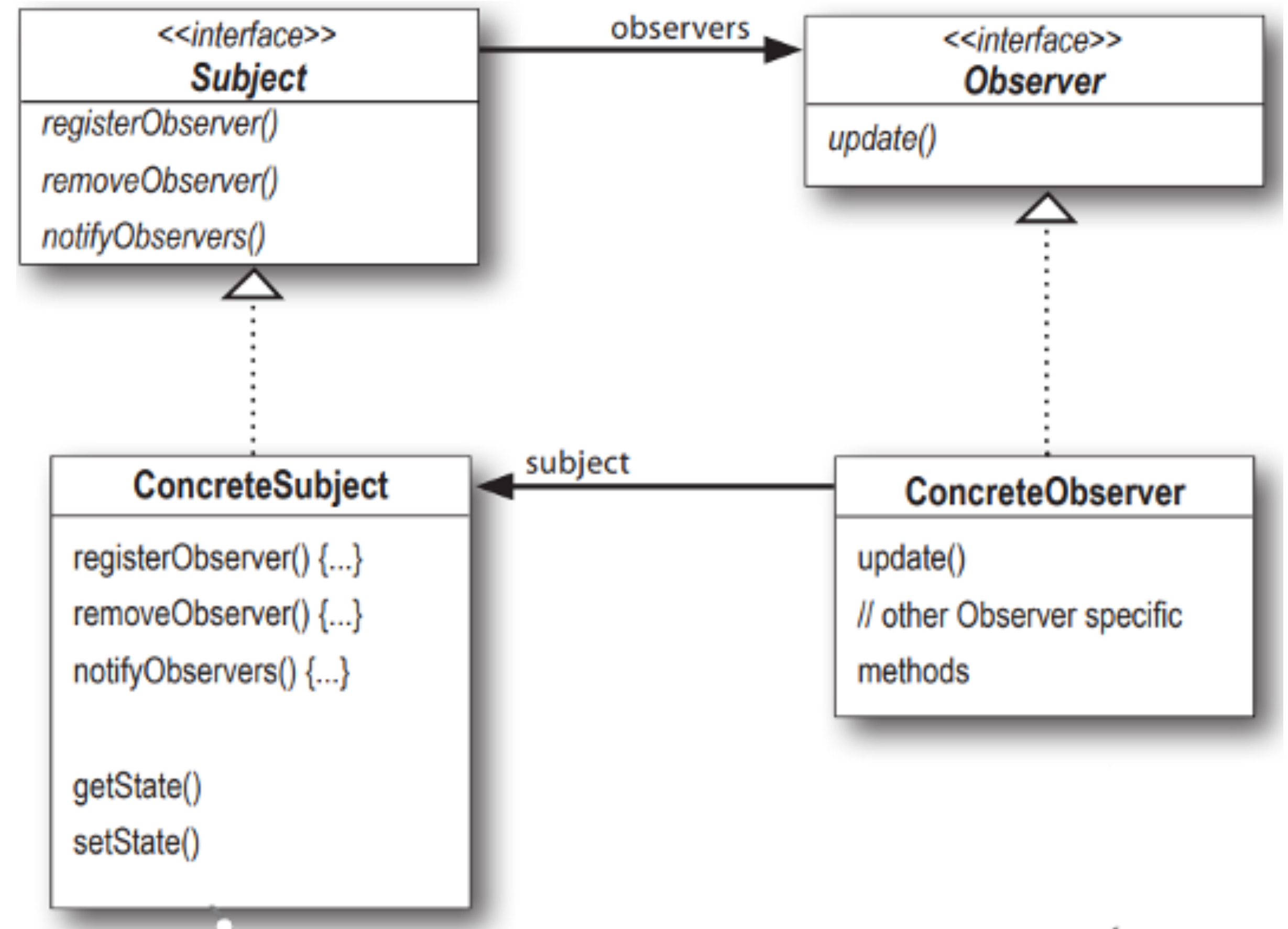
Observer Pattern - Core Structure

观察者模式 - 核心结构

Observer

这是观察者接口

update(): 当主题通知时被调用
(带参数或不带参数, 用于接收状态)



Observer Pattern - Core Structure

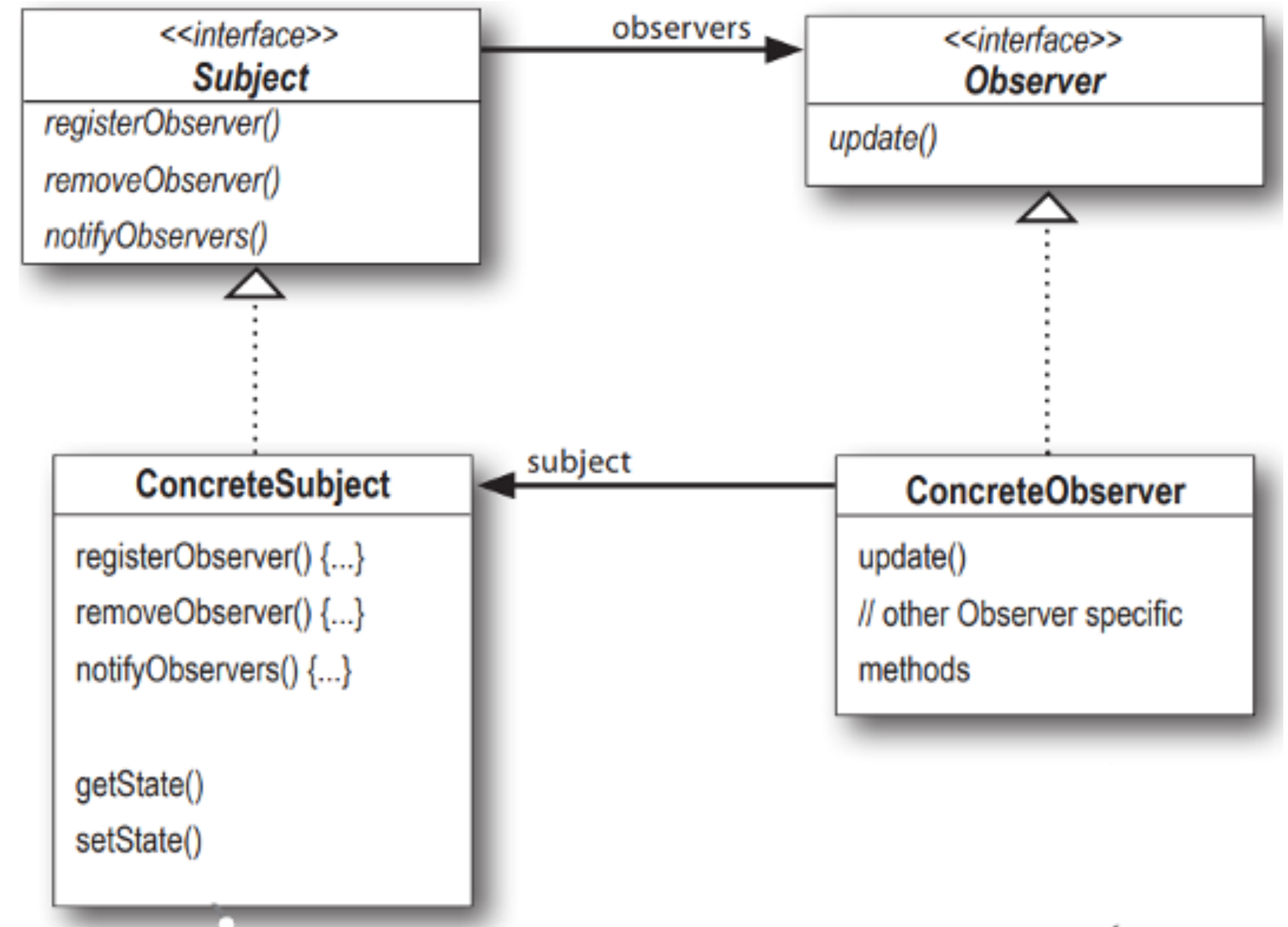
观察者模式 - 核心结构

ConcreteSubject

这是具体的主题（被观察者）

这是 Subject 接口的实现类：

- 它实现了对观察者的增删和通知逻辑；
- 额外定义了状态相关方法：
 - `getState()`：获取当前状态；
 - `setState()`：设置新状态，并调用 `notifyObservers()` 通知变化。



Observer Pattern - Core Structure

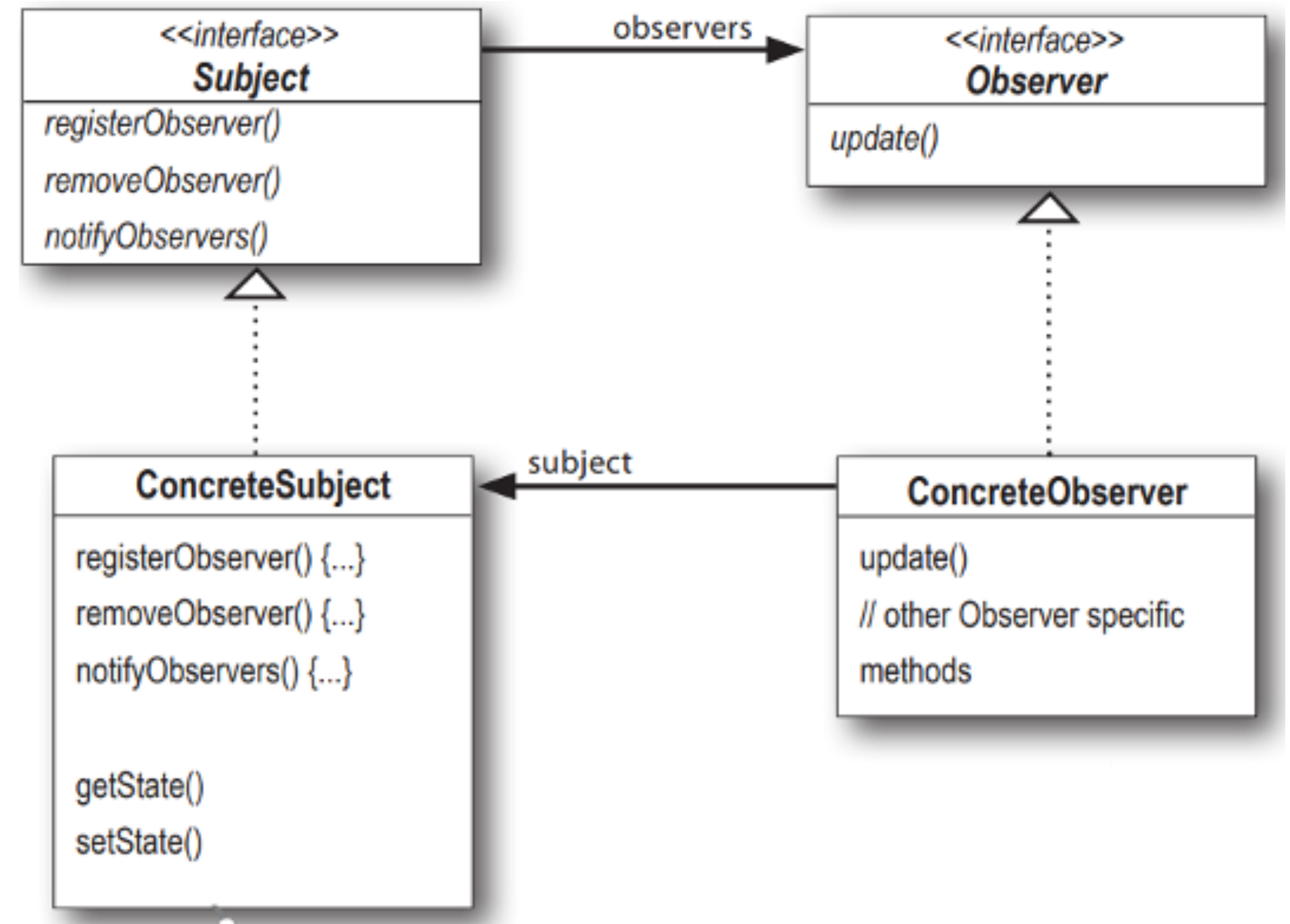
观察者模式 - 核心结构

ConcreteObserver

这是具体的观察者

实现了 Observer 接口，并提供：

- update() 方法：接收 ConcreteSubject 通知，并执行自定义逻辑；
- 可以包含其他观察者自己的逻辑，比如记录日志、更新界面等。

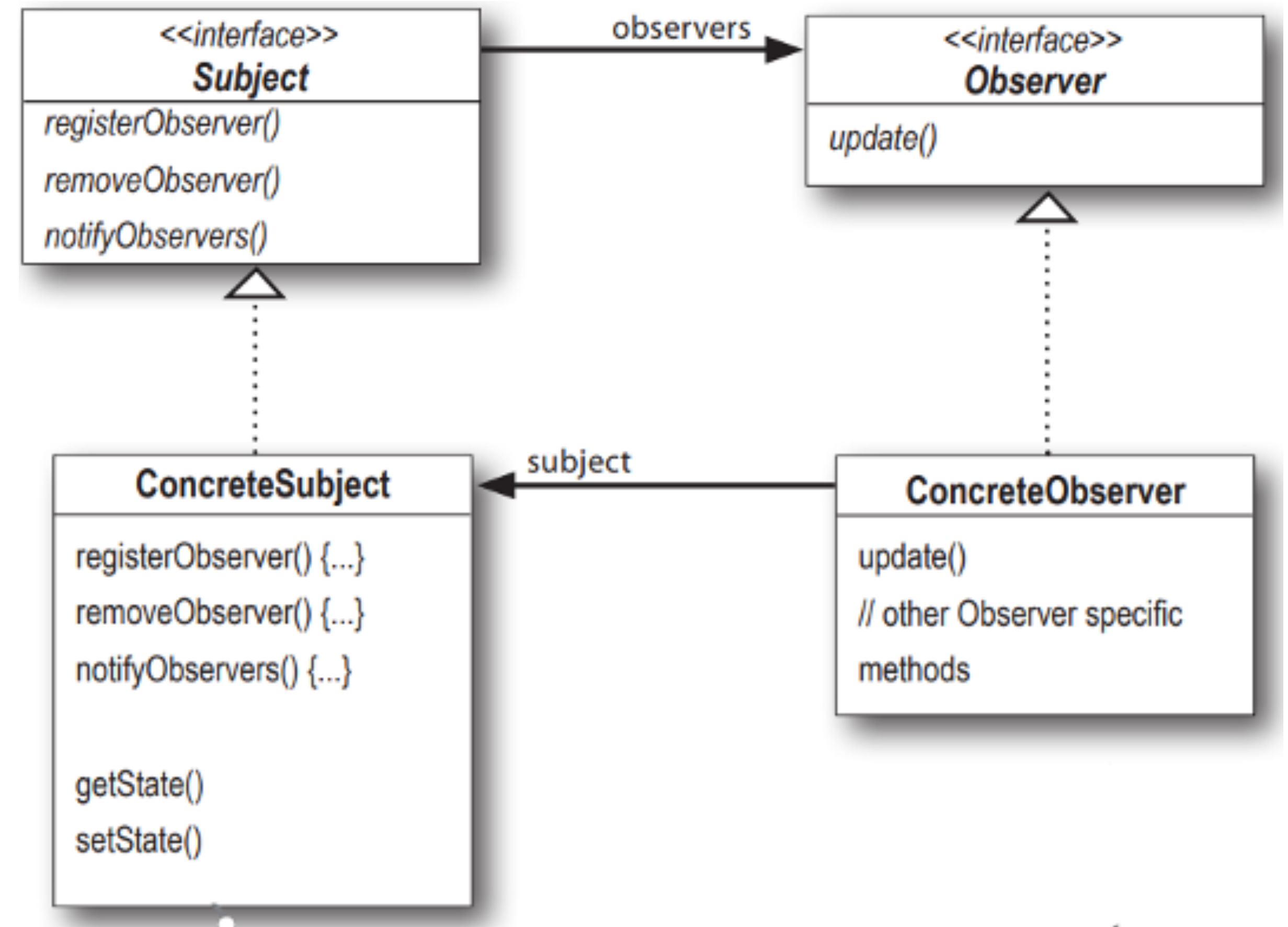


Observer Pattern - Core Structure

观察者模式 - 核心结构

根据模式创建一个程序：

1. 创建一个 ConcreteSubject (比如天气站)。
2. 创建多个 ConcreteObserver (比如手机 App)。
3. 通过 registerObserver() 把观察者注册到主题。
4. 当主题状态变化时：
 - 使用 setState() 更新内部状态；
 - 调用 notifyObservers() 通知所有观察者；
5. 所有观察者的 update() 方法被调用，并可通过 getState() 获取最新状态。



Observer Pattern - Example

观察者模式 - 示例

```
// observer接口
public interface Observer {
    // 当被观察者有新数据或状态变化时，调用此方法通知观察者
    void update(String weatherInfo);
}

// subject接口
public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// 观察者整合实现
public class WeatherApp implements Observer {
    private String appName;

    public WeatherApp(String appName) {
        this.appName = appName;
    }
    @Override
    public void update(String weatherInfo) {
        System.out.println(appName + " 收到天气更新: " + weatherInfo);
    }
}
```

```
// 天气观察实现
import java.util.ArrayList;
import java.util.List;

public class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String weatherInfo;

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

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

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

    // 当天气信息更新时，自动通知所有观察者
    public void setWeatherInfo(String info) {
        this.weatherInfo = info;
        notifyObservers();
    }
}
```

Observer Pattern - Implementation

观察者模式 - 实现

模拟一个“天气发布系统”：

- WeatherStation: 天气站，是被观察者 (**Subject**)
- WeatherApp: 天气App，是观察者 (**Observer**)
- 当天气变化时，WeatherStation 会自动通知所有已注册的WeatherApp。

Observer Pattern - Implementation

观察者模式 - 实现

1. 定义 Observer 接口

```
public interface Observer {  
    void update(String weatherInfo); // 接收通知  
}
```

此处规定观察者只需实现一个 `update()` 方法，用于被通知

Observer Pattern - Implementation

观察者模式 - 实现

2. 定义 Subject 接口

```
public interface Subject {  
    void registerObserver(Observer observer); // 注册  
    void removeObserver(Observer observer); // 移除  
    void notifyObservers(); // 通知所有  
}
```

此处被观察者维护一个观察者列表，并负责通知它们

Observer Pattern - Implementation

观察者模式 - 实现

3. 实现ConcreteSubject: WeatherStation

此处使用的就是接口实现

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

public class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String weatherInfo;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

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

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

    public void setWeather(String newWeather) {
        this.weatherInfo = newWeather;
        System.out.println("天气更新为: " + newWeather);
        notifyObservers(); // 自动通知
    }
}
```

Observer Pattern - Implementation

观察者模式 - 实现

4. 实现ConcreteObserver:

WeatherApp

此处使用也是接口实现

```
public class WeatherApp implements Observer {  
    private String appName;  
  
    public WeatherApp(String name) {  
        this.appName = name;  
    }  
  
    @Override  
    public void update(String weatherInfo) {  
        System.out.println(appName + " 收到天  
气推送: " + weatherInfo);  
    }  
}
```

Observer Pattern - Implementation

观察者模式 - 实现

5. 主类Main

整体调用

被观察者维护一组观察者，每当状态改变，就循环调用每个观察者的 `update()` 方法，实现自动通知机制

```
public class Main {  
    public static void main(String[] args) {  
        WeatherStation station = new WeatherStation();  
  
        WeatherApp app1 = new WeatherApp("天气通");  
        WeatherApp app2 = new WeatherApp("墨迹天气");  
        WeatherApp app3 = new WeatherApp("小米天气");  
  
        // 注册观察者  
        station.registerObserver(app1);  
        station.registerObserver(app2);  
        station.registerObserver(app3);  
  
        // 发布天气  
        station.setWeather("晴天 🌞");  
        station.setWeather("暴雨 ⚡");  
  
        // 移除一个观察者  
        station.removeObserver(app2);  
        station.setWeather("阴天 ☁");  
    }  
}
```

Observer Pattern

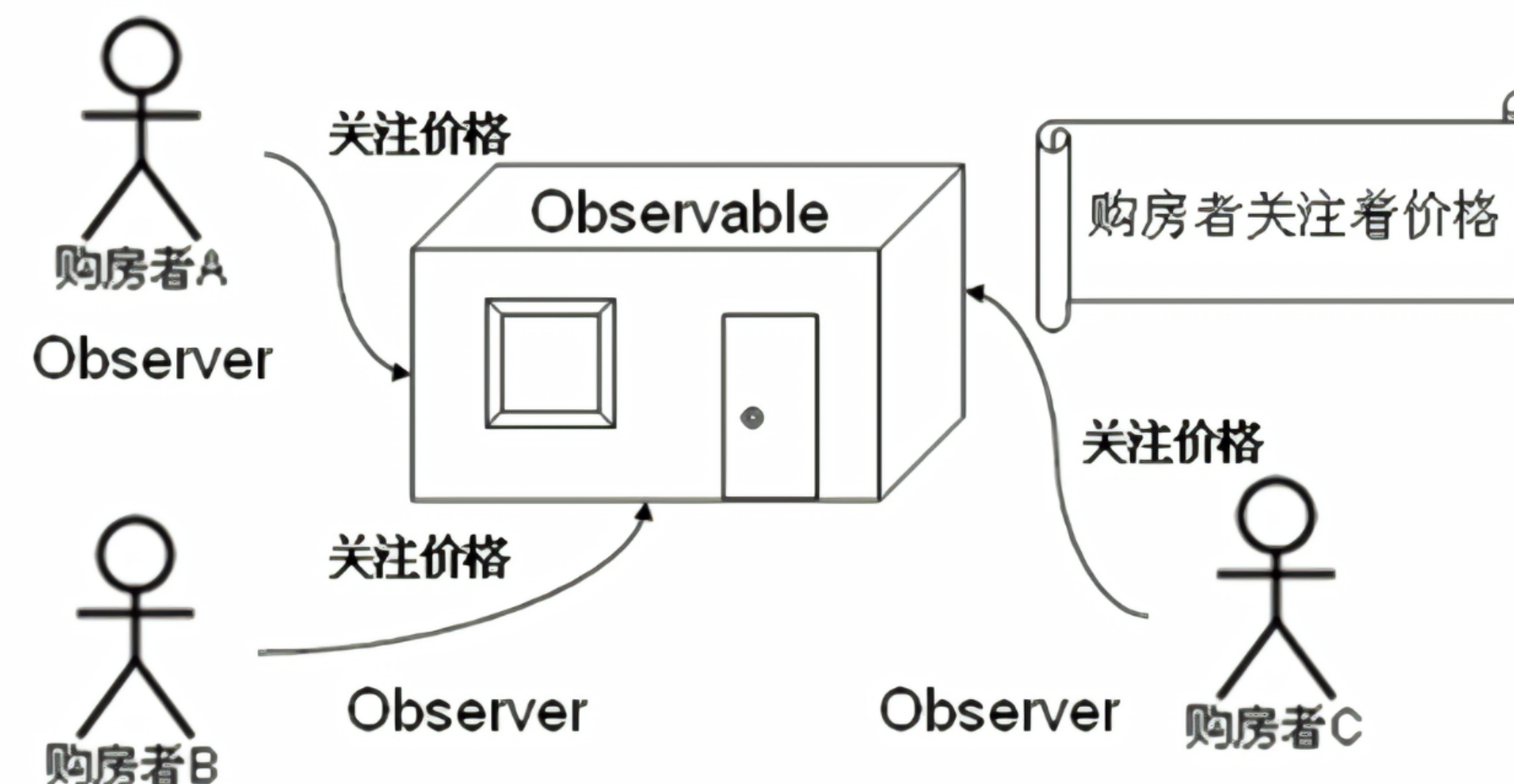
观察者模式

优点

- 1.解耦：被观察者与观察者只通过接口联系，被观察者并不知道观察者的具体实现。
- 2.灵活扩展：可以在运行时添加或移除观察者，不影响系统其余部分。
- 3.符合开闭原则：想要新增一种观察者，直接实现接口并注册即可，无需修改被观察者代码。

缺点

- 1.通知机制不保证先后顺序：如果有很多观察者，默认没指定通知次序。
- 2.可能导致频繁通知：如果被观察者变化很频繁，观察者都会收到多次更新，带来性能开销。
- 3.易出现循环依赖：观察者再去修改 Subject 的状态，可能再次引发通知，需要小心设计。



Observer Pattern

观察者模式

发布-订阅模式vs观察者模式

虽然常被混用，但严格来说：

- **观察者模式**是直接在 Subject 中维护 Observer 列表并逐一通知；
- **发布-订阅模式**通常有**第三方中介**（EventBus、MQ）做消息分发，被观察者（Publisher）只负责发布消息到中介，中介再分发给所有订阅者（Subscriber），耦合更低。

但就思路和用途而言，它们都属于“**一对多事件通知**”的范畴，可近似视为同一类思想在不同规模下的实现。

观察者模式



发布-订阅者模式



Observer Pattern

观察者模式

java内置的observer api

```
import java.util.Observable;  
import java.util.Observer;
```

在java 9后的版本不再使用
手动构造Observer或调用新API
例如EventListener

观察者模式



发布-订阅者模式



Observer Pattern - Practice

观察者模式 - 练习

场景描述

设计一个“股票市场系统”。在这个系统中，有一个股票数据发布器（Subject），负责维护某只股票的价格数据。系统中有多个观察者（Observer），比如用户显示界面或者分析应用，当股票价格更新时，这些观察者会收到通知并刷新自己的显示信息。

功能需求

- 定义一个 `Observer` 接口，包含一个方法用于接收股票价格更新；
 - 定义一个 `Subject` 接口，包含注册、移除和通知观察者的方法；
 - 实现一个具体的被观察者类（比如 `StockData` 或 `StockMarket`），维护股票代码和价格，并在价格发生变化时通知所有注册观察者；
 - 实现至少一个具体的观察者类（比如 `StockDisplay`），在收到更新时打印出新的价格信息；
 - 编写一个测试类（例如 `Main` 类），模拟股票价格变化的过程，并验证所有观察者都能正确收到通知。
-
- 思考并扩展：
 - 如何添加一个新的观察者，使其在股票价格超过某个阈值时发出特别提醒？
 - 如何修改 `StockData` 使得它不仅通知价格更新，还能传递更多的股票数据（如成交量、涨跌幅等）？

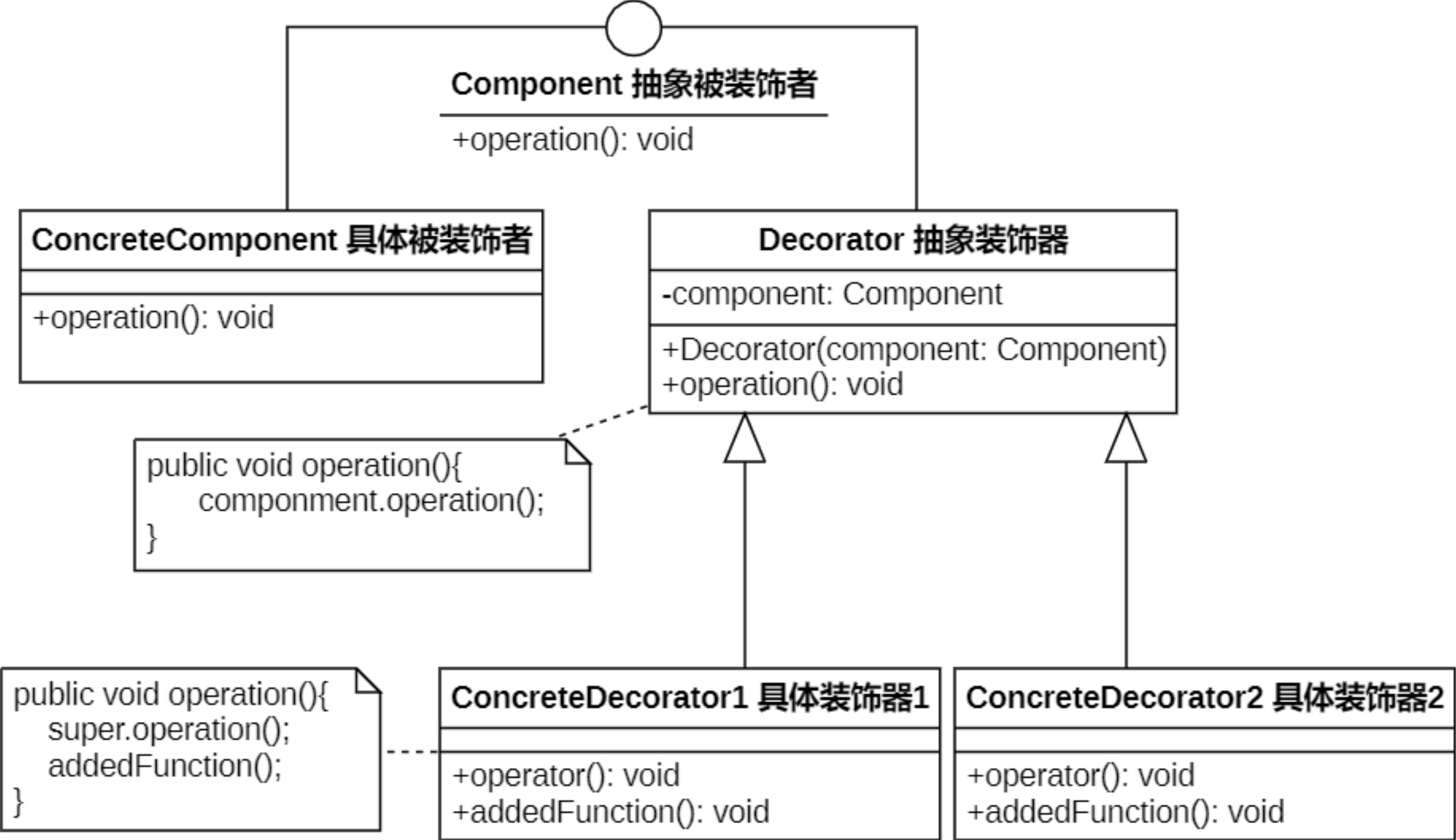
Decorator Pattern

装饰模式

Decorator Pattern

装饰者模式

装饰者模式（Decorator Pattern）允许你动态地为一个对象添加新的功能，而不改变其结构或类定义。

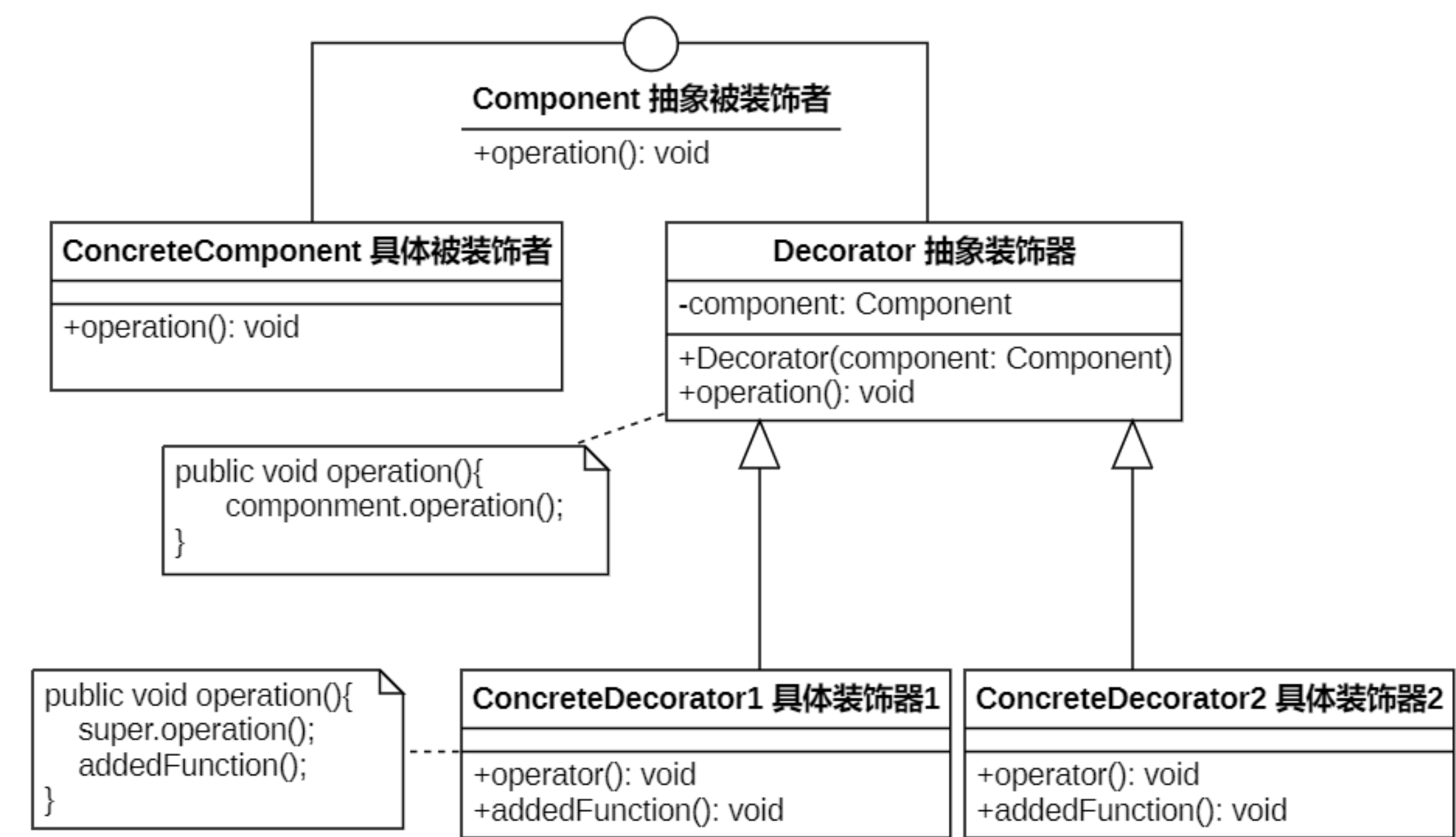


装饰器模式的结构

Decorator Pattern

装饰者模式

- Component（组件） - 原始功能的统一接口
- ConcreteComponent（具体组件） - 实现原始功能的类（可以被装饰）
- Decorator（装饰器抽象类） - 包含一个Component的引用
- ConcreteDecorator（装饰器的实现） - 为Component添加新功能实现



装饰器模式的结构

Decorator Pattern

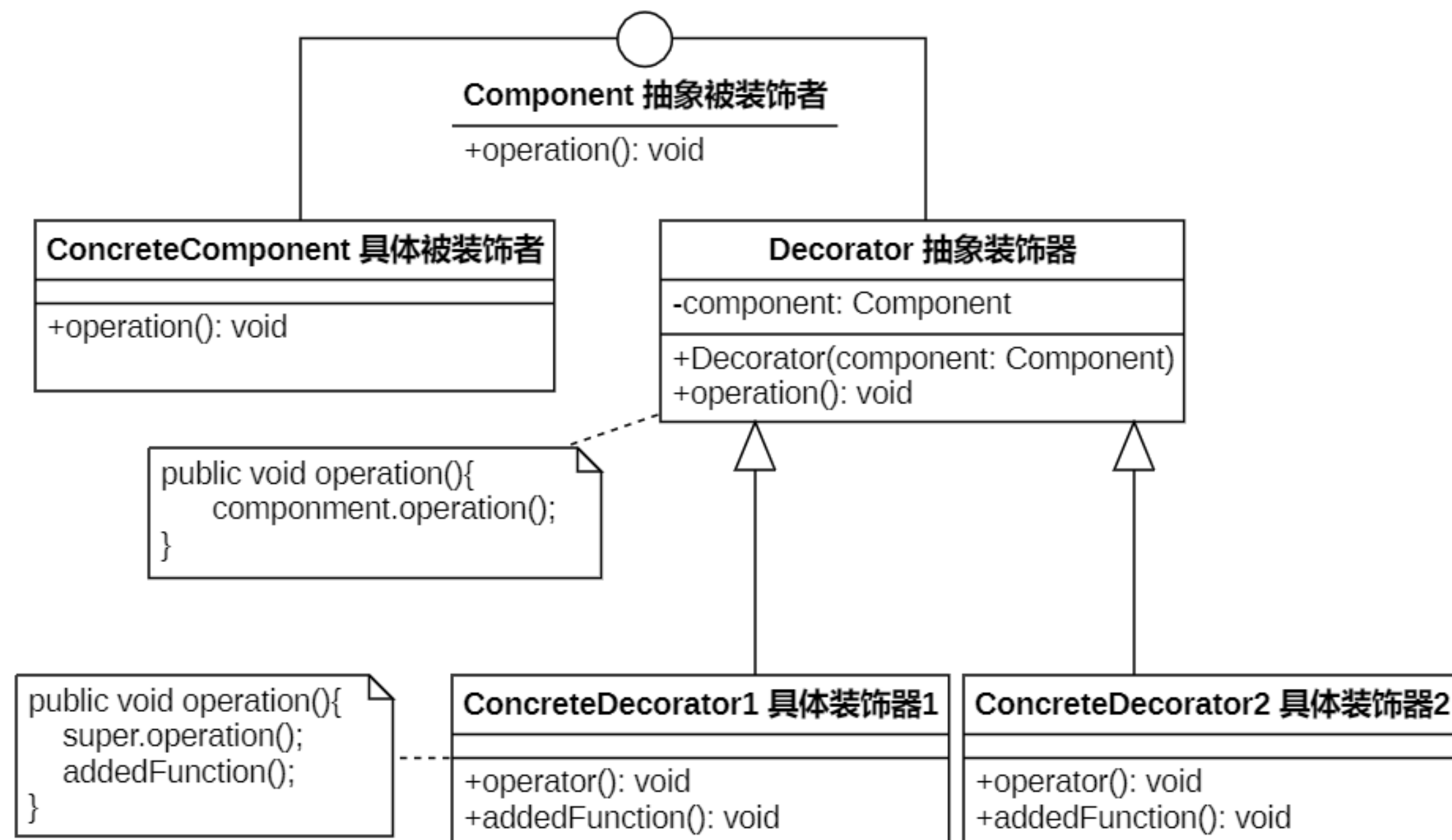
装饰者模式

Component（组件） - 原始功能的统一接口

ConcreteComponent（具体组件） - 实现原始功能的类（可以被装饰）

Decorator（装饰器抽象类） - 包含一个Component的引用

ConcreteDecorator（装饰器的实现） - 为Component添加新功能实现



装饰器模式的结构

比如：

- 一杯咖啡： BasicCoffee（原始组件）
- 加奶： MilkDecorator
- 加糖： SugarDecorator

Decorator Pattern - Example

装饰者模式 - 示例

Component（组件）- 原始功能的统一接口

ConcreteComponent（具体组件）- 实现原始功能的类（可以被装饰）

Decorator（装饰器抽象类）- 包含一个Component的引用

ConcreteDecorator（装饰器的实现）- 为Component添加新功能实现

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

Decorator Pattern - Example

装饰者模式 - 示例

Component（组件） - 原始功能的统一接口

ConcreteComponent（具体组件） - 实现原始功能的类（可以被装饰）

Decorator（装饰器抽象类） - 包含一个Component的引用

ConcreteDecorator（装饰器的实现） - 为Component添加新功能实现

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

```
public class BasicCoffee implements Coffee {  
    @Override  
    public String getDescription() {  
        return "原味咖啡";  
    }  
  
    @Override  
    public double getCost() {  
        return 10.0;  
    }  
}
```

Decorator Pattern - Example

装饰者模式 - 示例

Component（组件）- 原始功能的统一接口

ConcreteComponent（具体组件）- 实现原始功能的类（可以被装饰）

Decorator（装饰器抽象类）- 包含一个Component的引用

ConcreteDecorator（装饰器的实现）- 为Component添加新功能实现

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

```
public class BasicCoffee implements Coffee {  
    @Override  
    public String getDescription() {  
        return "原味咖啡";  
    }  
  
    @Override  
    public double getCost() {  
        return 10.0;  
    }  
}
```

```
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();  
    }  
}
```


Decorator Pattern - Example

装饰者模式 - 示例

Component（组件） - 原始功能的统一接口

ConcreteComponent（具体组件） - 实现原始功能的类（可以被装饰）

Decorator（装饰器抽象类） - 包含一个Component的引用

ConcreteDecorator（装饰器的实现） - 为Component添加新功能实现

```
public interface Coffee {  
    String getDescription();  
    double getCost();  
}  
  
public class BasicCoffee implements Coffee {  
    @Override  
    public String getDescription() {  
        return "原味咖啡";  
    }  
    @Override  
    public double getCost() {  
        return 10.0;  
    }  
}
```

```
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();  
    }  
}
```

```
public class MilkDecorator extends CoffeeDecorator {  
    public MilkDecorator(Coffee coffee) {  
        super(coffee);  
    }  
    @Override  
    public String getDescription() {  
        return super.getDescription() + " + 牛奶";  
    }  
    @Override  
    public double getCost() {  
        return super.getCost() + 2.0;  
    }  
}
```

Decorator Pattern - Example

装饰者模式 - 示例

Main.java 整合使用

```
public class Main {  
    public static void main(String[] args) {  
        Coffee coffee = new BasicCoffee();           // 原味  
        coffee = new MilkDecorator(coffee);          // 加牛奶  
        coffee = new SugarDecorator(coffee);          // 加糖  
        coffee = new SugarDecorator(coffee);          // 再加糖一次!  
  
        System.out.println("描述: " + coffee.getDescription());  
        System.out.println("总价: " + coffee.getCost() + " 元");  
    }  
}
```

> 描述: 原味咖啡 + 牛奶 + 糖 + 糖

> 总价: 14.0 元

Open-Closed Principle, OCP

开发-关闭 原则

软件实体（类、模块、函数等）应该对扩展开放，对修改关闭

“对扩展开放” = 可以添加新功能

“对修改关闭” = 不动已有代码

举个例子：

有一个类 BasicCoffee，你现在想给它加“牛奶、糖、巧克力”等新功能。

-  不应该的做法：直接去改 BasicCoffee 类，加入 if 判断。
-  应该的做法：通过新建装饰器类 MilkDecorator、SugarDecorator，扩展功能而不是改原类。

```
Coffee coffee = new BasicCoffee();  
coffee = new MilkDecorator(coffee); // 添加牛奶  
coffee = new SugarDecorator(coffee); // 添加糖
```

Decorator Pattern

装饰者模式

1. 符合开放-关闭原则

不改原类代码即可增加功能，扩展性强

2. 可组合、可层层叠加

多个装饰器可按需叠加，功能组合灵活

3. 可替代继承实现扩展

比类继承更灵活（不增加子类数量）

4. 可随时添加或移除装饰

运行时动态决定对象行为

5. 更细粒度的控制

每个装饰器只做一件事，遵循“单一职责”原则

6. 不影响原有系统

装饰器独立于被包装类，不会破坏原有结构

```
Coffee c = new BasicCoffee();  
c = new MilkDecorator(c);  
c = new SugarDecorator(c);
```

没动原始 BasicCoffee 类

每层装饰都是独立的功能模块

可以自由组合、嵌套、动态添加/移除

比继承 MilkCoffee, SugarCoffee, MilkSugarCoffee 更优雅

Decorator Pattern

装饰者模式

1. 结构复杂

每增加一种功能就要增加一个装饰类，可能类数量暴增

2. 调试不方便

多层包装时难以追踪到底哪个类干了什么

3. 顺序敏感

不同装饰器组合顺序不同会影响结果，易出错

4. 不支持依赖注入的场景

如果使用构造函数组合，需要写很多重复的传参代码

5. 客户端需要了解装饰器结构

客户端需要知道怎么组合装饰器，增加使用成本

6. 不适合太多可变功能

如果装饰类太多或功能切换频繁，会变成“嵌套地狱”

你想实现 4 种功能的组合：

基础功能 + A + B + C

```
Component c =  
new CDecorator(  
    new BDecorator(  
        new ADecorator(  
            new BasicComponent()))));
```

这时候的层级看起来就很繁琐，而且：

- 想看 getDescription() 做了啥，得一层层调试进去；
- 如果顺序写错了，逻辑就变了

Decorator Pattern

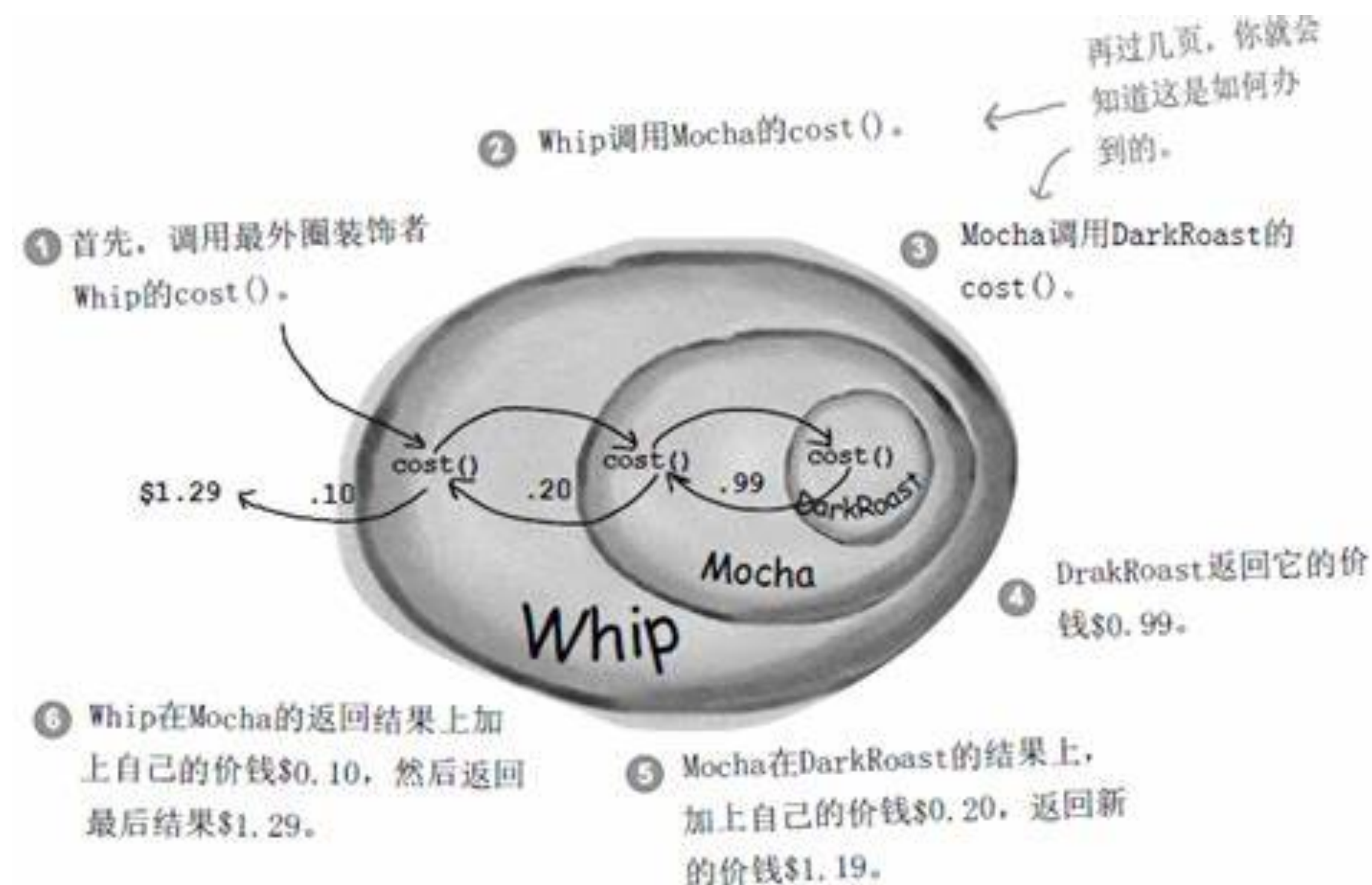
装饰者模式

适合场景：

- 功能扩展是可选的、可组合的
- 不想破坏原类结构（不能改、来自第三方库等）
- 不想写太多子类（继承层级过深）
- 功能较为独立、关注点单一

不适合场景：

- 功能变化不多，简单 if/else 就能解决
- 功能组合顺序敏感且依赖复杂
- 要处理状态、生命周期复杂
- 对性能敏感（装饰器链过长带来额外调用开销）



Assignment

课后作业

- 预习 Headfirst Design 中 Singleton Pattern, Composite Design Pattern
- 使用装饰器模式设计编写以下任务：
- 顾客订购一个基础三明治（BasicSandwich），随后可以根据口味添加各种装饰，例如奶酪（Cheese）、番茄（Tomato）等。每种配料会增加一定的描述和价格。
要求使用装饰器模式，使得代码在不修改基础三明治类的情况下，通过“包装”实现动态扩展。

作业目标

1. 定义一个 Sandwich 接口，包含两个方法：
 - `String getDescription();`
 - `double getCost();`
2. 实现一个基础的三明治类 BasicSandwich，作为基本组件。
3. 编写一个抽象装饰器 SandwichDecorator，实现 Sandwich 接口，并持有一个 Sandwich 对象。
4. 实现至少两个具体装饰器类，例如 CheeseDecorator 和 TomatoDecorator，分别对描述和价格进行“装饰”。
5. 编写一个主类 Main，模拟顾客订单：构造一个基础三明治，然后依次装饰，最终输出最终的描述和价格。