

# 코딩으로 익히는 디자인 패턴

누군가 이미 그 문제를 풀어놨다면?

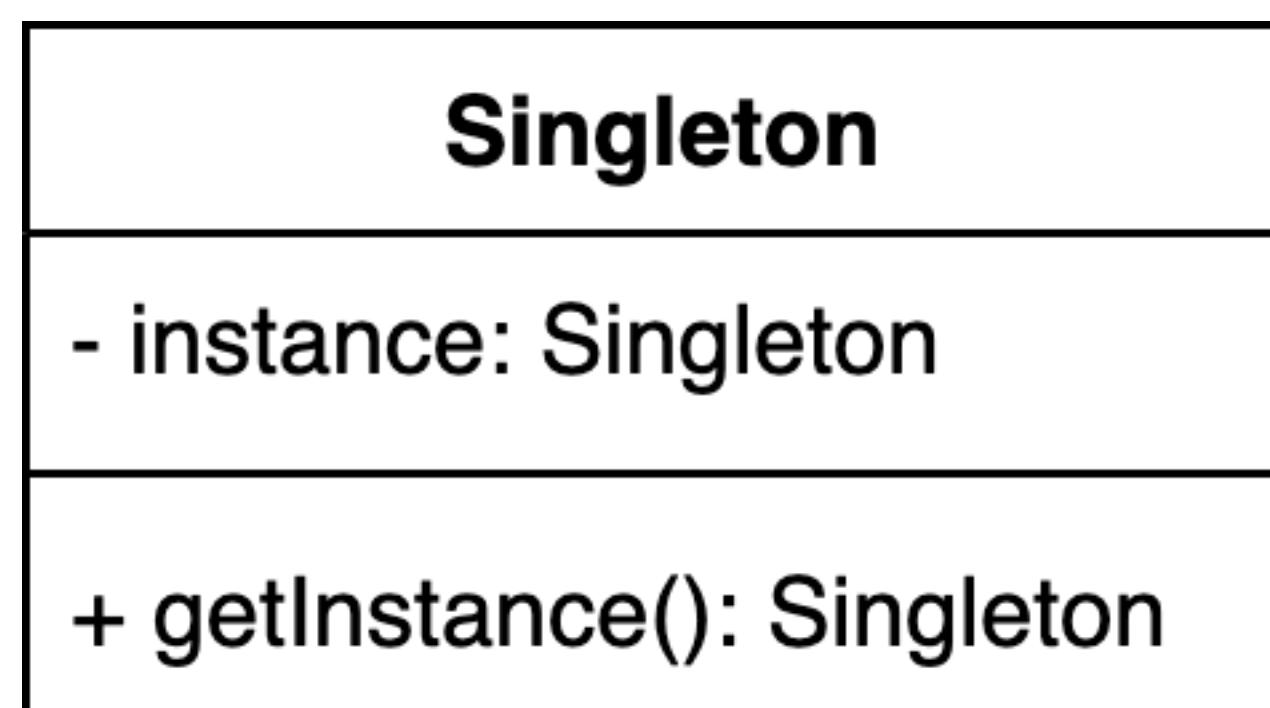
백기선 (WHITESHIP)

# **생성 패턴 (Creational Patterns)**

# 싱글톤 (Singleton) 패턴

인스턴스를 오직 한개만 제공하는 클래스

- 시스템 런타임, 환경 세팅에 대한 정보 등, 인스턴스가 여러개 일 때 문제가 생길 수 있는 경우가 있다. **인스턴스를 오직 한개만 만들어 제공하는 클래스가 필요하다.**



# 싱글톤 (Singleton) 패턴 구현 방법 1

private 생성자에 static 메소드

```
public class Settings {  
  
    private static Settings instance;  
  
    private Settings() {}  
  
    public static Settings getInstance() {  
        if (instance == null) {  
            instance = new Settings();  
        }  
  
        return instance;  
    }  
  
}
```

다음을 직접 설명해 보세요.

1. 생성자를 private으로 만든 이유?
2. getInstance() 메소드를 static으로 선언한 이유?
3. getInstance()가 멀티쓰레드 환경에서 안전하지 않은 이유?

# 싱글톤 (Singleton) 패턴 구현 방법 2

동기화(synchronized)를 사용해 멀티쓰레드 환경에 안전하게 만드는 방법

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

다음을 직접 설명해 보세요.

1. 자바의 동기화 블럭 처리 방법은?
2. getInstance() 메소드 동기화시 사용하는 락(lock)은 인스턴스의 락인가 클래스의 락인가? 그 이유는?

# 싱글톤 (Singleton) 패턴 구현 방법 3

이른 초기화 (eager initialization)을 사용하는 방법

```
private static final Settings INSTANCE = new Settings();
```

```
private Settings() {}
```

```
public static Settings getInstance() {  
    return INSTANCE;  
}
```

다음을 직접 설명해 보세요.

1. 이른 초기화가 단점이 될 수도 있는 이유?
2. 만약에 생성자에서 checked 예외를 던진다면  
이 코드를 어떻게 변경해야 할까요?

# 싱글톤 (Singleton) 패턴 구현 방법 4

double checked locking으로 효율적인 동기화 블럭 만들기

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

다음을 직접 설명해 보세요.

1. double check locking이라고 부르는 이유?
2. instance 변수는 어떻게 정의해야 하는가?  
그 이유는?

# 싱글톤 (Singleton) 패턴 구현 방법 5

static inner 클래스를 사용하는 방법

```
private Settings() {}

private static class SettingsHolder {
    private static final Settings SETTINGS = new Settings();
}

public static Settings getInstance() {
    return SettingsHolder.SETTINGS;
}
```

다음을 직접 설명해 보세요.

1. 이 방법은 static final을 썼는데도 왜 지연 초기화 (lazy initialization)라고 볼 수 있는가?

# 싱글톤 (Singleton) 패턴 구현 깨트리는 방법 1

리플렉션을 사용한다면?

```
Settings settings = Settings.getInstance();
```

```
Constructor<Settings> declaredConstructor = Settings.class.getDeclaredConstructor();
declaredConstructor.setAccessible(true);
Settings settings1 = declaredConstructor.newInstance();
```

```
System.out.println(settings == settings1);
```

다음을 직접 설명해 보세요.

1. 리플렉션에 대해 설명하세요.

2. setAccessible(true)를 사용하는 이유는?

# 싱글톤 (Singleton) 패턴 구현 깨트리는 방법 2

직렬화 & 역직렬화를 사용한다면?

```
Settings settings = Settings.getInstance();
Settings settings1 = null;
try (ObjectOutput out = new ObjectOutputStream(new FileOutputStream("settings.obj"))) {
    out.writeObject(settings);
}

try (ObjectInput in = new ObjectInputStream(new FileInputStream("settings.obj"))) {
    settings1 = (Settings) in.readObject();
}

System.out.println(settings == settings1);
```

다음을 직접 설명해 보세요.

1. 자바의 직렬화 & 역직렬화에 대해 설명하세요.
2. SerializableId란 무엇이며 왜 쓰는가?
3. try-resource 블럭에 대해 설명하세요.

# 싱글톤 (Singleton) 패턴 구현 방법 6

## enum을 사용하는 방법

```
public enum Settings {  
    INSTANCE;  
}
```

다음을 직접 설명해 보세요.

1. enum 타입의 인스턴스를 리팩토링을 만들 수 있는가?
2. enum으로 싱글톤 타입을 구현할 때의 단점은?
3. 직렬화 & 역직렬화 시에 별도로 구현해야 하는 메소드가 있는가?

# 싱글톤 (Singleton) 패턴 복습

직접 말하며 설명하고 코딩해 보세요.

- 자바에서 enum을 사용하지 않고 싱글톤 패턴을 구현하는 방법은?
- private 생성자와 static 메소드를 사용하는 방법의 단점은?
- enum을 사용해 싱글톤 패턴을 구현하는 방법의 장점과 단점은?
- static inner 클래스를 사용해 싱글톤 패턴을 구현하라.

# 싱글톤 (Singleton) 패턴

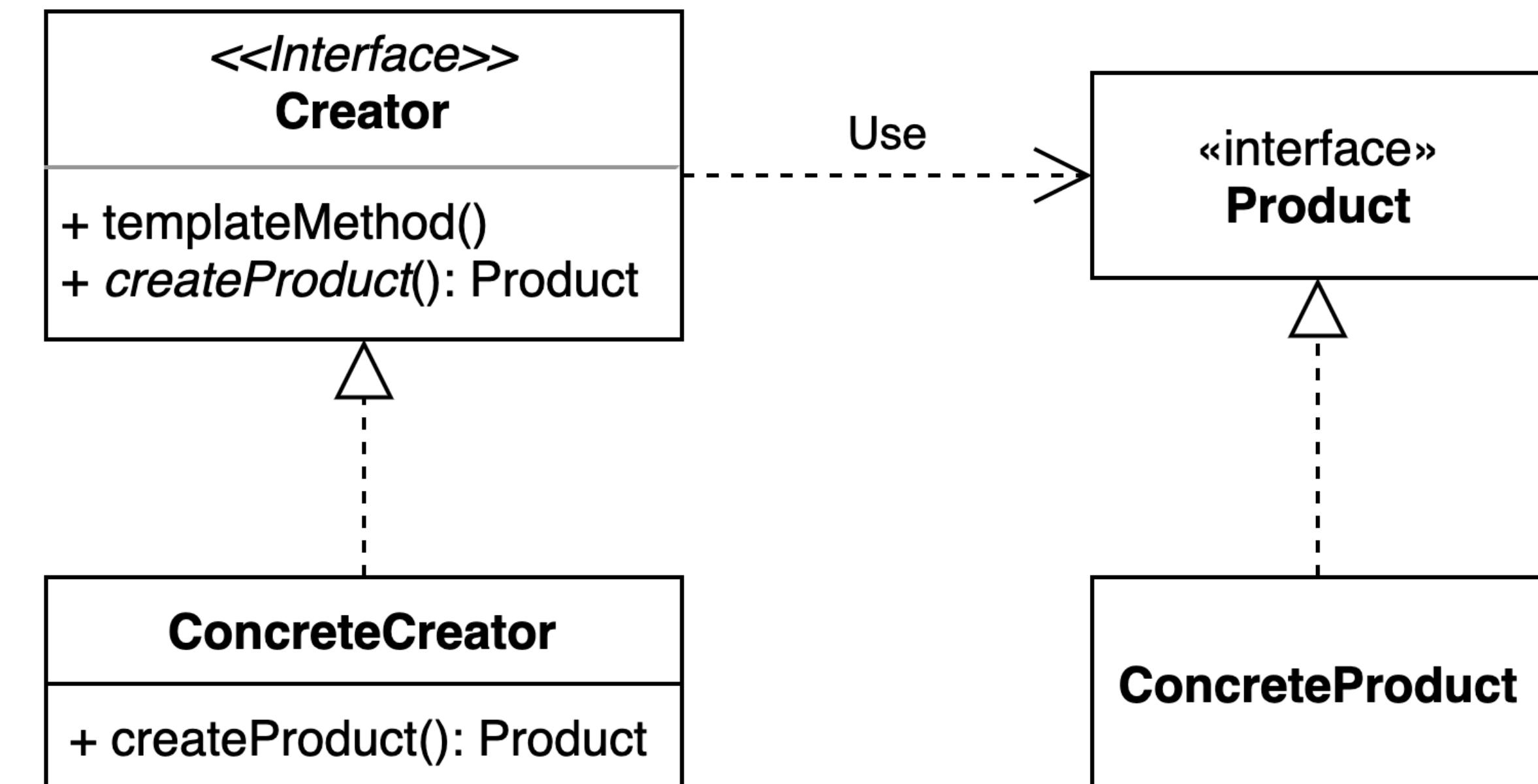
실무에서는 어떻게 쓰이나?

- 스프링에서 빈의 스코프 중에 하나로 싱글톤 스코프.
- 자바 `java.lang.Runtime`
- 다른 디자인 패턴(빌더, 퍼사드, 추상 팩토리 등) 구현체의 일부로 쓰이기도 한다.

# 팩토리 메소드 (Factory method) 패턴

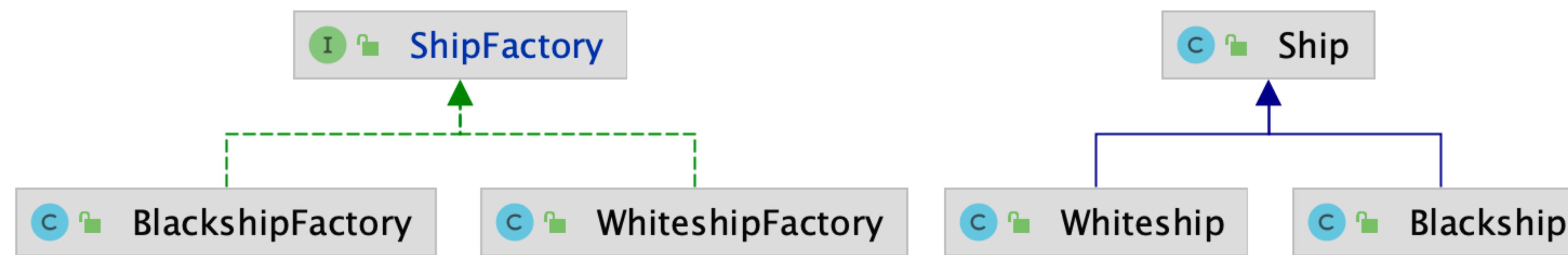
구체적으로 어떤 인스턴스를 만들지는 서브 클래스가 정한다.

- 다양한 구현체 (Product)가 있고, 그중에서 특정한 구현체를 만들 수 있는 다양한 팩토리 (Creator)를 제공할 수 있다.



# 팩토리 메소드 (Factory method) 패턴 구현 방법

확장에 열려있고 변경에 달혀있는 구조로 만들어보자.



# 팩토리 메소드 (Factory method) 패턴 복습

구체적으로 어떤 것을 만들지는 서브 클래스가 정한다.

- 팩토리 메소드 패턴을 적용했을 때의 장점은? 단점은?
- “확장에 열려있고 변경에 닫혀있는 객체 지향 원칙”을 설명하세요.
- 자바 8에 추가된 default 메소드에 대해 설명하세요.

# 팩토리 메소드 (Factory method) 패턴

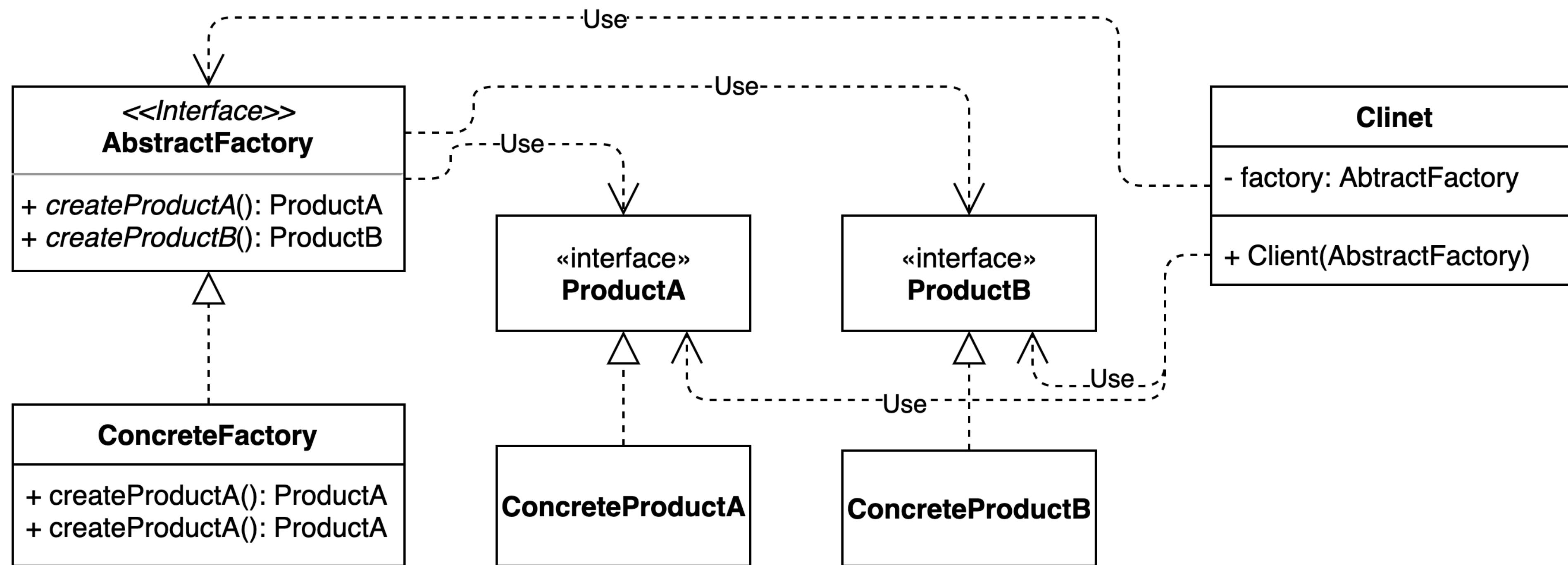
실무에서는 어떻게 쓰이나?

- 단순한 팩토리 패턴
  - 매개변수의 값에 따라 또는 메소드에 따라 각기 다른 인스턴스를 리턴하는 단순한 버전의 팩토리 패턴
  - `java.lang.Calendar` 또는 `java.lang.NumberFormat`
- 스프링 BeanFactory
  - `Object` 타입의 `Product`를 만드는 `BeanFactory`라는 Creator!

# 추상 팩토리 (Abstract factory) 패턴

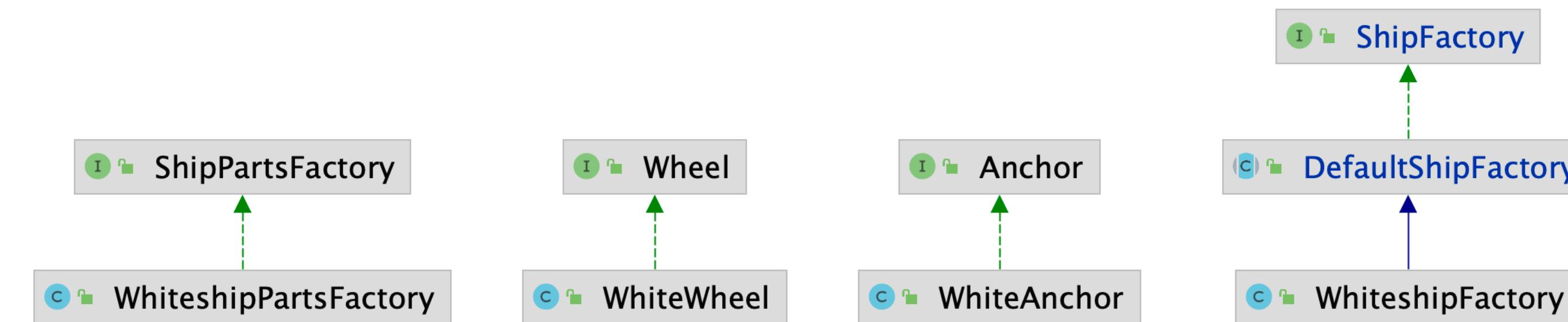
서로 관련있는 여러 객체를 만들어주는 인터페이스.

- 구체적으로 어떤 클래스의 인스턴스를(concrete product)를 사용하는지 감출 수 있다.



# 추상 팩토리 (Abstract factory) 패턴 구현 방법

클라이언트 코드에서 구체적인 클래스의 의존성을 제거한다.



# 추상 팩토리 (Abstract factory) 패턴 복습

팩토리 메소드 패턴과 굉장히 흡사한데 무엇이 다른건가.

- 모양과 효과는 비슷하지만...
  - 둘 다 구체적인 객체 생성 과정을 추상화한 인터페이스를 제공한다.
- 관점이 다르다.
  - 팩토리 메소드 패턴은 “팩토리를 구현하는 방법 (inheritance)”에 초점을 둔다.
  - 추상 팩토리 패턴은 “팩토리를 사용하는 방법 (composition)”에 초점을 둔다.
- 목적이 조금 다르다.
  - 팩토리 메소드 패턴은 구체적인 객체 생성 과정을 하위 또는 구체적인 클래스로 옮기는 것이 목적.
  - 추상 팩토리 패턴은 관련있는 여러 객체를 구체적인 클래스에 의존하지 않고 만들 수 있게 해주는 것이 목적.

# 추상 팩토리 (Abstract factory) 패턴

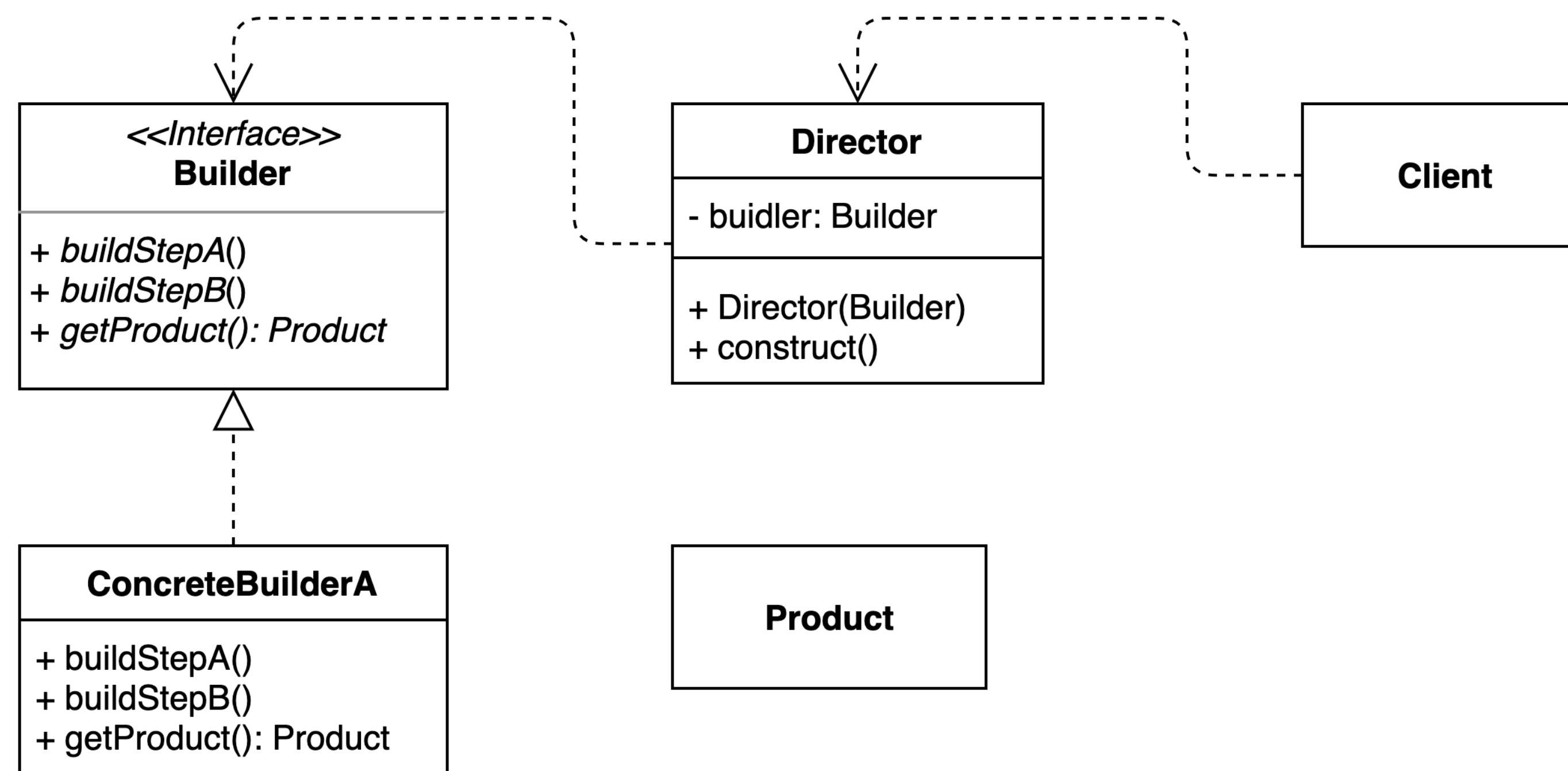
실무에서는 어떻게 쓰이나?

- 자바 라이브러리
  - javax.xml.xpath.XPathFactory#newInstance()
  - javax.xml.transform.TransformerFactory#newInstance()
  - javax.xml.parsers.DocumentBuilderFactory#newInstance()
- 스프링
  - FactoryBean과 그 구현체

# 빌더 (Builder) 패턴

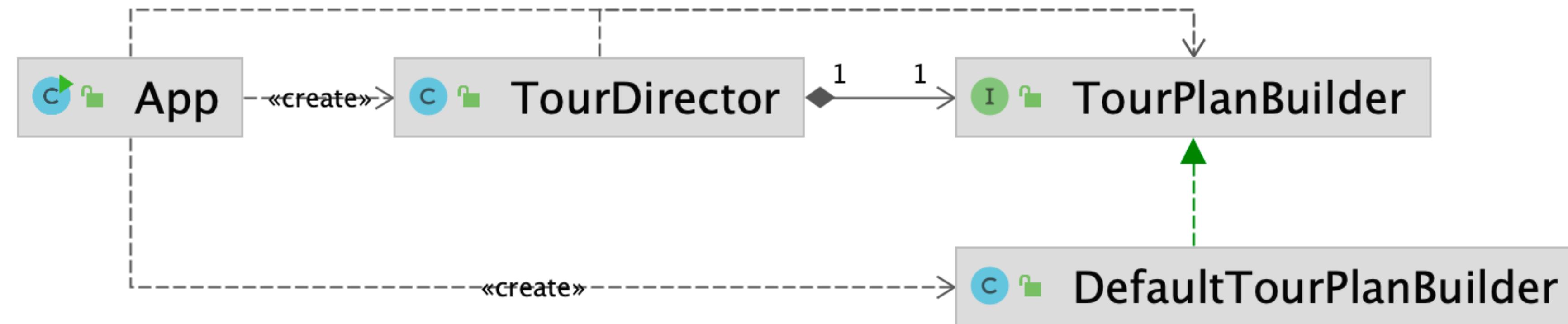
동일한 프로세스를 거쳐 다양한 구성의 인스턴스를 만드는 방법.

- (복잡한) 객체를 만드는 프로세스를 독립적으로 분리할 수 있다.



# 빌더 (Builder) 패턴 구현 방법

동일한 프로세스를 거쳐 다양한 구성의 인스턴스를 만드는 방법



# 빌더 (Builder) 패턴 구현 복습

동일한 프로세스를 거쳐 다양한 구성의 인스턴스를 만드는 방법

- 장점
  - 만들기 복잡한 객체를 순차적으로 만들 수 있다.
  - 복잡한 객체를 만드는 구체적인 과정을 숨길 수 있다.
  - 동일한 프로세스를 통해 각기 다르게 구성된 객체를 만들 수도 있다.
  - 불완전한 객체를 사용하지 못하도록 방지할 수 있다.
- 단점
  - 원하는 객체를 만들려면 빌더부터 만들어야 한다.
  - 구조가 복잡해 진다. (트레이드 오프)

# 빌더 (Builder) 패턴

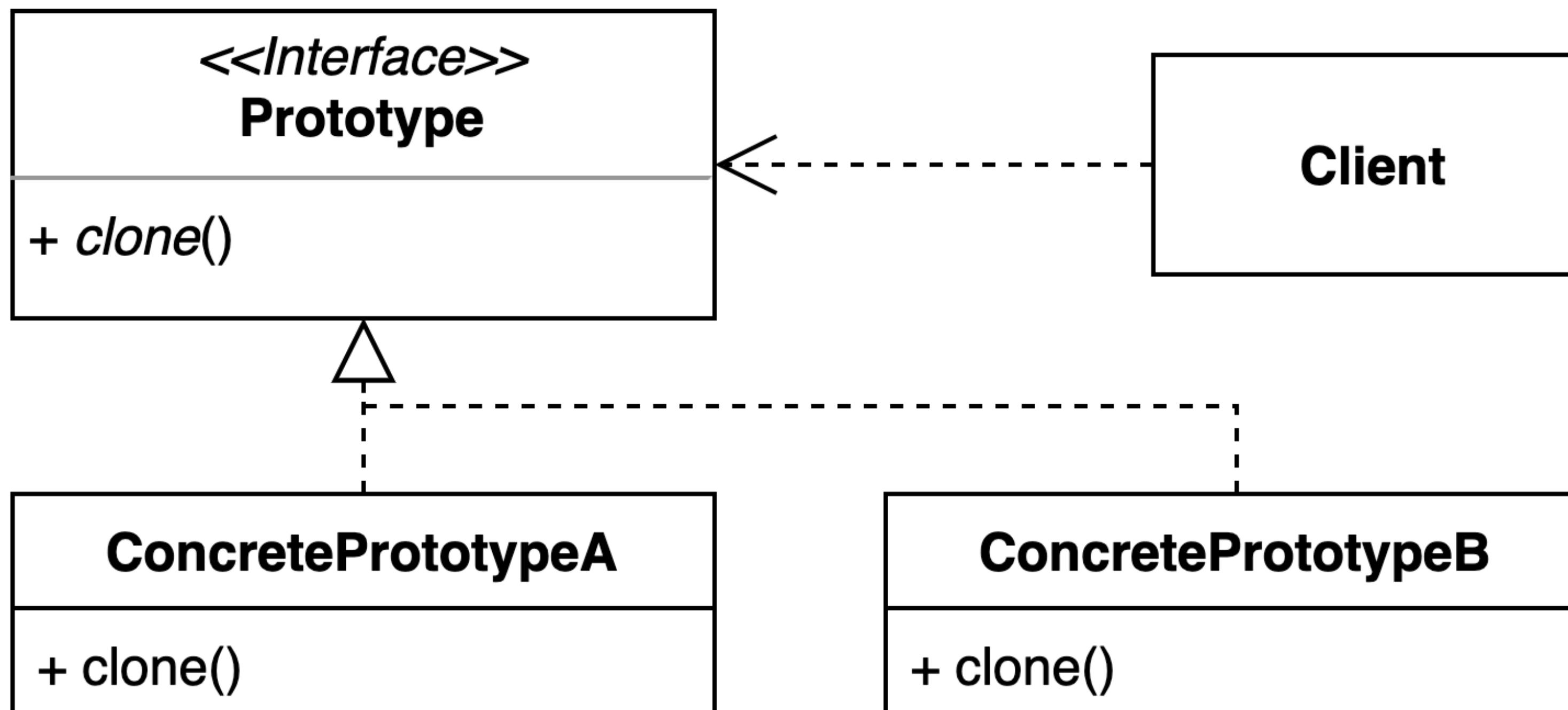
실무에서 어떻게 쓰이나?

- 자바 8 Stream.Buidler API
- StringBuilder는 빌더 패턴일까?
- 롬복의 @Builder
  - <https://projectlombok.org/features/Builder>
- 스프링
  - UriComponentsBuilder
  - MockMvcWebClientBuilder
  - ...Builder

# 프로토타입 (Prototype) 패턴

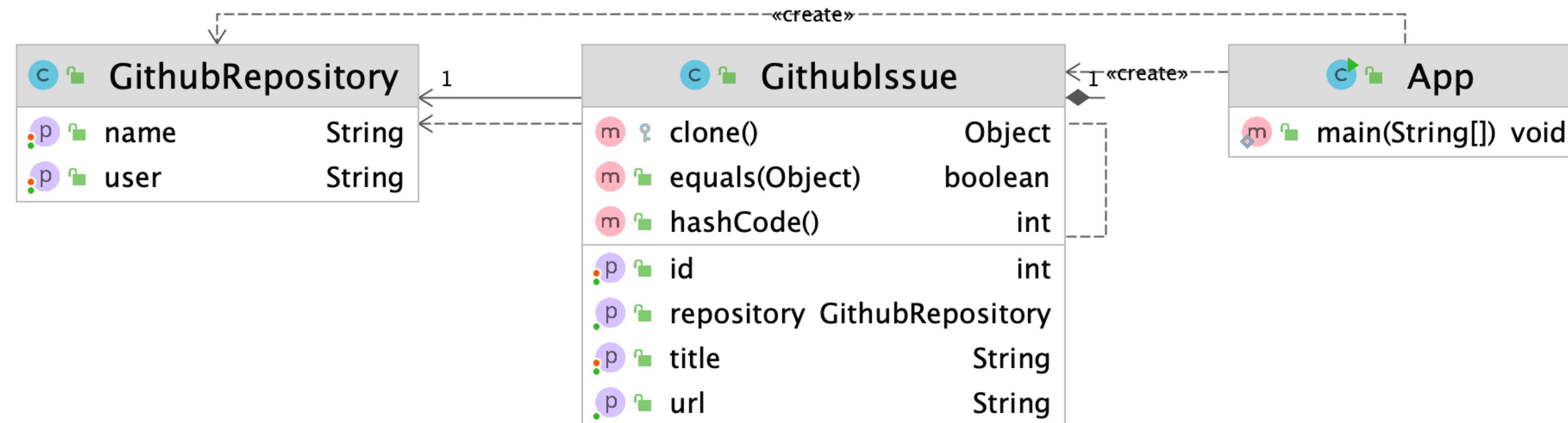
기존 인스턴스를 복제하여 새로운 인스턴스를 만드는 방법

- 복제 기능을 갖추고 있는 기존 인스턴스를 프로토타입으로 사용해 새 인스턴스를 만들 수 있다.



# 프로토타입 (Prototype) 패턴 구현 방법

기존 인스턴스를 복제하여 새로운 인스턴스를 만드는 방법



# 프로토타입 (Prototype) 패턴 구현 복습

기존 인스턴스를 복제하여 새로운 인스턴스를 만드는 방법

- 장점
  - 복잡한 객체를 만드는 과정을 숨길 수 있다.
  - 기존 객체를 복제하는 과정이 새 인스턴스를 만드는 것보다 비용(시간 또는 메모리)적인 면에서 효율적일 수도 있다.
  - 추상적인 타입을 리턴할 수 있다.
- 단점
  - 복제한 객체를 만드는 과정 자체가 복잡할 수 있다. (특히, 순환 참조가 있는 경우)

# **프로토타입 (Prototype) 패턴**

실무에서 어떻게 쓰이나?

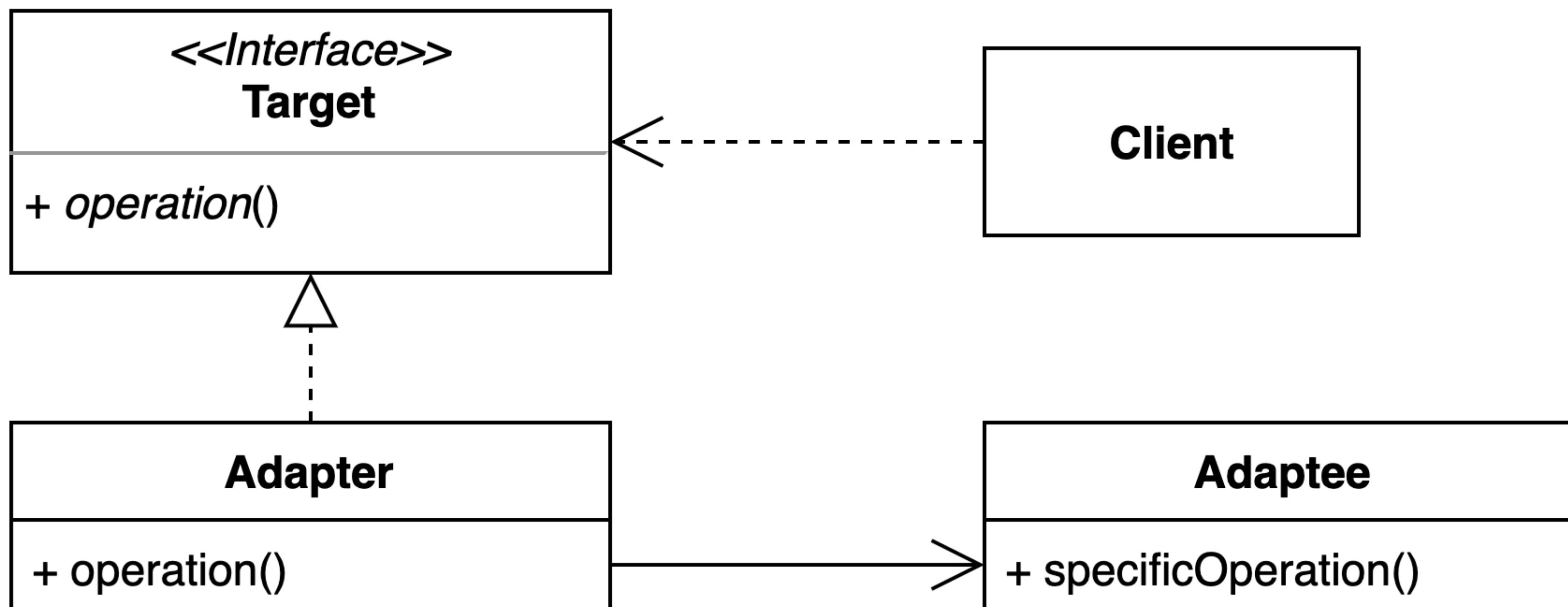
- 자바 Object 클래스의 clone 메소드와 Cloneable 인터페이스
- shallow copy와 deep copy
- ModelMapper

# **구조적인 패턴 (Structural Patterns)**

# 어댑터 (Adapter) 패턴

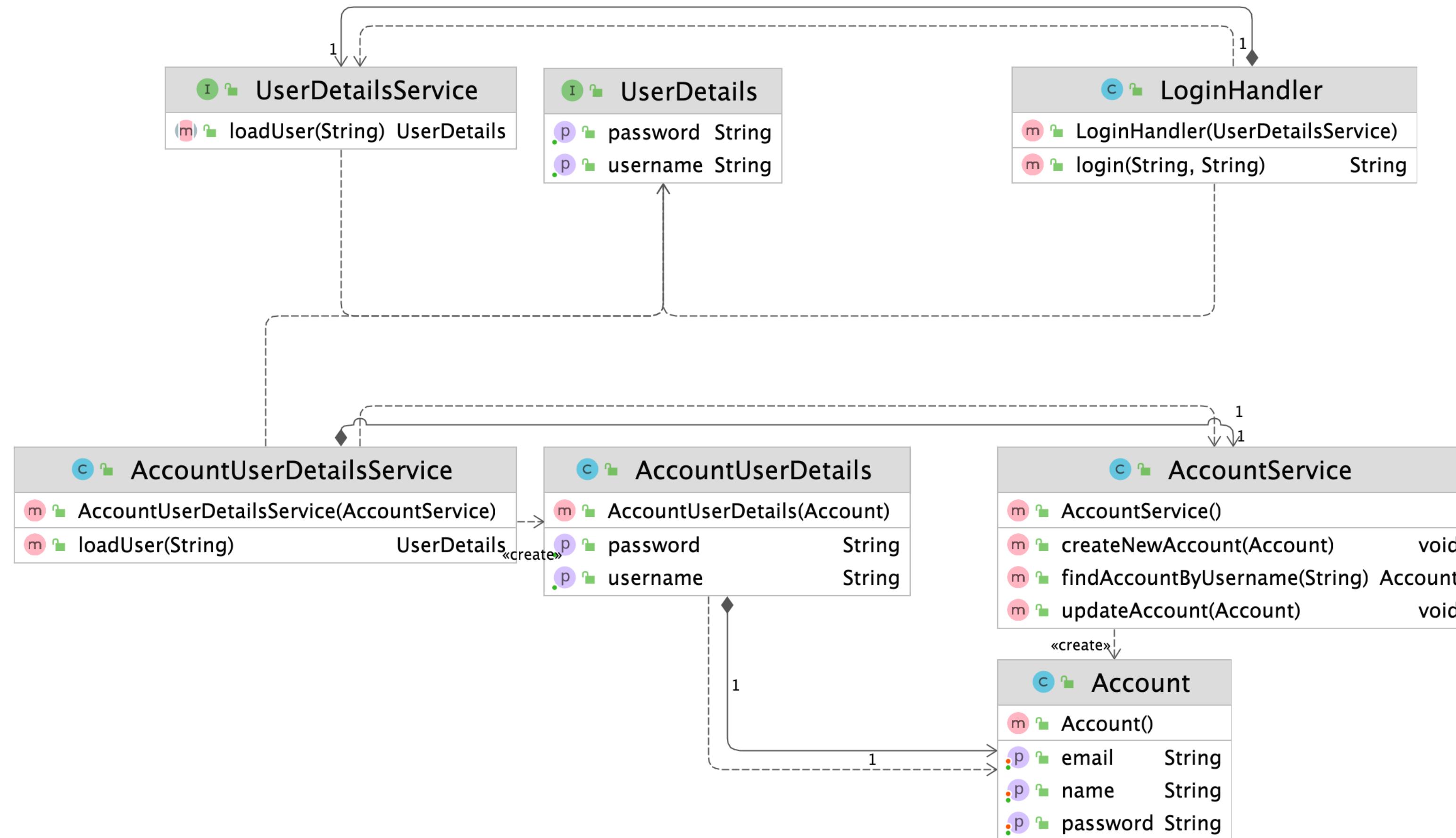
기존 코드를 클라이언트가 사용하는 인터페이스의 구현체로 바꿔주는 패턴

- 클라이언트가 사용하는 인터페이스를 따르지 않는 기존 코드를 재사용할 수 있게 해준다.



# 어댑터 (Adapter) 패턴

기존 코드를 클라이언트가 사용하는 인터페이스의 구현체로 바꿔주는 패턴



# 어댑터 (Adapter) 패턴

기존 코드를 클라이언트가 사용하는 인터페이스의 구현체로 바꿔주는 패턴

- 장점
  - 기존 코드를 변경하지 않고 원하는 인터페이스 구현체를 만들어 재사용할 수 있다.
  - 기존 코드가 하던 일과 특정 인터페이스 구현체로 변환하는 작업을 각기 다른 클래스로 분리하여 관리할 수 있다.
- 단점
  - 새 클래스가 생겨 복잡도가 증가할 수 있다. 경우에 따라서는 기존 코드가 해당 인터페이스를 구현하도록 수정하는 것이 좋은 선택이 될 수도 있다.

# 어댑터 (Adapter) 패턴

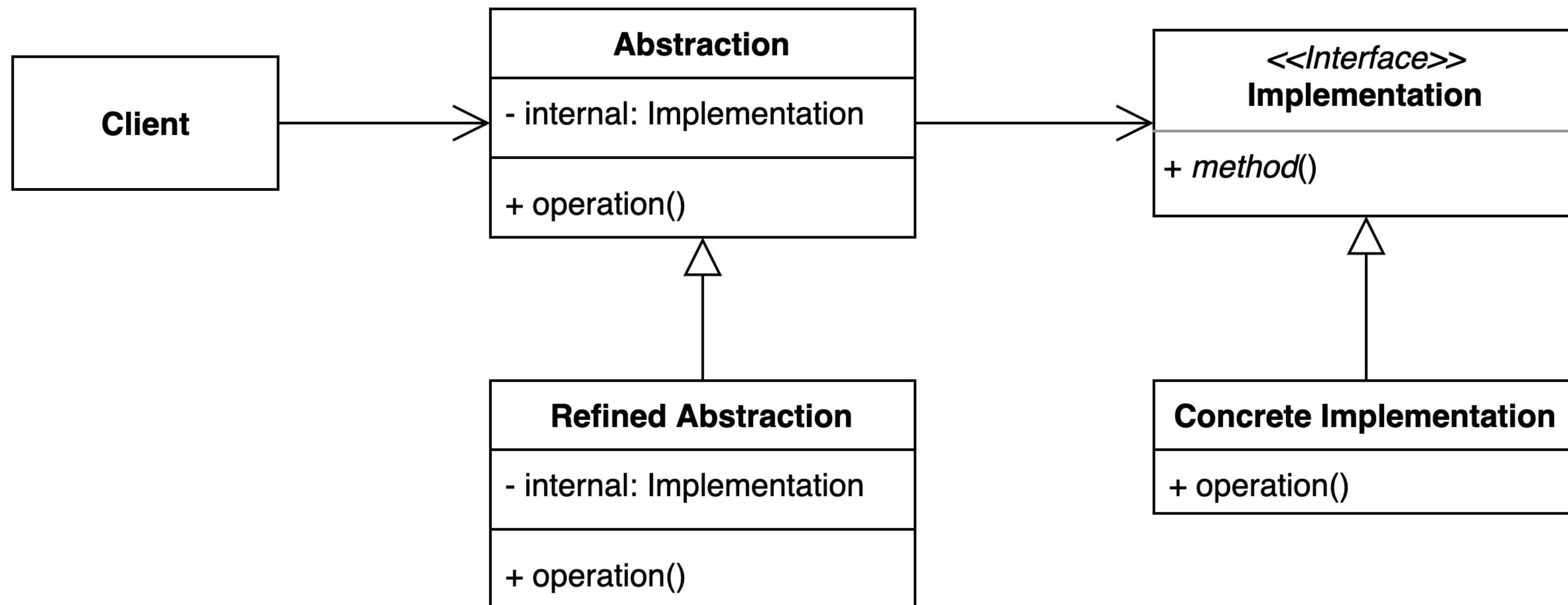
실무에서 어떻게 쓰이나?

- 자바
  - `java.util.Arrays#asList(T...)`
  - `java.util.Collections#list(Enumeration)`, `java.util.Collections#enumeration()`
  - `java.io.InputStreamReader(InputStream)`
  - `java.io.OutputStreamWriter(OutputStream)`
- 스프링
  - HandlerAdpter: 우리가 작성하는 다양한 형태의 핸들러 코드를 스프링 MVC가 실행할 수 있는 형태로 변환해주는 어댑터용 인터페이스.

# 브릿지 (Bridge) 패턴

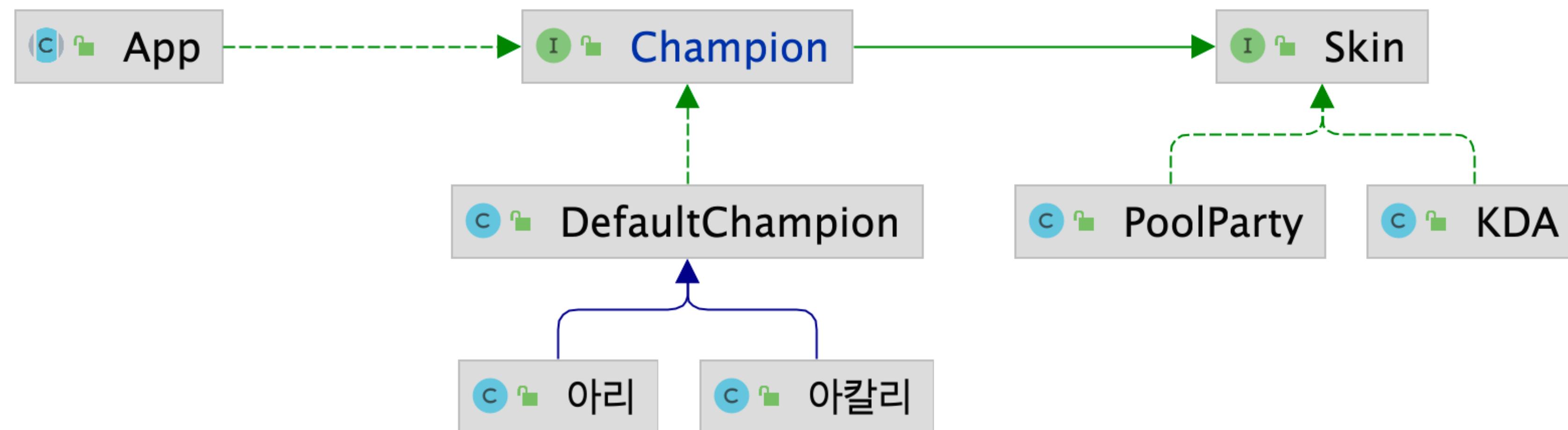
추상적인 것과 구체적인 것을 분리하여 연결하는 패턴

- 하나의 계층 구조일 때 보다 각기 나누었을 때 독립적인 계층 구조로 발전 시킬 수 있다.



# 브릿지 (Bridge) 패턴

추상적인 것과 구체적인 것을 분리하여 연결하는 패턴



# 브릿지 (Bridge) 패턴

추상적인 것과 구체적인 것을 분리하여 연결하는 패턴

- 장점
  - 추상적인 코드를 구체적인 코드 변경 없이도 독립적으로 확장할 수 있다.
  - 추상적인 코드와 구체적인 코드를 분리하여 수 있다.
- 단점
  - 계층 구조가 늘어나 복잡도가 증가할 수 있다.

# 브릿지 (Bridge) 패턴

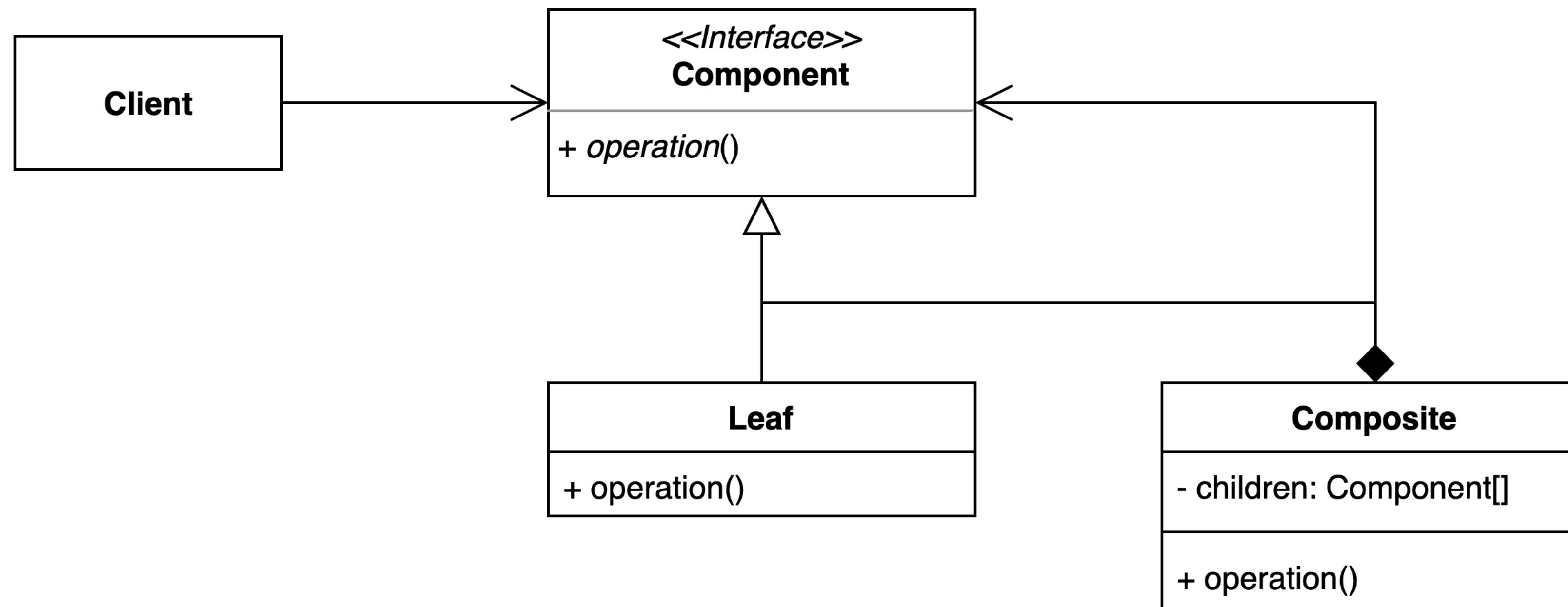
실무에서 어떻게 쓰이나?

- 자바
  - JDBC API, DriverManger와 Driver
  - SLF4J, 로깅 퍼사드와 로거
- 스프링
  - Portable Service Abstraction

# 컴포짓 (Composite) 패턴

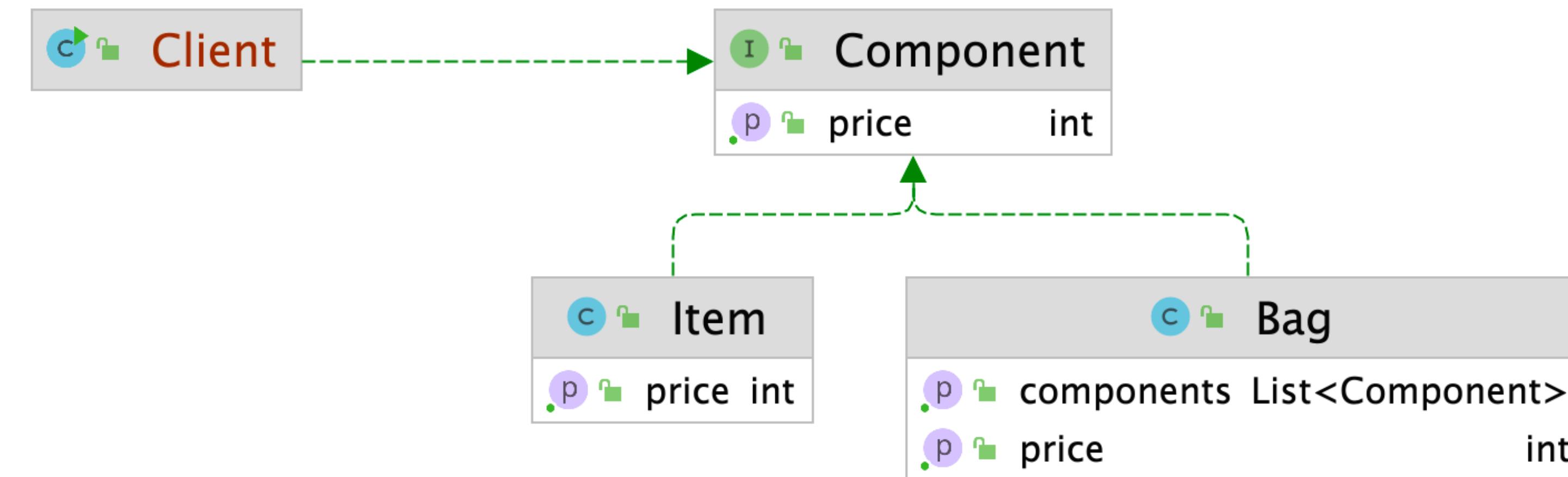
그룹 전체와 개별 객체를 동일하게 처리할 수 있는 패턴.

- 클라이언트 입장에서는 ‘전체’나 ‘부분’이나 모두 동일한 컴포넌트로 인식할 수 있는 계층 구조를 만든다. (Part-Whole Hierarchy)



# 컴포짓 (Composite) 패턴

그룹 전체와 개별 객체를 동일하게 처리할 수 있는 패턴.



# 컴포짓 (Composite) 패턴

그룹 전체와 개별 객체를 동일하게 처리할 수 있는 패턴.

- 장점
  - 복잡한 트리 구조를 편리하게 사용할 수 있다.
  - 다형성과 재귀를 활용할 수 있다.
  - 클라이언트 코드를 변경하지 않고 새로운 엘리먼트 타입을 추가할 수 있다.
- 단점
  - 트리를 만들어야 하기 때문에 (공통된 인터페이스를 정의해야 하기 때문에) 지나치게 일 반화 해야 하는 경우도 생길 수 있다.

# 컴포짓 (Composite) 패턴

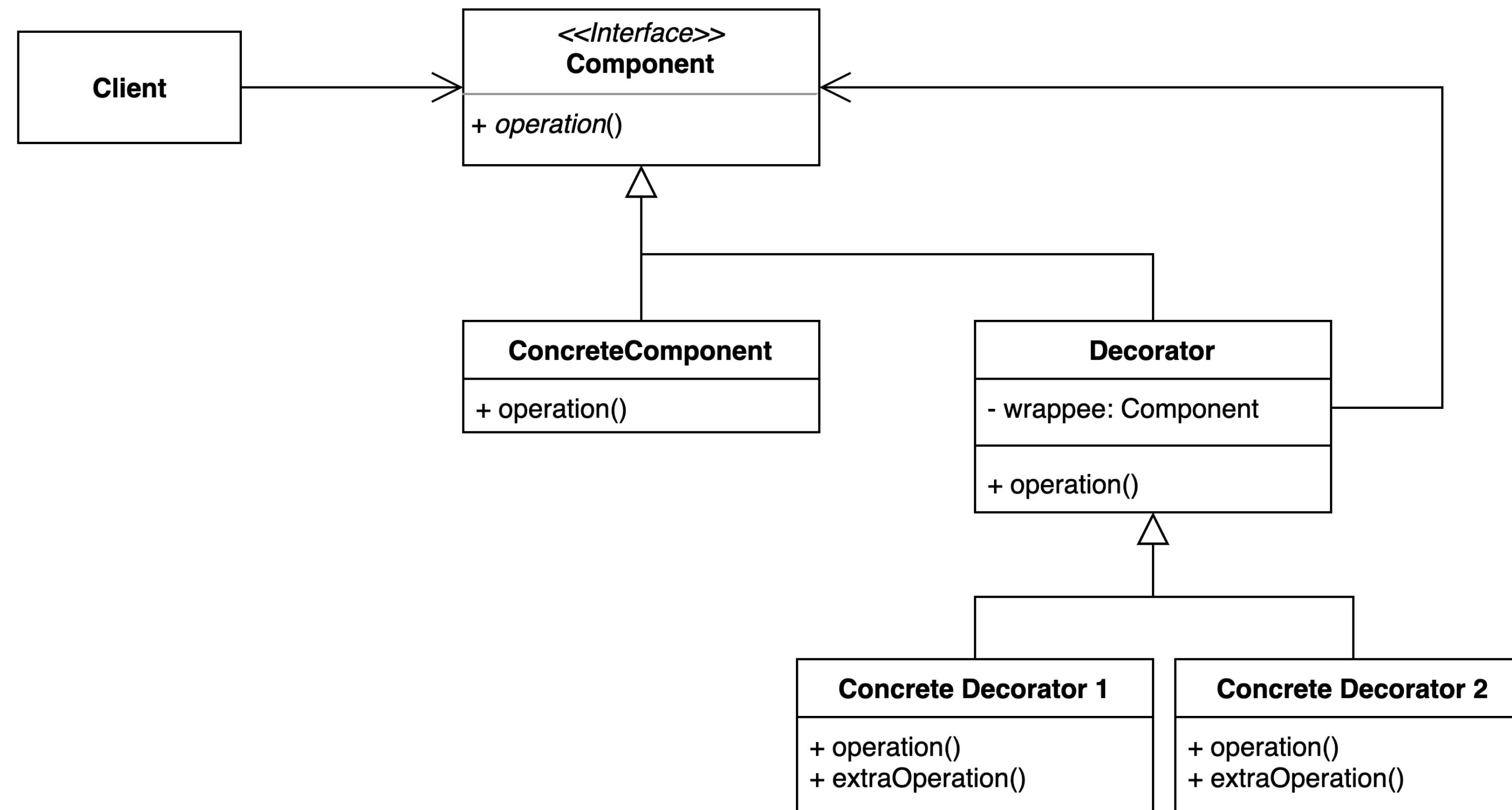
실무에서 어떻게 쓰이나?

- 자바
  - Swing 라이브러리
  - JSF (JavaServer Faces) 라이브러리

# 데코레이터 (Decorator) 패턴

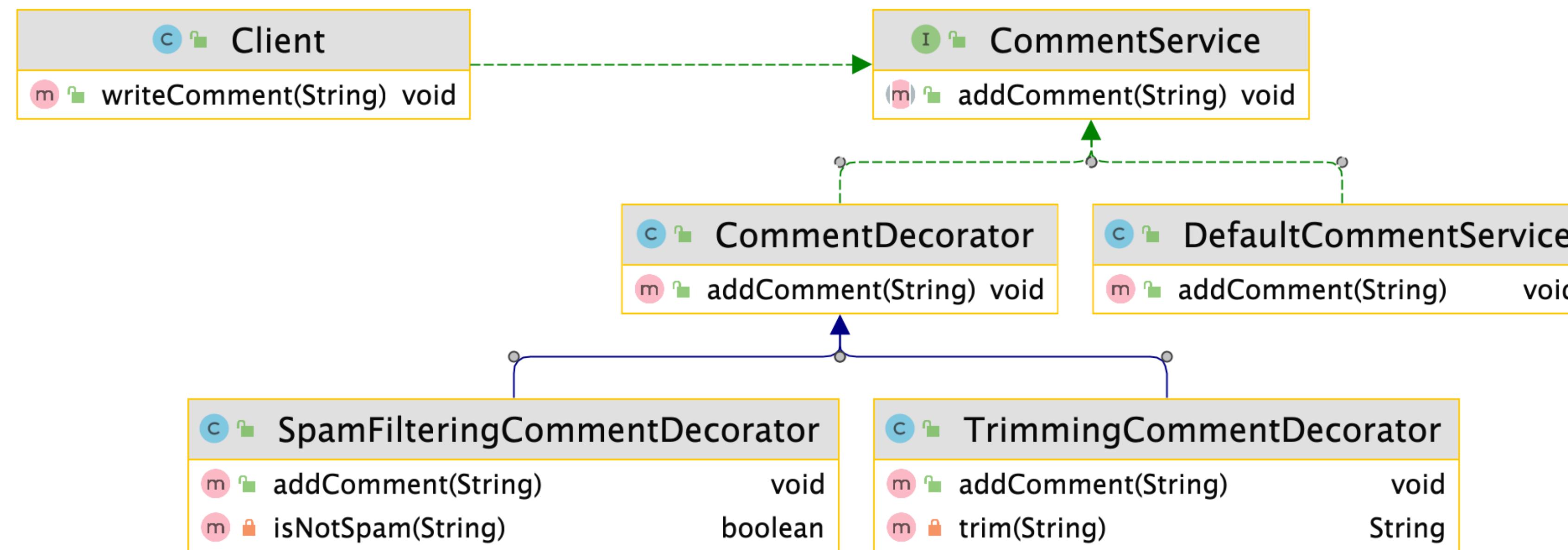
기존 코드를 변경하지 않고 부가 기능을 추가하는 패턴

- 상속이 아닌 위임을 사용해서 보다 유연하게(런타임에) 부가 기능을 추가하는 것도 가능하다.



# 데코레이터 (Decorator) 패턴

기존 코드를 변경하지 않고 부가 기능을 추가하는 패턴



# 데코레이터 (Decorator) 패턴

기존 코드를 변경하지 않고 부가 기능을 추가하는 패턴

- 장점
  - 새로운 클래스를 만들지 않고 기존 기능을 조합할 수 있다.
  - 컴파일 타임이 아닌 런타임에 동적으로 기능을 변경할 수 있다.
- 단점
  - 데코레이터를 조합하는 코드가 복잡할 수 있다.

# 데코레이터 (Decorator) 패턴

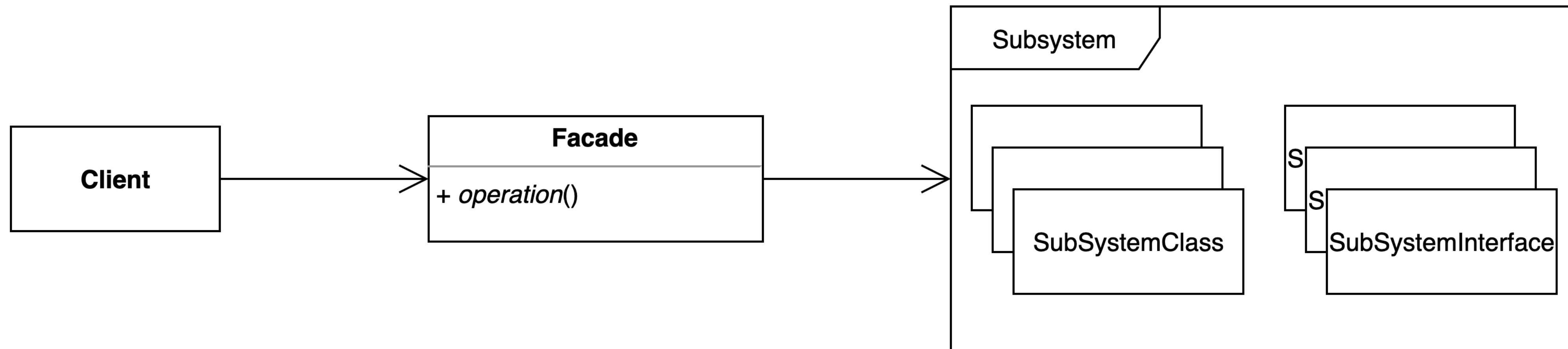
실무에서 어떻게 쓰이나?

- 자바
  - InputStream, OutputStream, Reader, Writer의 생성자를 활용한 랩퍼
  - java.util.Collections가 제공하는 메소드들 활용한 랩퍼
  - javax.servlet.http.HttpServletRequest/ResponseWrapper
- 스프링
  - ServerHttpRequestDecorator

# 퍼사드 (Facade) 패턴

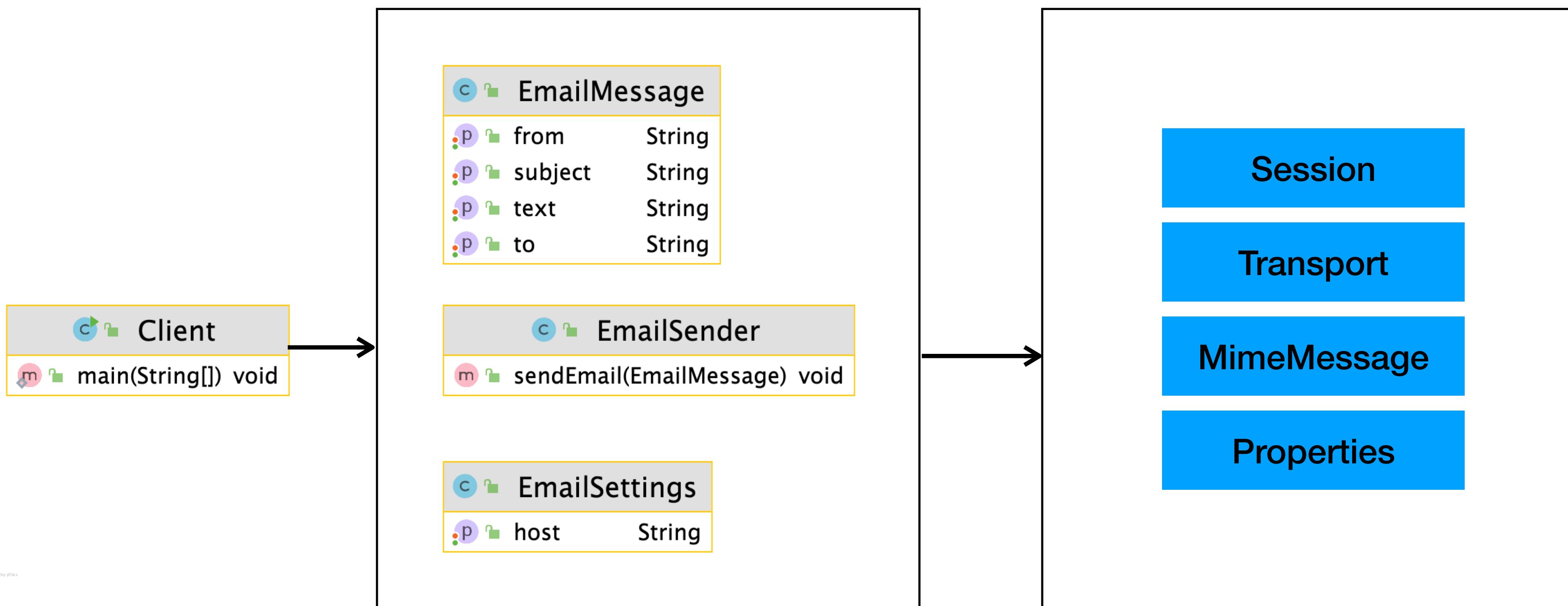
복잡한 서브 시스템 의존성을 최소화하는 방법.

- 클라이언트가 사용해야 하는 복잡한 서브 시스템 의존성을 간단한 인터페이스로 추상화 할 수 있다.



# 퍼사드 (Facade) 패턴

복잡한 서브 시스템 의존성을 최소화하는 방법.



# 퍼사드 (Facade) 패턴

복잡한 서브 시스템 의존성을 최소화하는 방법.

- 장점
  - 서브 시스템에 대한 의존성을 한곳으로 모을 수 있다.
- 단점
  - 퍼사드 클래스가 서브 시스템에 대한 모든 의존성을 가지게 된다.

# 퍼사드 (Facade) 패턴

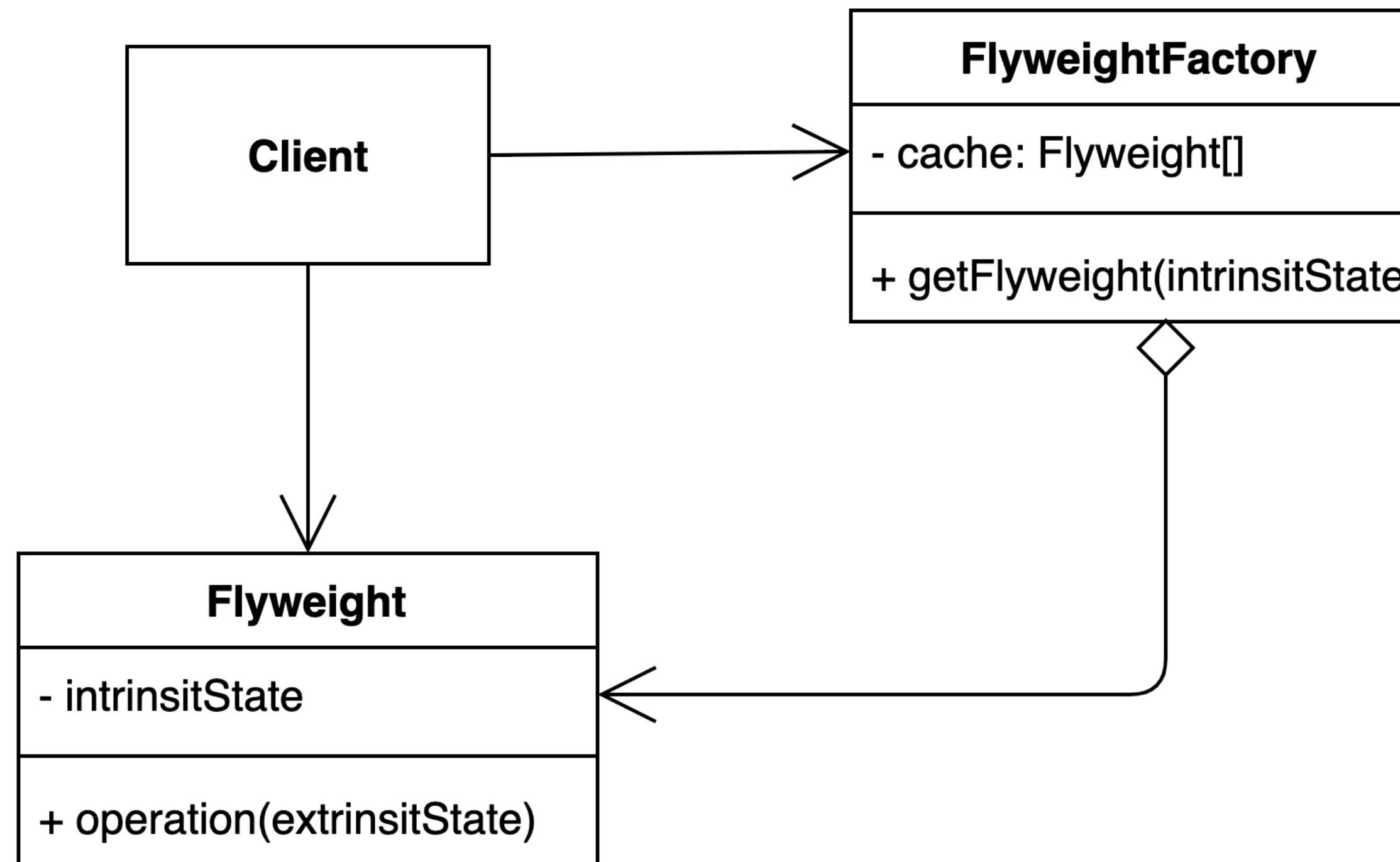
복잡한 서브 시스템 의존성을 최소화하는 방법.

- 스프링
  - Spring MVC
  - 스프링이 제공하는 대부분의 기술 독립적인 인터페이스와 그 구현체

# 플라이웨이트 (Flyweight) 패턴

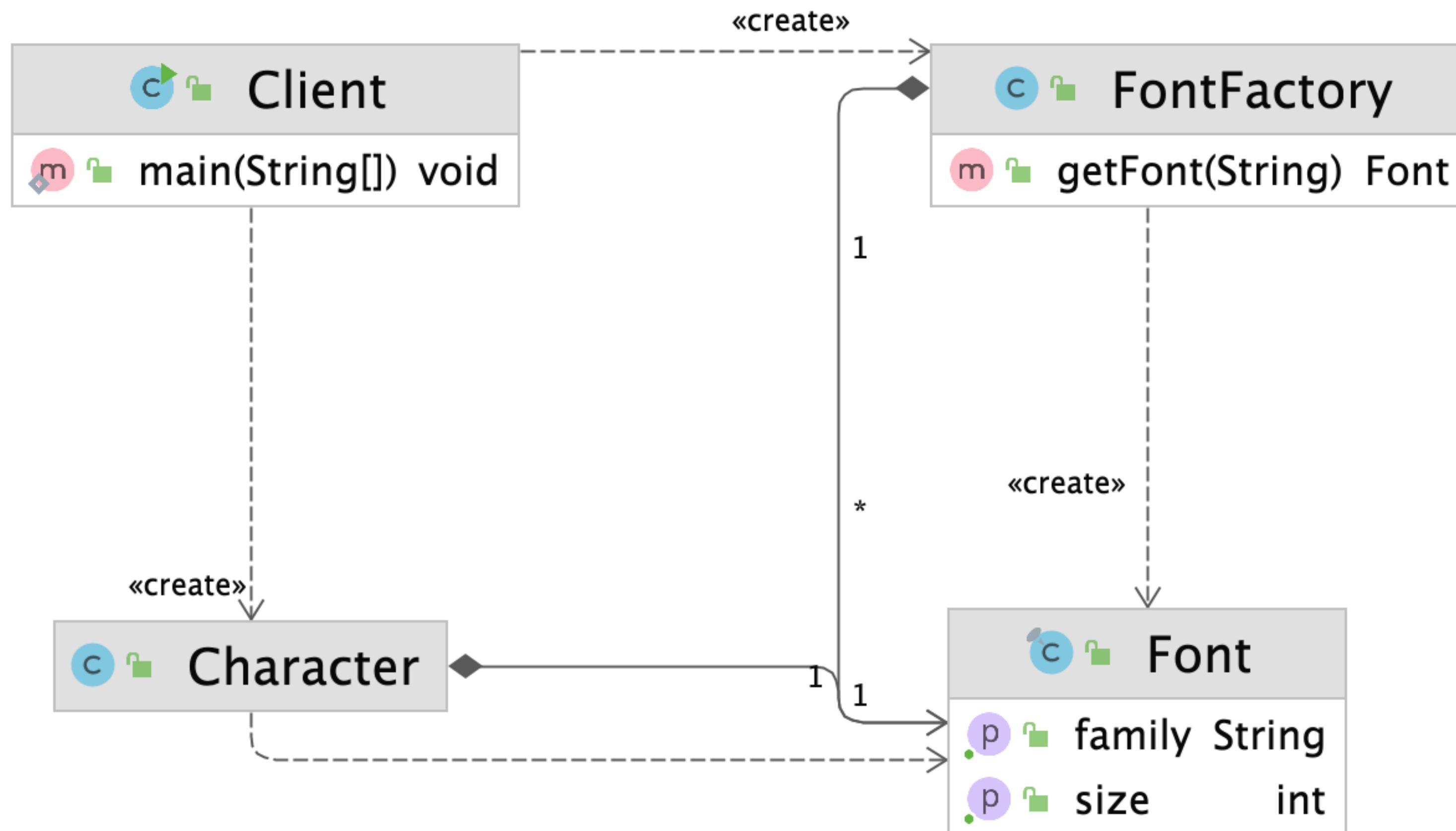
객체를 가볍게 만들어 메모리 사용을 줄이는 패턴.

- 자주 변하는 속성(또는 외적인 속성, extrinsic)과 변하지 않는 속성(또는 내적인 속성, intrinisit)을 분리하고 재사용하여 메모리 사용을 줄일 수 있다.



# 플라이웨이트 (Flyweight) 패턴

객체를 가볍게 만들어 메모리 사용을 줄이는 패턴.



# 플라이웨이트 (Flyweight) 패턴

객체를 가볍게 만들어 메모리 사용을 줄이는 패턴.

- 장점
  - 애플리케이션에서 사용하는 메모리를 줄일 수 있다.
- 단점
  - 코드의 복잡도가 증가한다.

# 플라이웨이트 (Flyweight) 패턴

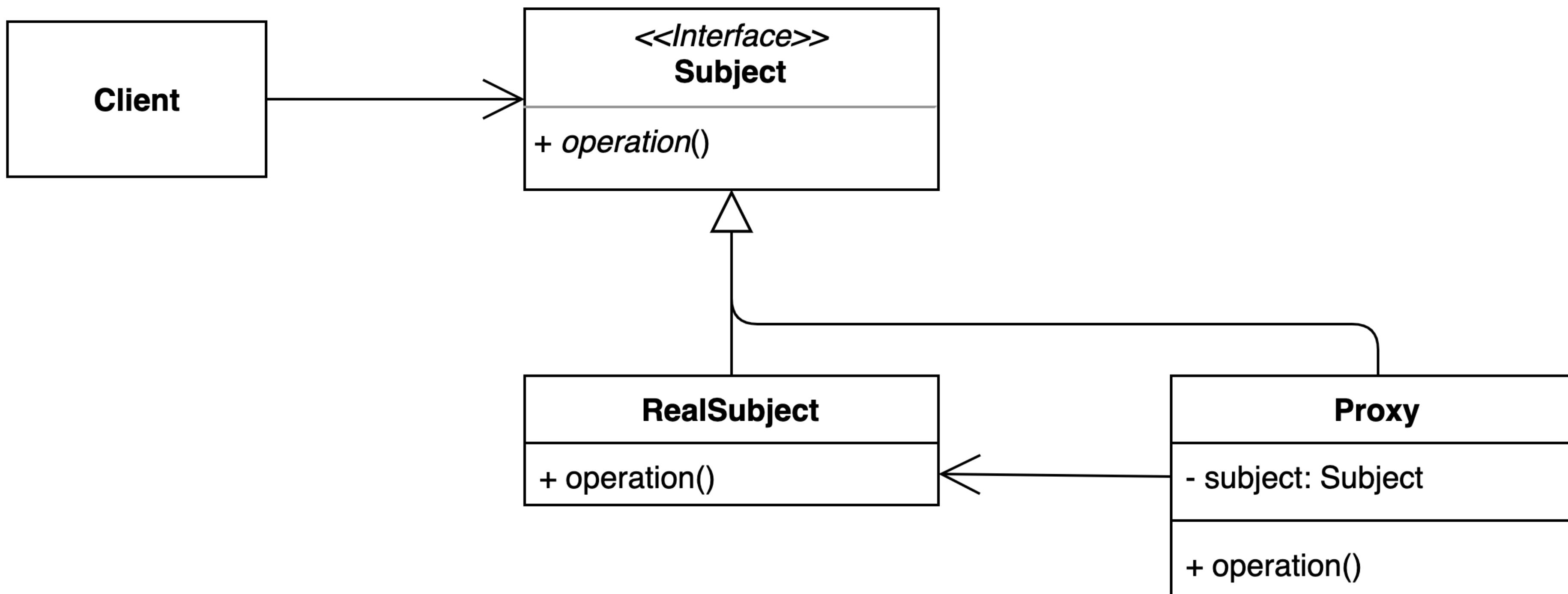
객체를 가볍게 만들어 메모리 사용을 줄이는 패턴.

- 자바
  - Integer.valueOf(int)
  - 캐시를 제공한다.
  - <https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#valueOf-int->

# 프록시 (Proxy) 패턴

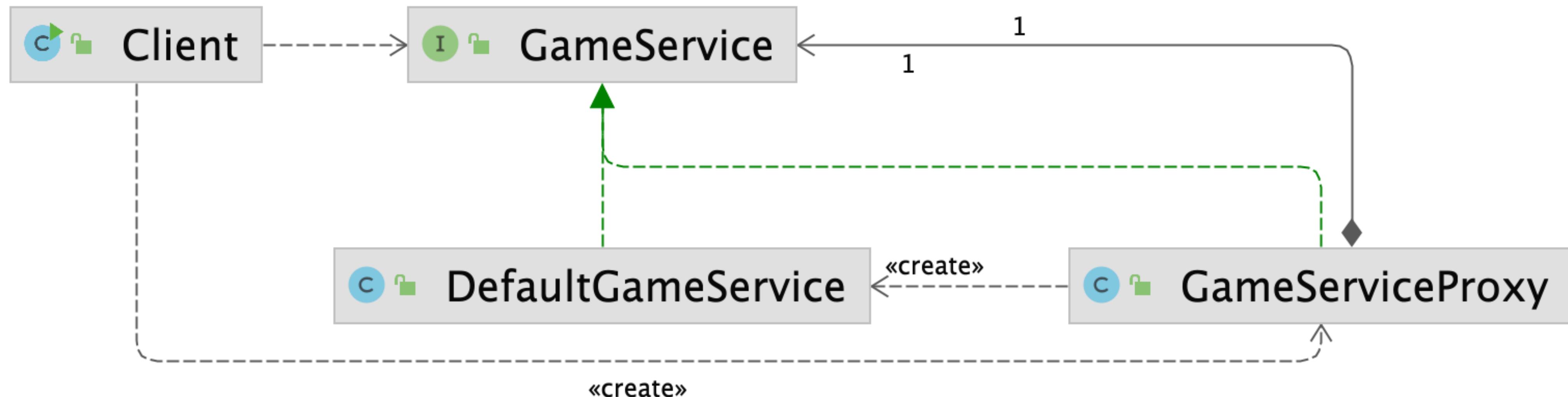
특정 객체에 대한 접근을 제어하거나 기능을 추가할 수 있는 패턴.

- 초기화 지연, 접근 제어, 로깅, 캐싱 등 다양하게 응용해 사용 할 수 있다.



# 프록시 (Proxy) 패턴

특정 객체에 대한 접근을 제어하거나 기능을 추가할 수 있는 패턴.



# 프록시 (Proxy) 패턴

특정 객체에 대한 접근을 제어하거나 기능을 추가할 수 있는 패턴.

- 장점
  - 기존 코드를 변경하지 않고 새로운 기능을 추가할 수 있다.
  - 기존 코드가 해야 하는 일만 유지할 수 있다.
  - 기능 추가 및 초기화 지연 등으로 다양하게 활용할 수 있다.
- 단점
  - 코드의 복잡도가 증가한다.

# 프록시 (Proxy) 패턴

특정 객체에 대한 접근을 제어하거나 기능을 추가할 수 있는 패턴.

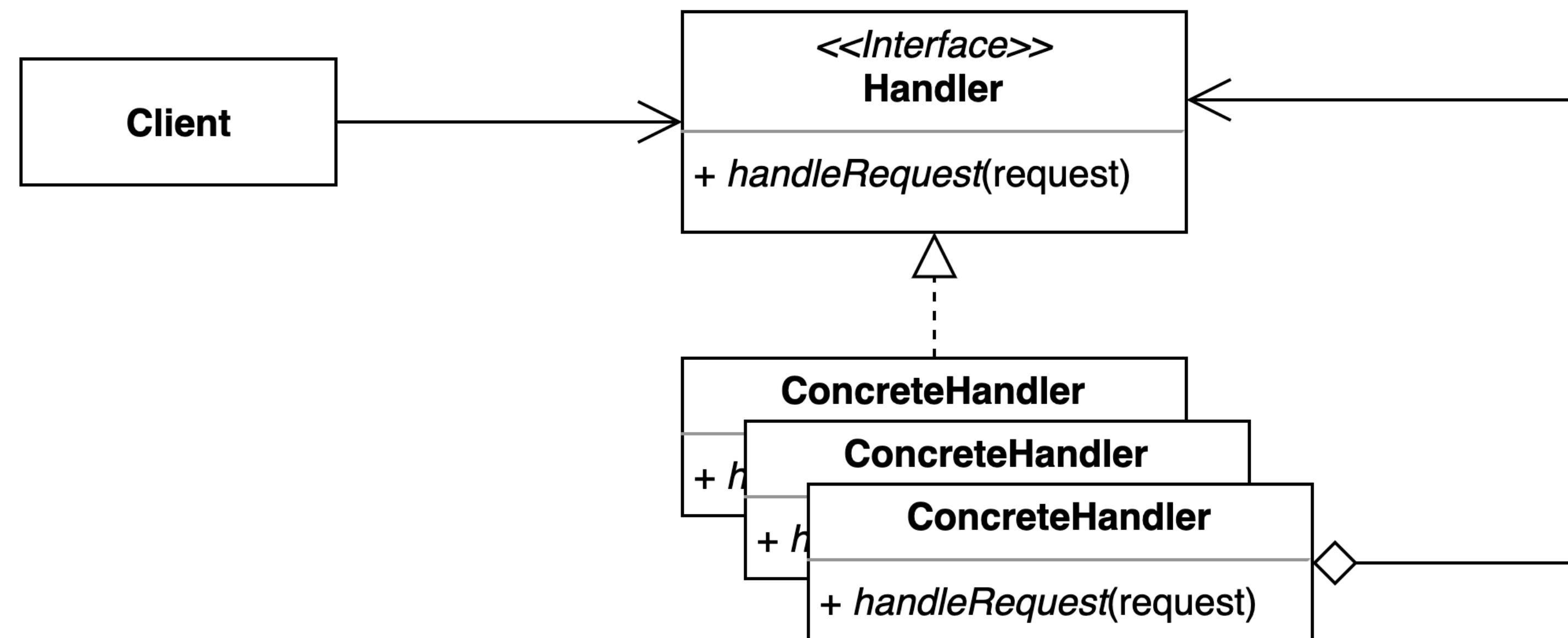
- 자바
  - 다이나믹 프록시, `java.lang.reflect.Proxy`
- 스프링
  - 스프링 AOP

# **행동 관련 패턴 (Behavioral Patterns)**

# 책임 연쇄 패턴 (Chain-of-Responsibility) 패턴

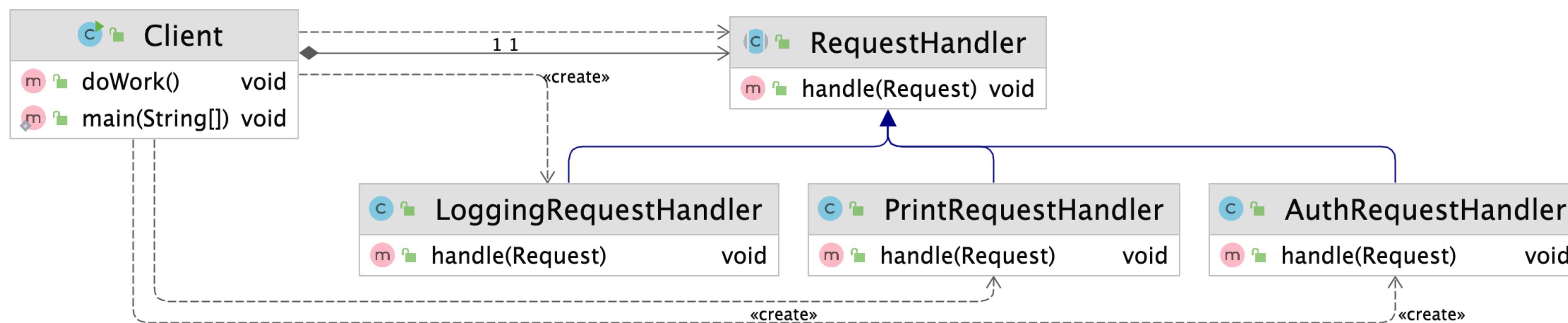
요청을 보내는 쪽(sender)과 요청을 처리하는 쪽(receiver)의 분리하는 패턴

- 핸들러 체인을 사용해서 요청을 처리한다.



# 책임 연쇄 패턴 (Chain-of-responsibility) 패턴

요청을 보내는 쪽(sender)과 요청을 처리하는 쪽(receiver)의 분리하는 패턴



# 책임 연쇄 패턴 (Chain-of-responsibility) 패턴

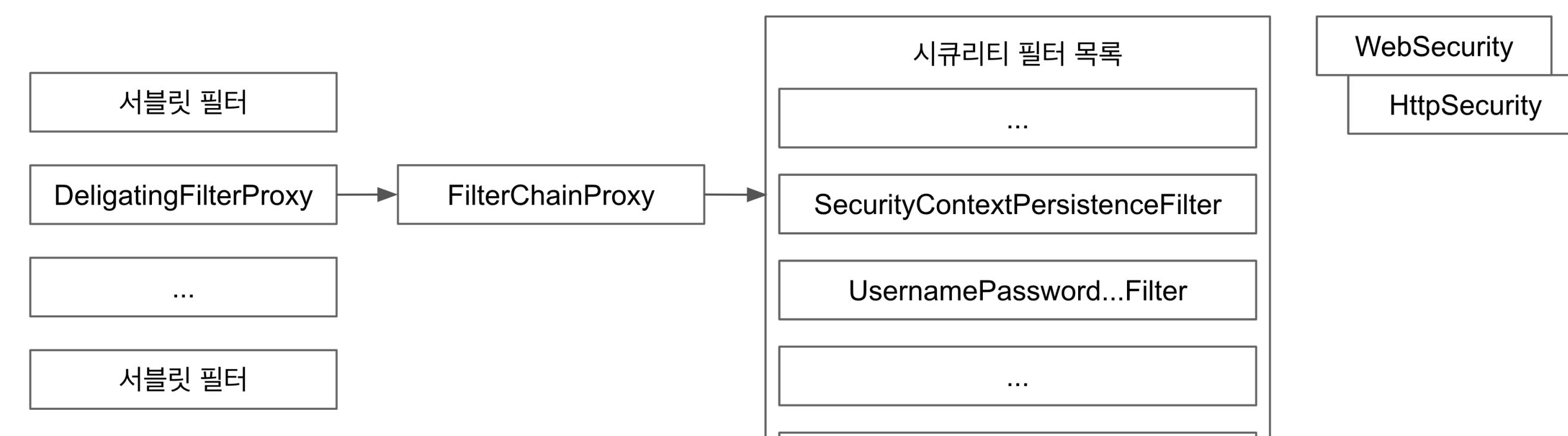
요청을 보내는 쪽(sender)과 요청을 처리하는 쪽(receiver)의 분리하는 패턴

- 장점
  - 클라이언트 코드를 변경하지 않고 새로운 핸들러를 체인에 추가할 수 있다.
  - 각각의 체인은 자신이 해야하는 일만 한다.
  - 체인을 다양한 방법으로 구성할 수 있다.
- 단점
  - 디버깅이 조금 어렵다.

# 책임 연쇄 패턴 (Chain-of-responsibility) 패턴

요청을 보내는 쪽(sender)과 요청을 처리하는 쪽(receiver)의 분리하는 패턴

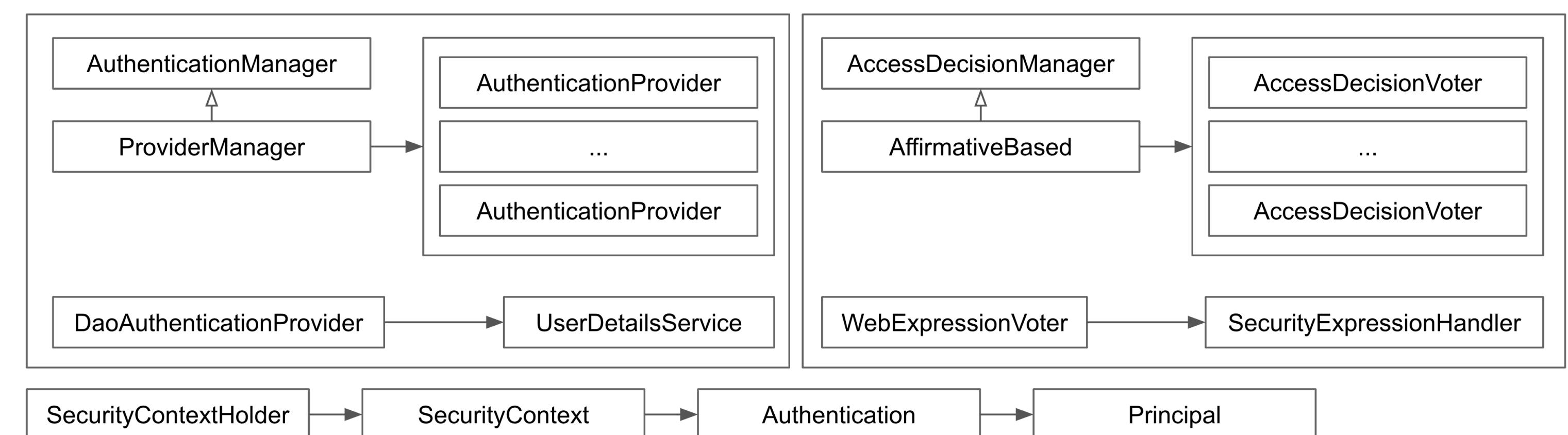
- 자바



- 서블릿 필터

- 스프링

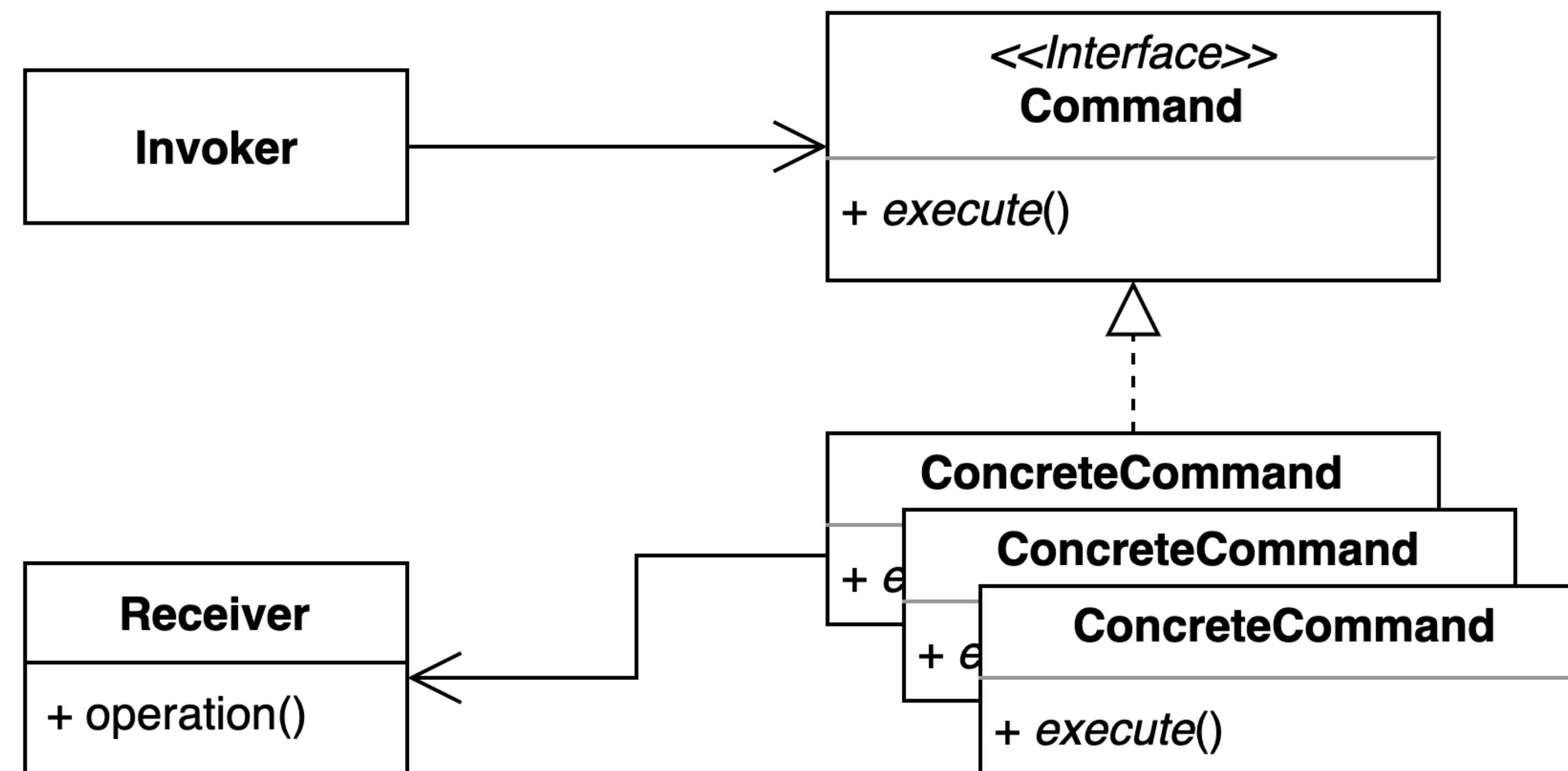
- 스프링 시큐리티 필터



# 커맨드 (Command) 패턴

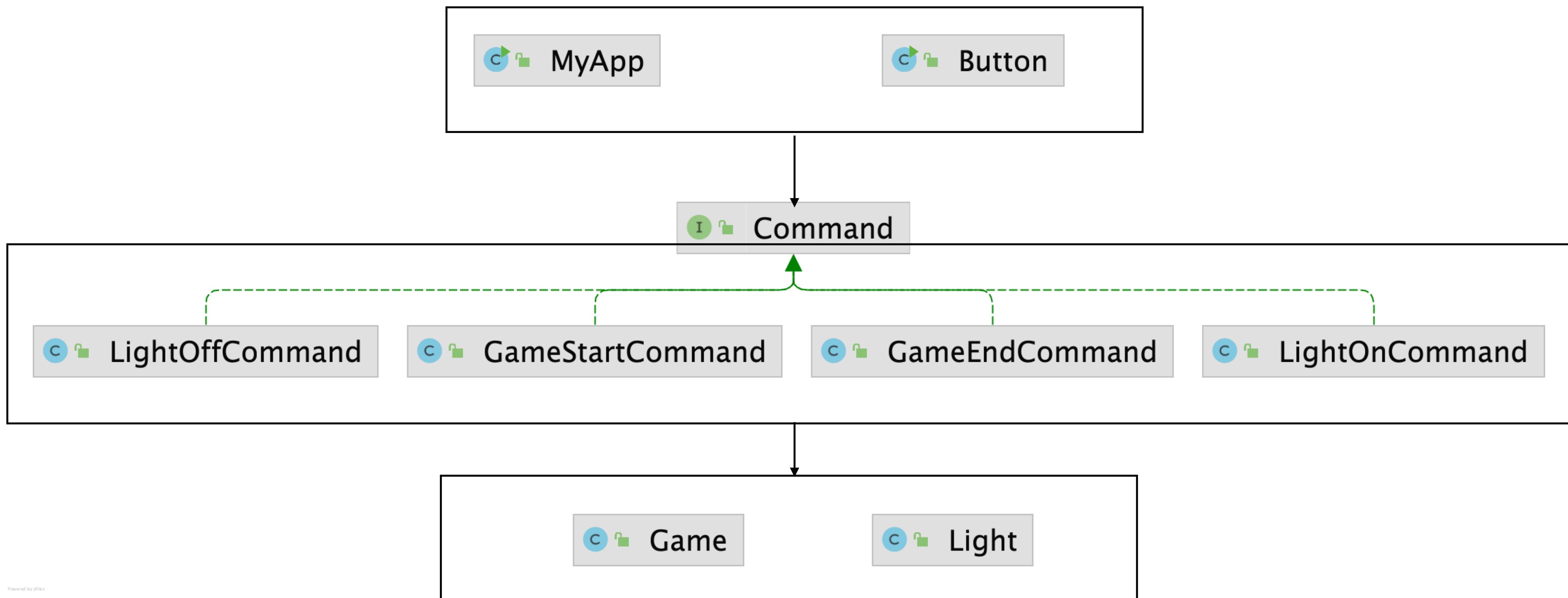
요청을 캡슐화 하여 호출자(invoker)와 수신자(receiver)를 분리하는 패턴.

- 요청을 처리하는 방법이 바뀌더라도, 호출자의 코드는 변경되지 않는다.



# 커맨드 (Command) 패턴

요청을 캡슐화 하여 호출자(invoker)와 수신자(receiver)를 분리하는 패턴.



# 커맨드 (Command) 패턴

요청을 캡슐화 하여 호출자(invoker)와 수신자(receiver)를 분리하는 패턴.

- 장점
  - 기존 코드를 변경하지 않고 새로운 커맨드를 만들 수 있다.
  - 수신자의 코드가 변경되어도 호출자의 코드는 변경되지 않는다.
  - 커맨드 객체를 로깅, DB에 저장, 네트워크로 전송 하는 등 다양한 방법으로 활용할 수도 있다.
- 단점
  - 코드가 복잡하고 클래스가 많아진다.

# 커맨드 (Command) 패턴

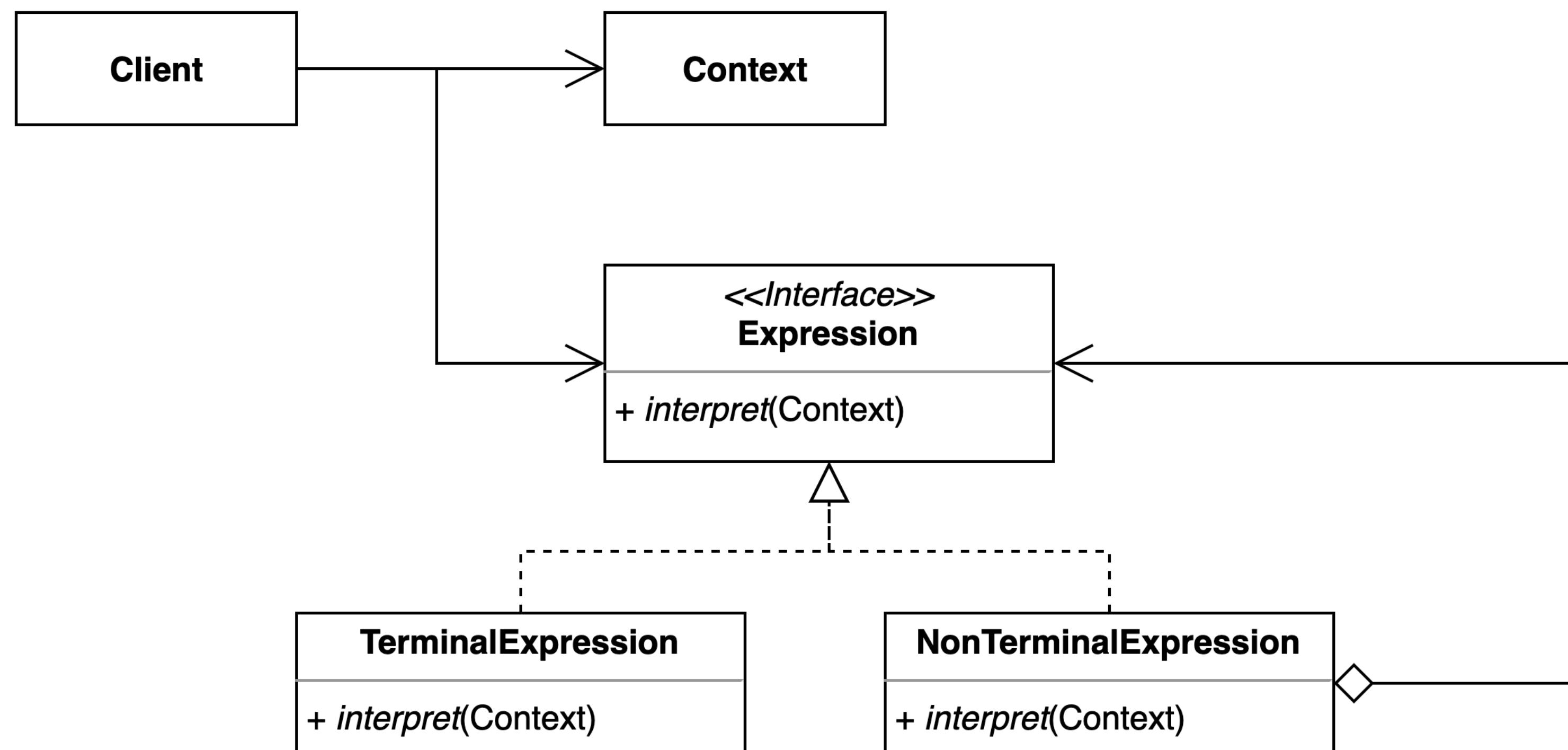
요청을 캡슐화 하여 호출자(invoker)와 수신자(receiver)를 분리하는 패턴.

- 자바
  - Runnable
  - 람다
  - 메소드 레퍼런스
- 스프링
  - SimpleJdbcInsert
  - SimpleJdbcCall

# 인터프리터 (Interpreter) 패턴

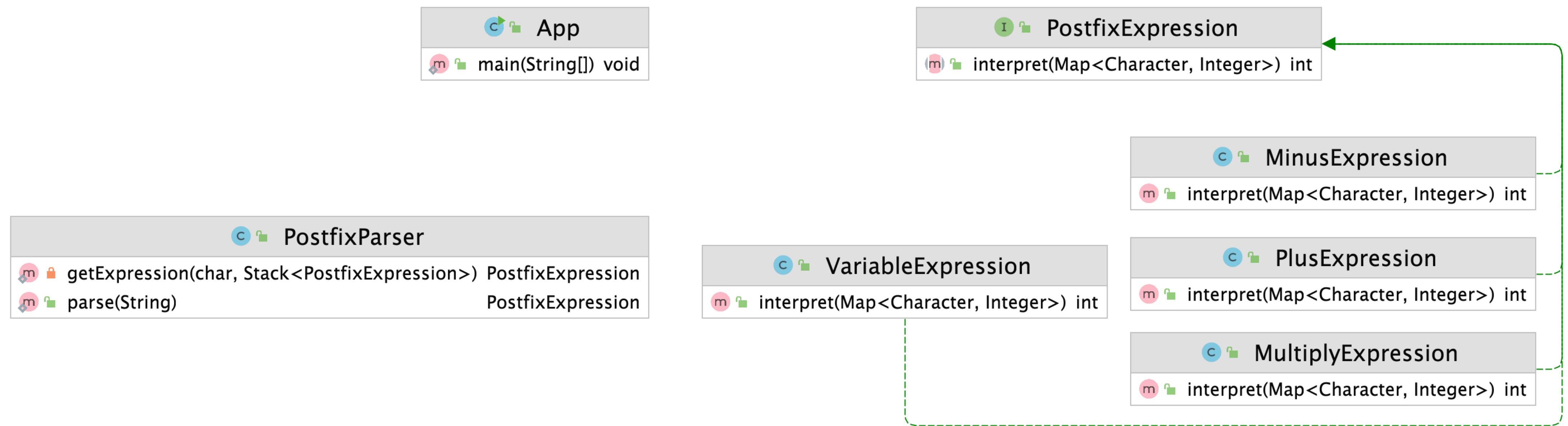
자주 등장하는 문제를 간단한 언어로 정의하고 재사용하는 패턴.

- 반복되는 문제 패턴을 언어 또는 문법으로 정의하고 확장할 수 있다.



# 인터프리터 (Interpreter) 패턴

요청을 캡슐화하여 호출자(invoker)와 수신자(receiver)를 분리하는 패턴.



# 인터프리터 (Interpreter) 패턴

요청을 캡슐화 하여 호출자(invoker)와 수신자(receiver)를 분리하는 패턴.

- 장점
  - 자주 등장하는 문제 패턴을 언어와 문법으로 정의할 수 있다.
  - 기존 코드를 변경하지 않고 새로운 Expression을 추가할 수 있다.
- 단점
  - 복잡한 문법을 표현하려면 Expression과 Parser가 복잡해진다.

# 인터프리터 (Interpreter) 패턴

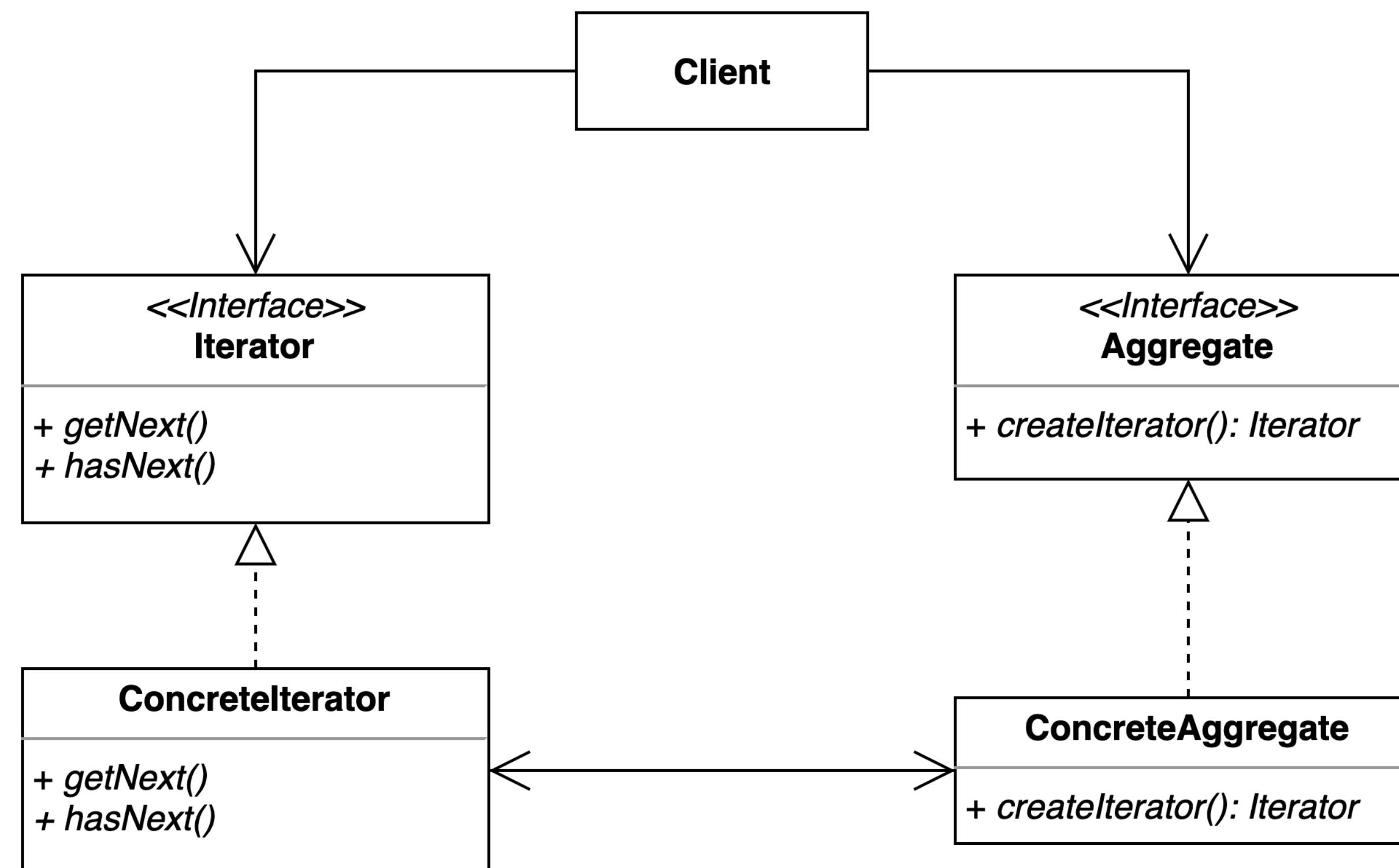
요청을 캡슐화하여 호출자(invoker)와 수신자(receiver)를 분리하는 패턴.

- 자바
  - 자바 컴파일러
  - 정규 표현식
- 스프링
  - SpEL (스프링 Expression Language)

# 이터레이터 (Iterator) 패턴

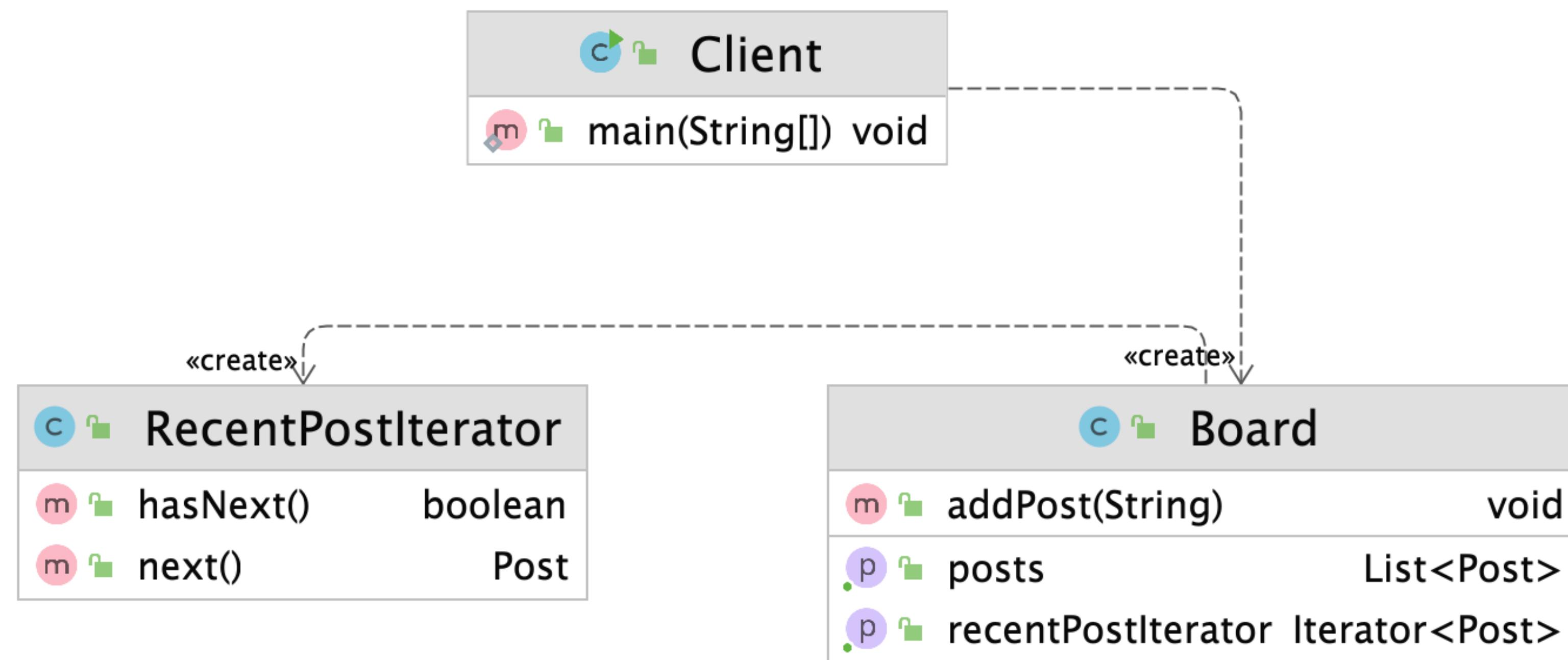
집합 객체 내부 구조를 노출시키지 않고 순회 하는 방법을 제공하는 패턴 .

- 집합 객체를 순회하는 클라이언트 코드를 변경하지 않고 다양한 순회 방법을 제공할 수 있다.



# 이터레이터 (Iterator) 패턴

집합 객체 내부 구조를 노출시키지 않고 순회 하는 방법을 제공하는 패턴 .



# 이터레이터 (Iterator) 패턴

집합 객체 내부 구조를 노출시키지 않고 순회 하는 방법을 제공하는 패턴.

- 장점
  - 집합 객체가 가지고 있는 객체들에 손쉽게 접근할 수 있다.
  - 일관된 인터페이스를 사용해 여러 형태의 집합 구조를 순회할 수 있다.
- 단점
  - 클래스가 늘어나고 복잡도가 증가한다.

# 이터레이터 (Iterator) 패턴

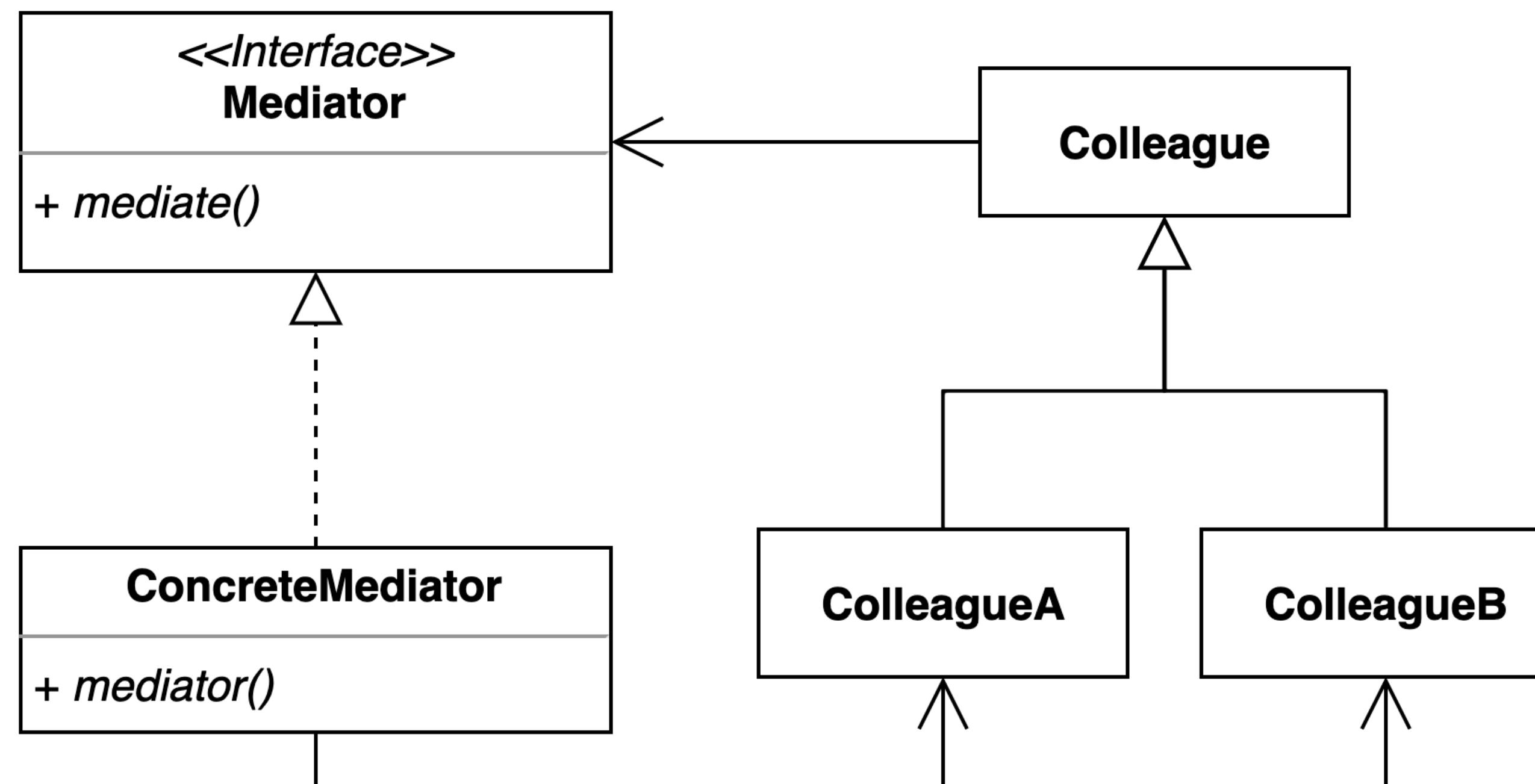
집합 객체 내부 구조를 노출시키지 않고 순회 하는 방법을 제공하는 패턴 .

- 자바
  - `java.util Enumeration`과 `java.util Iterator`
  - Java StAX (Streaming API for XML)의 Iterator 기반 API
    - `XmlEventReader`, `XmlEventWriter`
- 스프링
  - `CompositeIterator`

# 중재자 (Mediator) 패턴

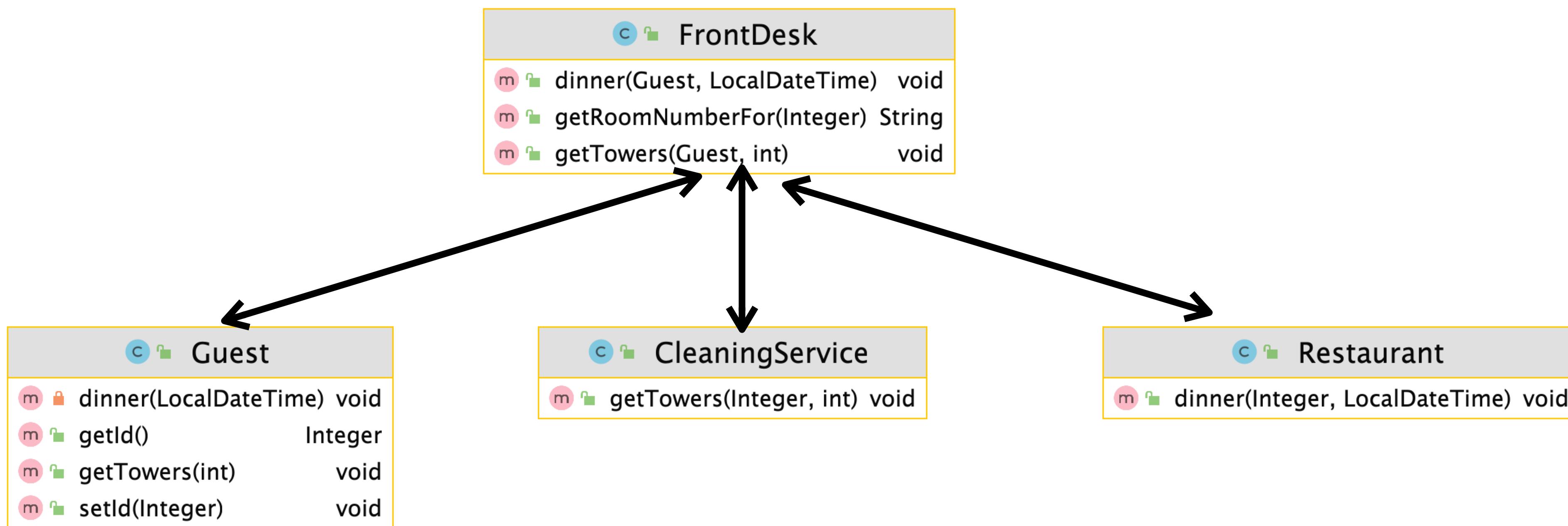
여러 객체들이 소통하는 방법을 캡슐화하는 패턴.

- 여러 컴포넌트간의 결합도를 중재자를 통해 낮출 수 있다.



# 중재자 (Mediator) 패턴

여러 객체들이 소통하는 방법을 캡슐화하는 패턴.



# 중재자 (Mediator) 패턴

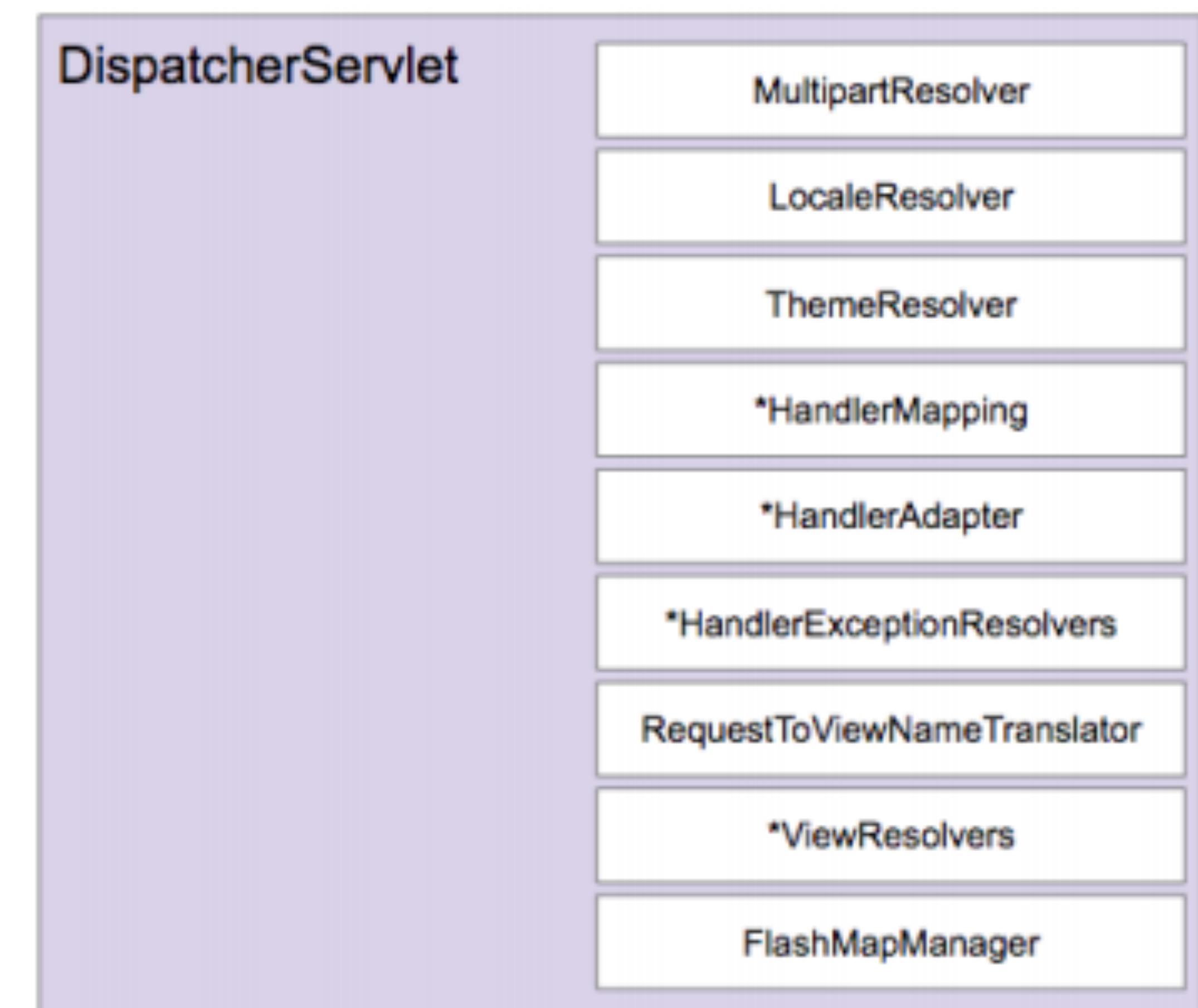
여러 객체들이 소통하는 방법을 캡슐화하는 패턴.

- 장점
  - 컴포넌트 코드를 변경하지 않고 새로운 중재자를 만들어 사용할 수 있다.
  - 각각의 컴포넌트 코드를 보다 간결하게 유지할 수 있다.
- 단점
  - 중재자 역할을 하는 클래스의 복잡도와 결합도가 증가한다.

# 중재자 (Mediator) 패턴

여러 객체들이 소통하는 방법을 캡슐화하는 패턴.

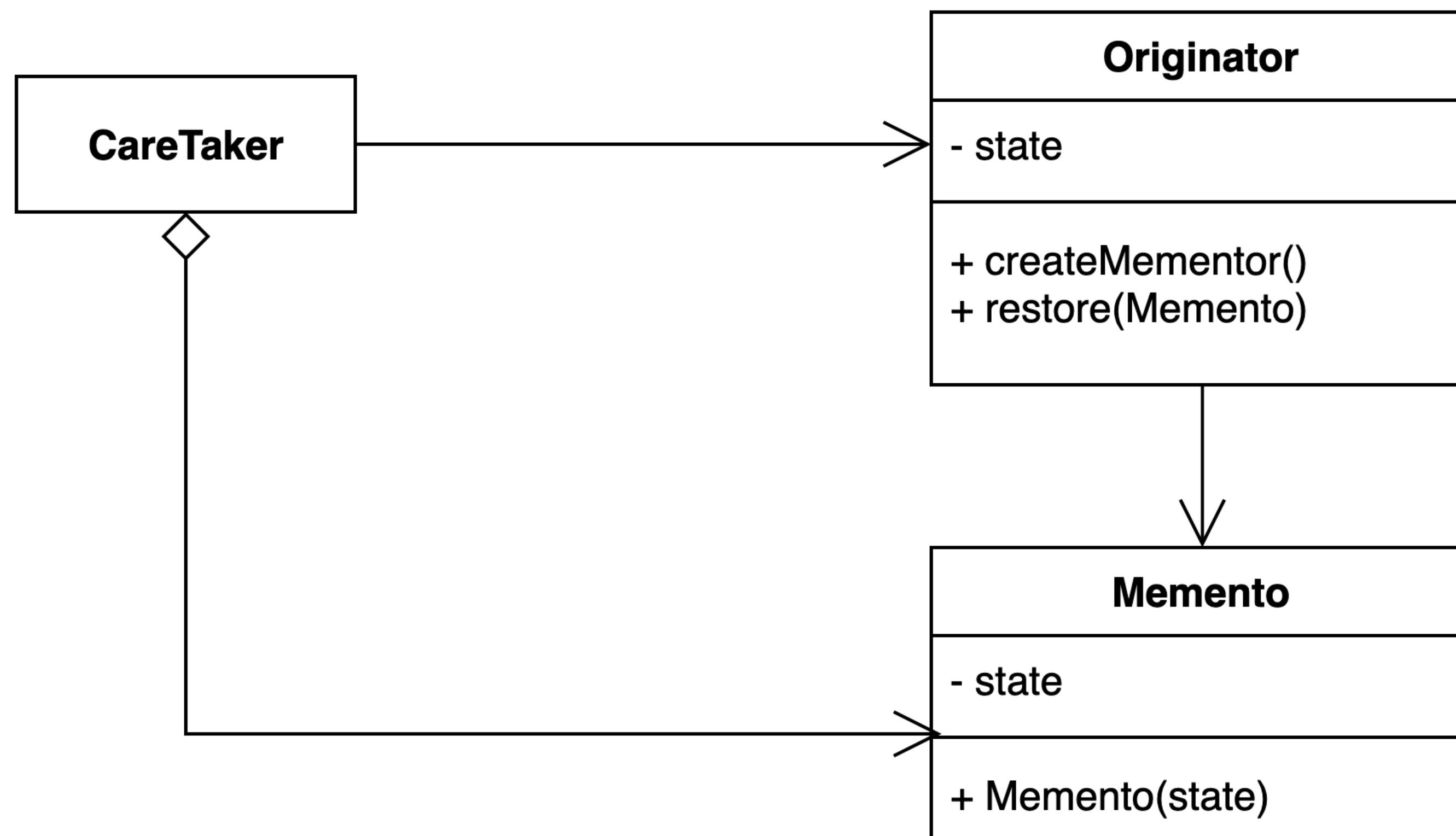
- 자바
  - ExecutorService
  - Executor
- 스프링
  - DispatcherServlet



# 메멘토 (Memento) 패턴

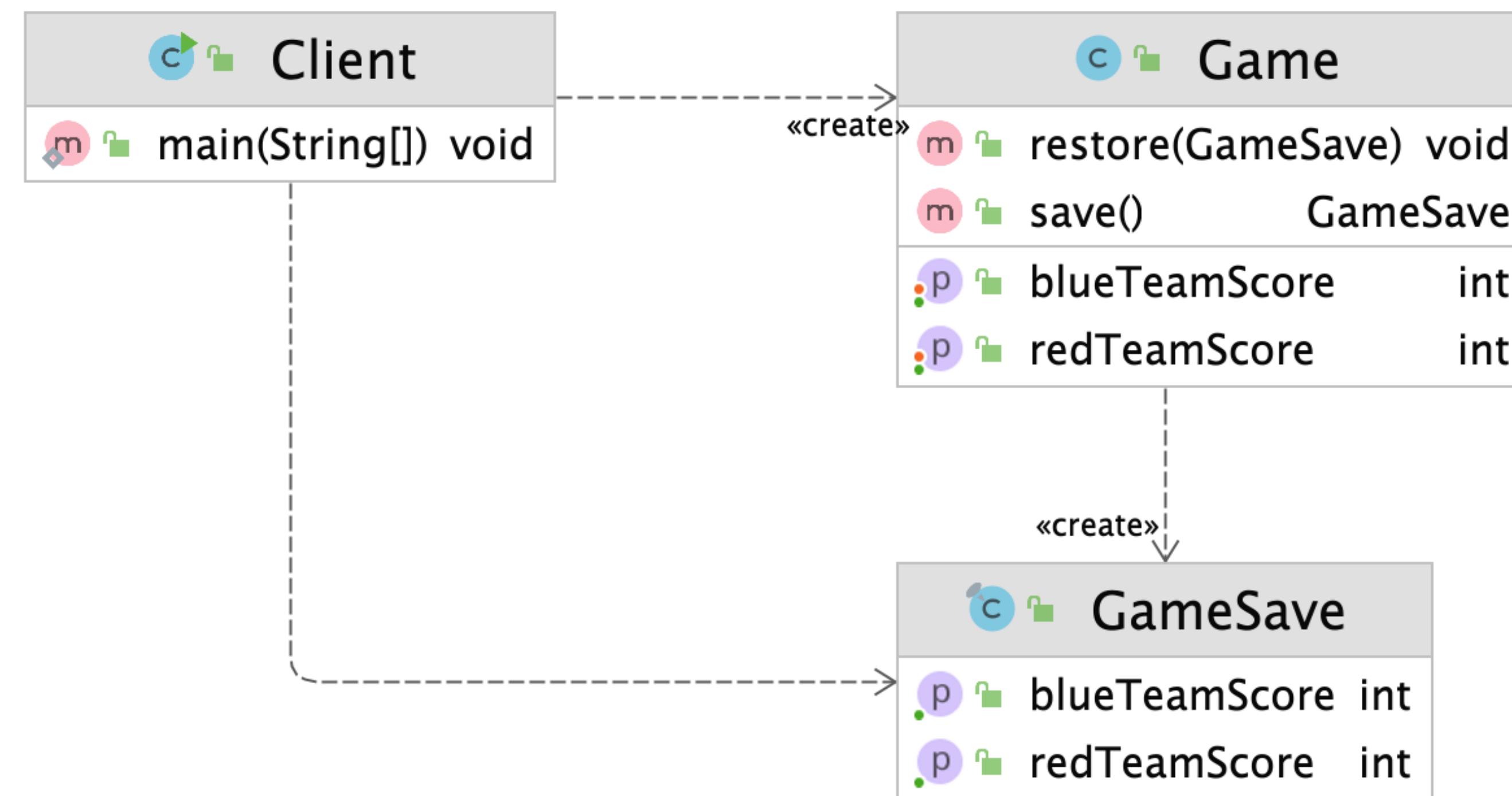
캡슐화를 유지하면서 객체 내부 상태를 외부에 저장하는 방법.

- 객체 상태를 외부에 저장했다가 해당 상태로 다시 복구할 수 있다.



# 메멘토 (Memento) 패턴

캡슐화를 유지하면서 객체 내부 상태를 외부에 저장하는 방법.



# 메멘토 (Memento) 패턴

캡슐화를 유지하면서 객체 내부 상태를 외부에 저장하는 방법.

- 장점
  - 캡슐화를 지키면서 상태 객체 상태 스냅샷을 만들 수 있다.
  - 객체 상태 저장하고 또는 복원하는 역할을 CareTaker에게 위임할 수 있다.
  - 객체 상태가 바뀌어도 클라이언트 코드는 변경되지 않는다.
- 단점
  - 많은 정보를 저장하는 Mementor를 자주 생성하는 경우 메모리 사용량에 많은 영향을 줄 수 있다.

# 메멘토 (Memento) 패턴

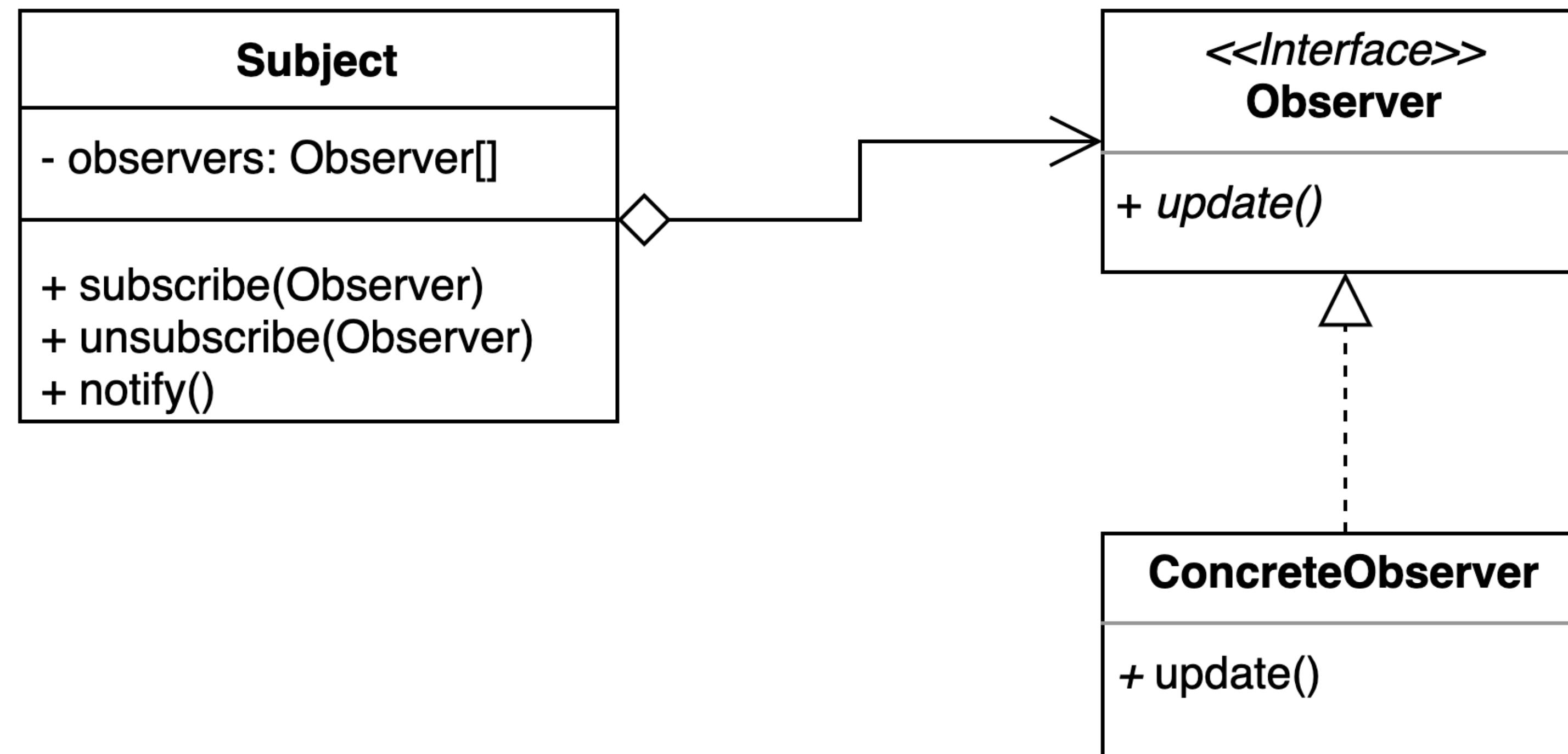
캡슐화를 유지하면서 객체 내부 상태를 외부에 저장하는 방법.

- 자바
  - 객체 직렬화, `java.io.Serializable`
  - `java.util.Date`

# 옵저버 (Observer) 패턴

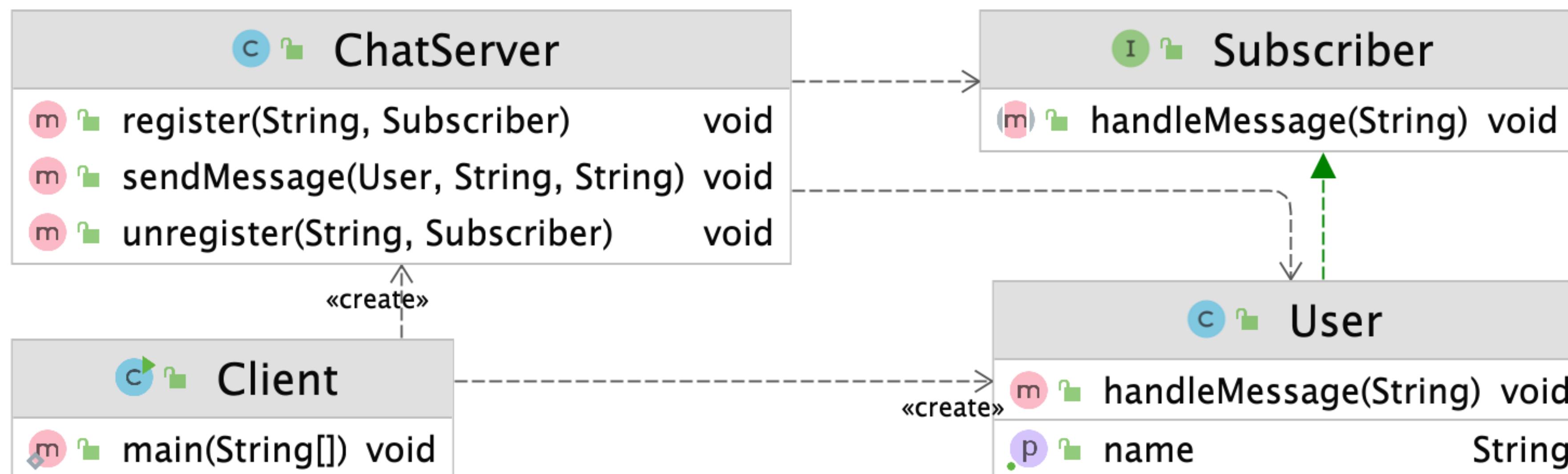
다수의 객체가 특정 객체 상태 변화를 감지하고 알림을 받는 패턴.

- 발행(publish)-구독(subscribe) 패턴을 구현할 수 있다.



# 옵저버 (Observer) 패턴

다수의 객체가 특정 객체 상태 변화를 감지하고 알림을 받는 패턴.



# 옵저버 (Observer) 패턴

다수의 객체가 특정 객체 상태 변화를 감지하고 알림을 받는 패턴.

- 장점
  - 상태를 변경하는 객체(publisher)와 변경을 감지하는 객체(subscriber)의 관계를 느슨하게 유지할 수 있다.
  - Subject의 상태 변경을 주기적으로 조회하지 않고 자동으로 감지할 수 있다.
  - 런타임에 옵저버를 추가하거나 제거할 수 있다.
- 단점
  - 복잡도가 증가한다.
  - 다수의 Observer 객체를 등록 이후 해지 않는다면 memory leak이 발생할 수도 있다.

# 옵저버 (Observer) 패턴

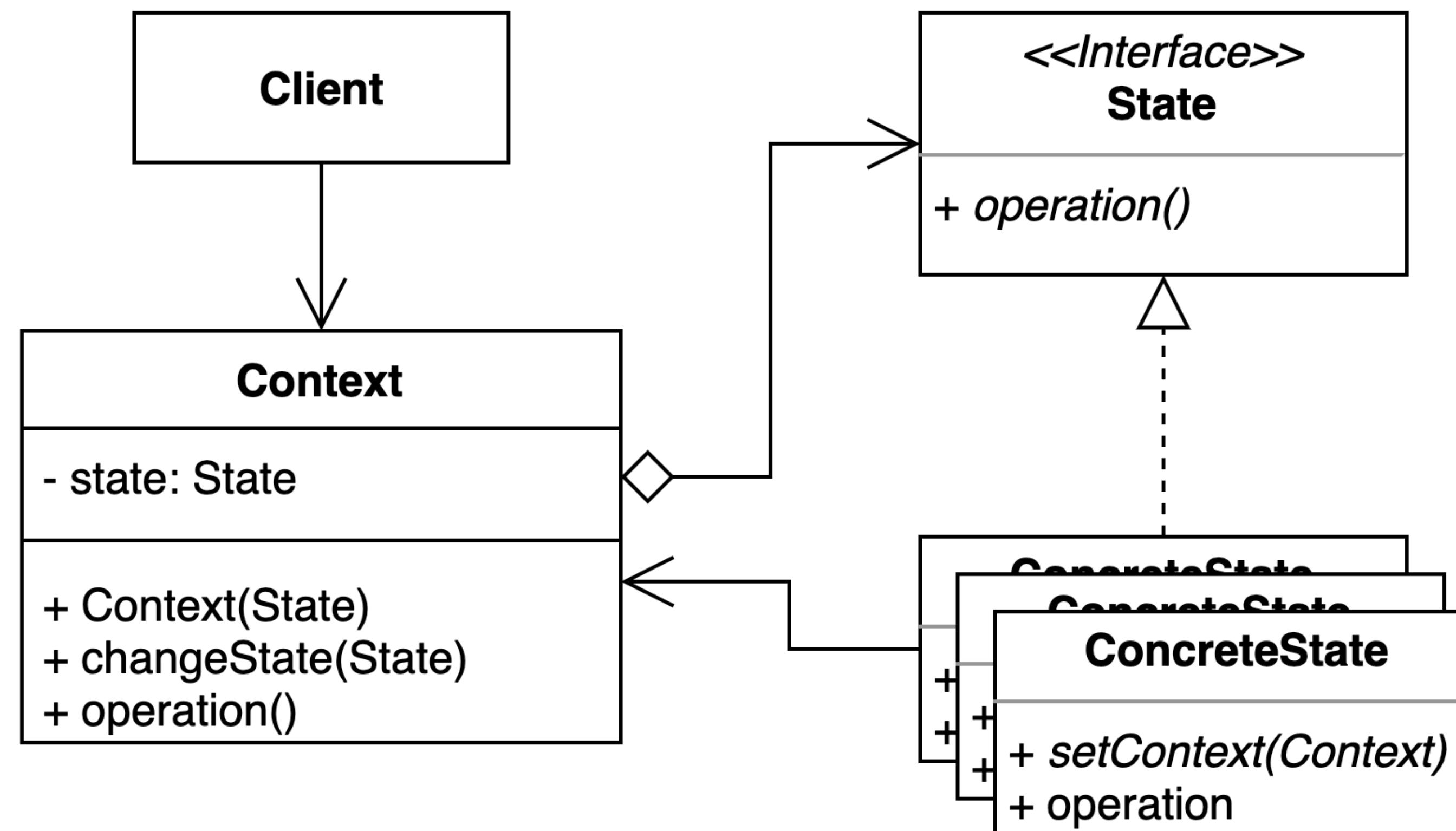
다수의 객체가 특정 객체 상태 변화를 감지하고 알림을 받는 패턴.

- 자바
  - Observable과 Observer (자바 9부터 deprecated)
  - 자바 9 이후 부터는
    - PropertyChangeListener, PropertyChangeEvent
    - Flow API
  - SAX (Simple API for XML) 라이브러리
- 스프링
  - ApplicationContext와 ApplicationEvent

# 상태 (State) 패턴

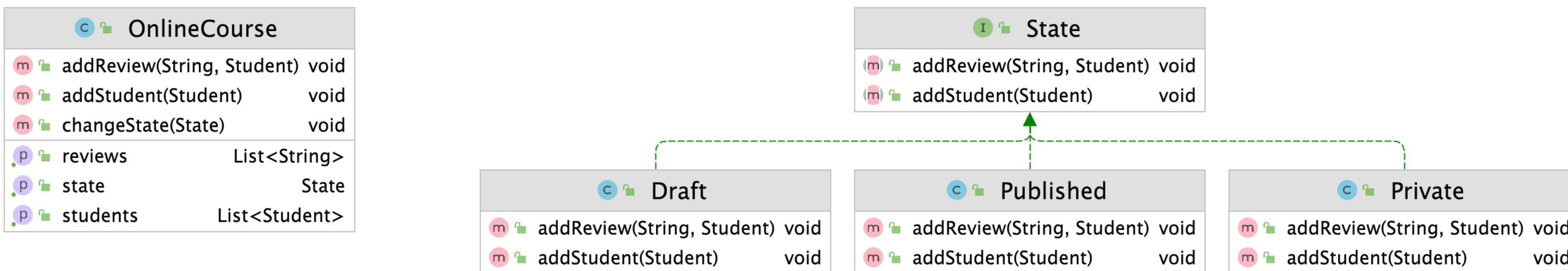
객체 내부 상태 변경에 따라 객체의 행동이 달라지는 패턴.

- 상태에 특화된 행동들을 분리해 낼 수 있으며, 새로운 행동을 추가하더라도 다른 행동에 영향을 주지 않는다.



# 상태 (State) 패턴

객체 내부 상태 변경에 따라 객체의 행동이 달라지는 패턴.



# 상태 (State) 패턴

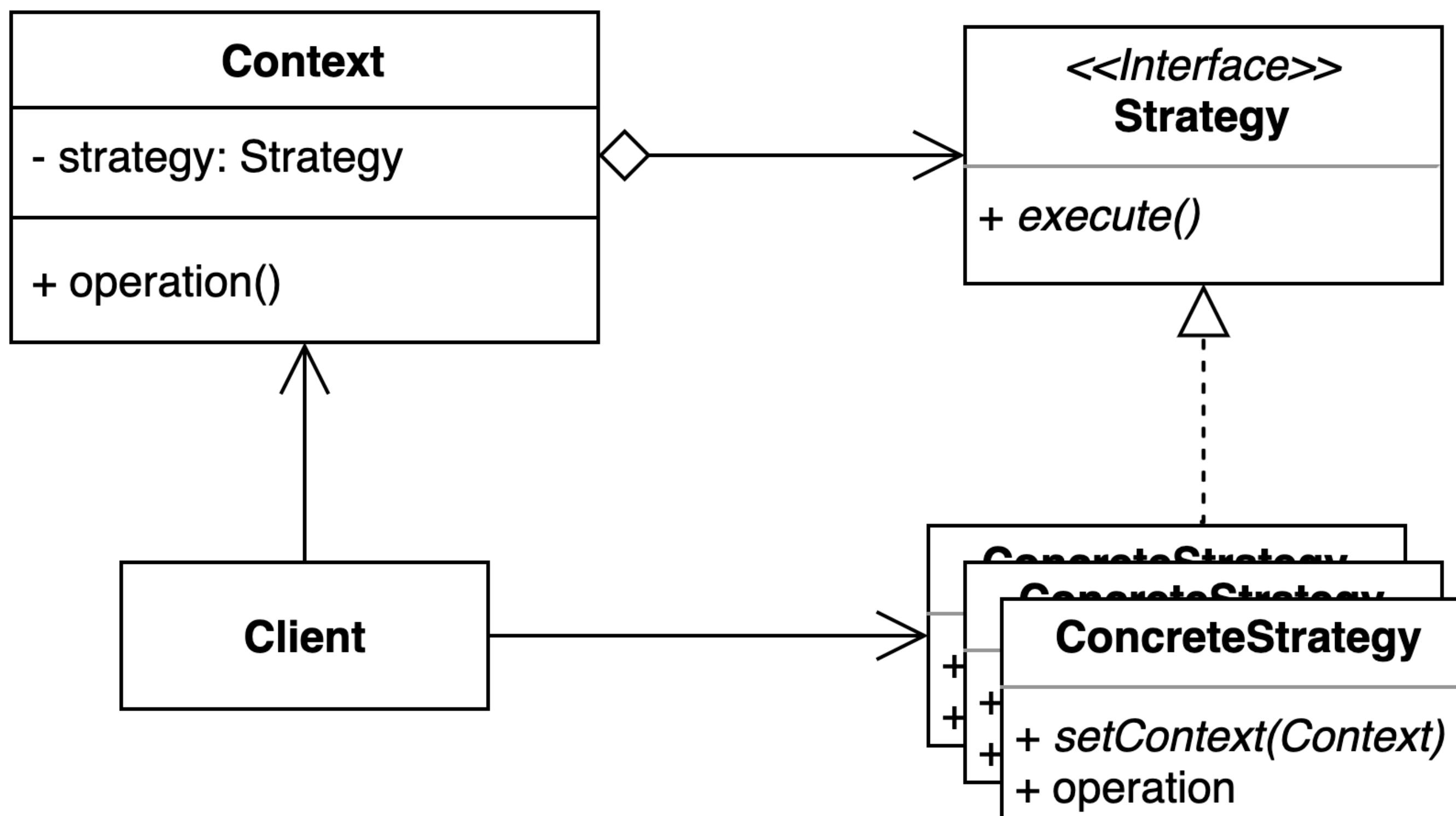
객체 내부 상태 변경에 따라 객체의 행동이 달라지는 패턴.

- 장점
  - 상태에 따른 동작을 개별 클래스로 옮겨서 관리할 수 있다.
  - 기존의 특정 상태에 따른 동작을 변경하지 않고 새로운 상태에 다른 동작을 추가할 수 있다.
  - 코드 복잡도를 줄일 수 있다.
- 단점
  - 복잡도가 증가한다.

# 전략 (Strategy) 패턴

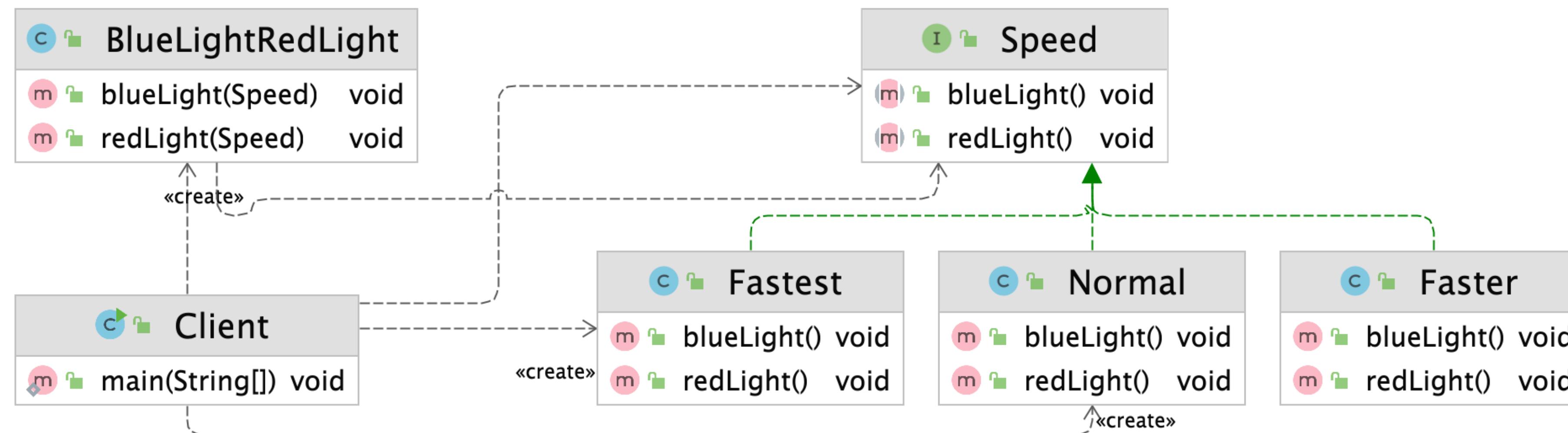
여러 알고리듬을 캡슐화하고 상호 교환 가능하게 만드는 패턴.

- 컨텍스트에서 사용할 알고리듬을 클라이언트 선택한다.



# 전략 (Strategy) 패턴

여러 알고리듬을 캡슐화하고 상호 교환 가능하게 만드는 패턴.



# 전략 (Strategy) 패턴

여러 알고리듬을 캡슐화하고 상호 교환 가능하게 만드는 패턴.

- 장점
  - 새로운 전략을 추가하더라도 기존 코드를 변경하지 않는다.
  - 상속 대신 위임을 사용할 수 있다.
  - 런타임에 전략을 변경할 수 있다.
- 단점
  - 복잡도가 증가한다.
  - 클라이언트 코드가 구체적인 전략을 알아야 한다.

# 전략 (Strategy) 패턴

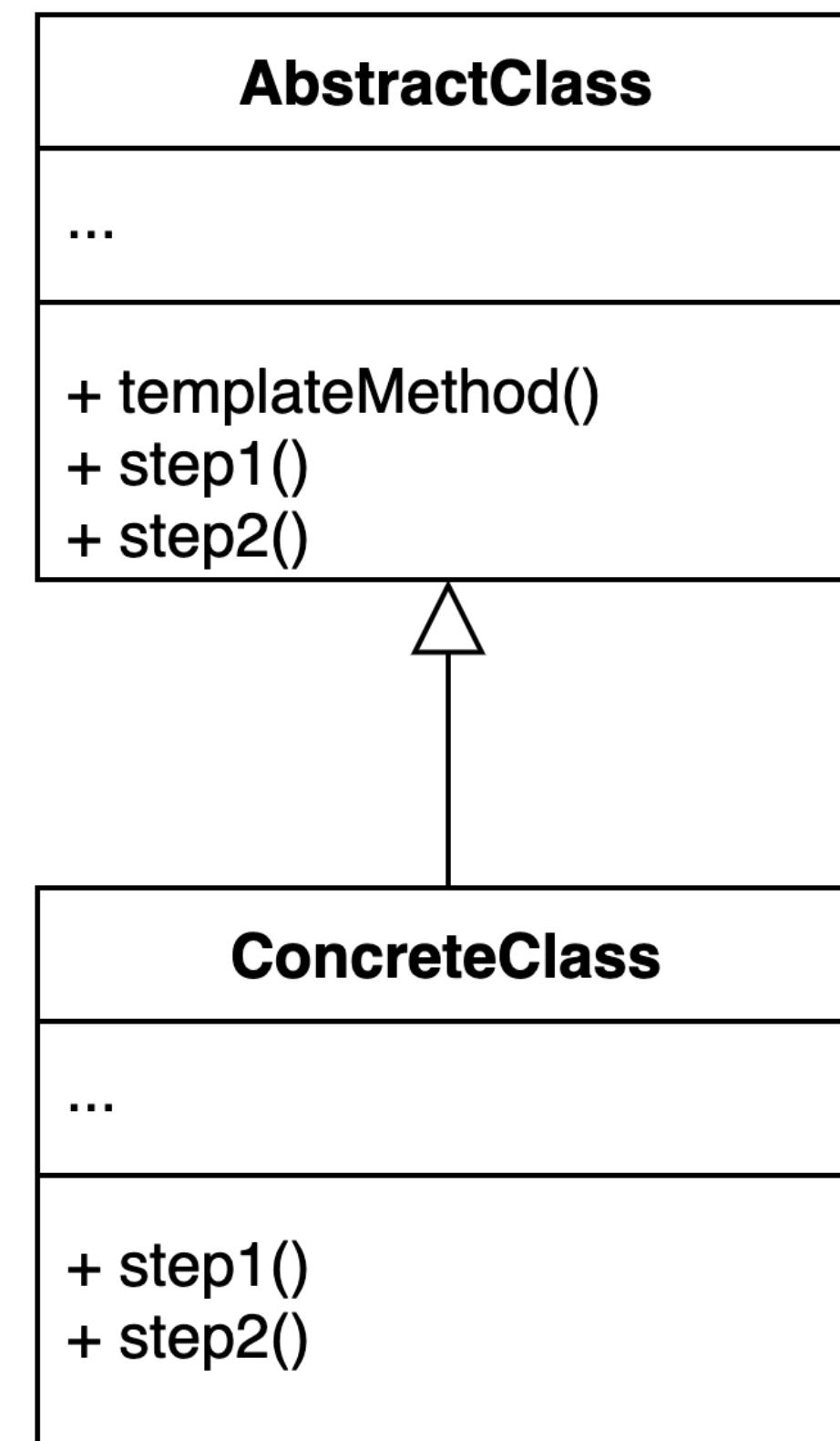
여러 알고리듬을 캡슐화하고 상호 교환 가능하게 만드는 패턴.

- 자바
  - Comparator
  - 스프링
    - ApplicationContext
    - PlatformTransactionManager
  - ...

# 템플릿 메소드 (Template method) 패턴

알고리듬 구조를 서브 클래스가 확장할 수 있도록 템플릿으로 제공하는 방법.

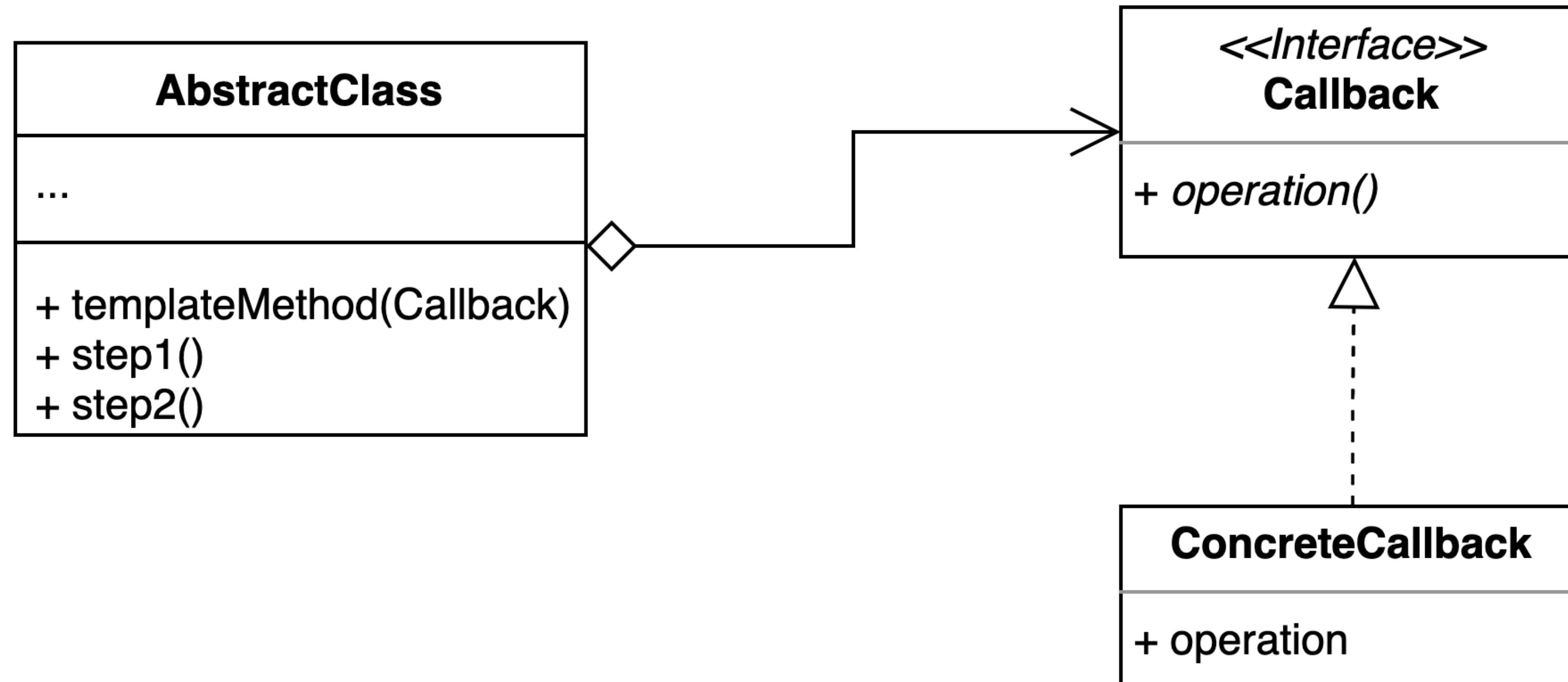
- 추상 클래스는 템플릿을 제공하고 하위 클래스는 구체적인 알고리듬을 제공한다.



# 템플릿 콜백 (Template-Callback) 패턴

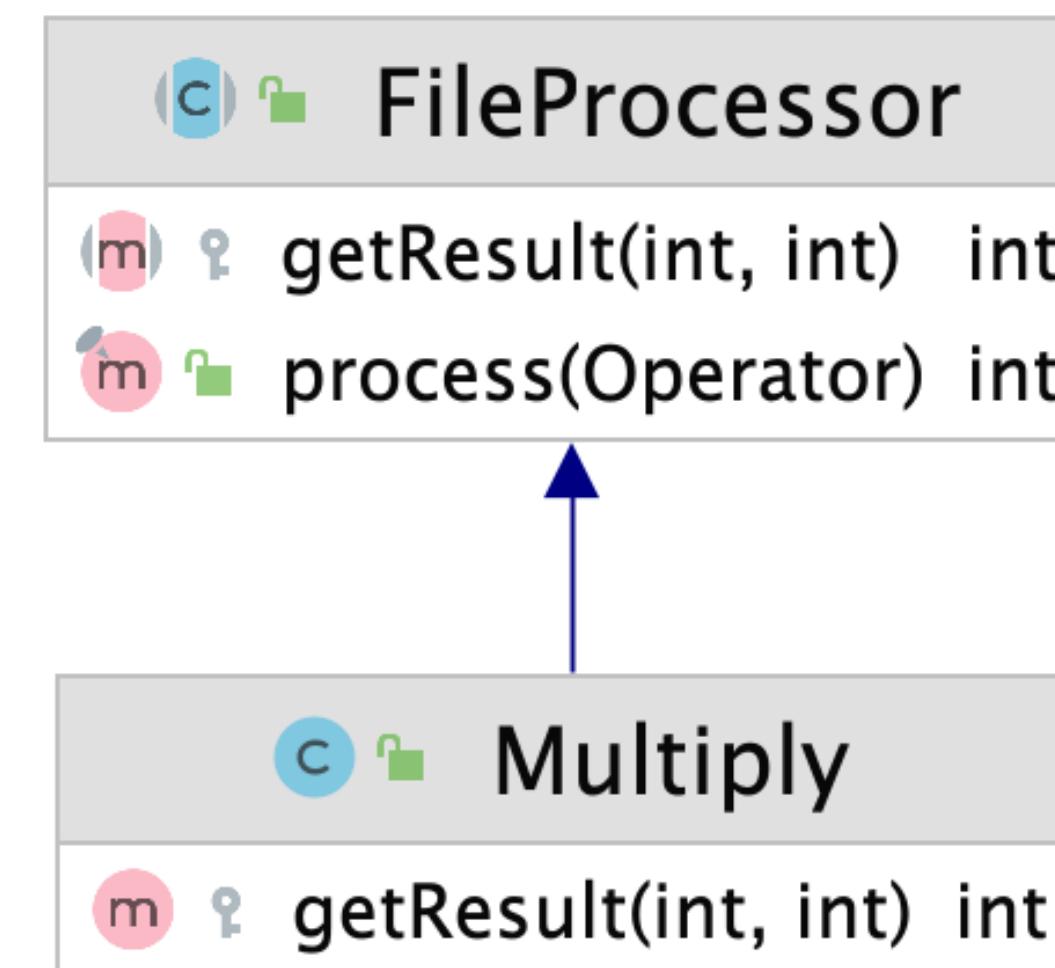
콜백으로 상속 대신 위임을 사용하는 템플릿 패턴.

- 상속 대신 익명 내부 클래스 또는 람다 표현식을 활용할 수 있다.



# 템플릿 메소드 (Template method) 패턴

알고리듬 구조를 서브 클래스가 확장할 수 있도록 템플릿으로 제공하는 방법.



# 템플릿 메소드 (Template method) 패턴

알고리듬 구조를 서브 클래스가 확장할 수 있도록 템플릿으로 제공하는 방법.

- 장점
  - 템플릿 코드를 재사용하고 중복 코드를 줄일 수 있다.
  - 템플릿 코드를 변경하지 않고 상속을 받아서 구체적인 알고리듬만 변경할 수 있다.
- 단점
  - 리스코프 치환 원칙을 위반할 수도 있다.
  - 알고리듬 구조가 복잡할 수록 템플릿을 유지하기 어려워진다.

# 템플릿 메소드 (Template method) 패턴

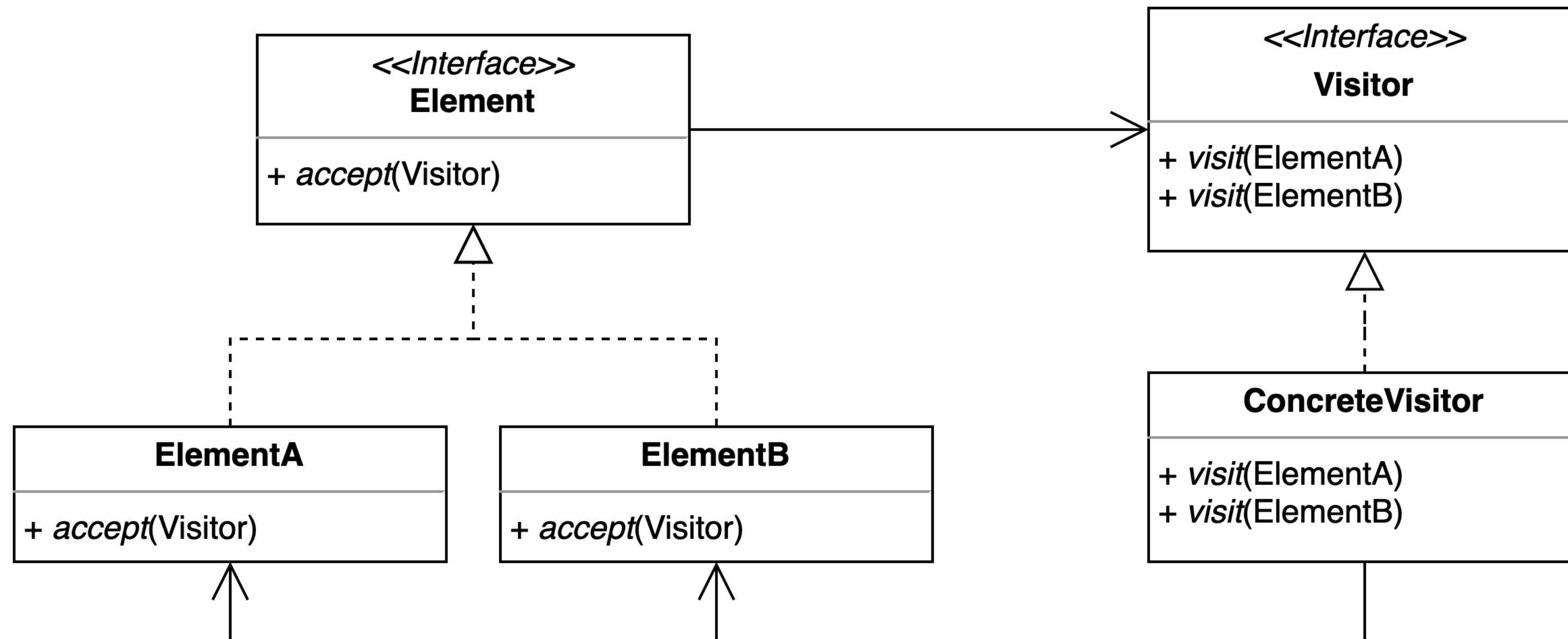
알고리듬 구조를 서브 클래스가 확장할 수 있도록 템플릿으로 제공하는 방법 .

- 자바
  - HttpServlet
- 스프링
  - 템플릿 메소드 패턴
    - Configuration
  - 템플릿 콜백 패턴
    - JdbcTemplate
    - RestTemplate
  - ...

# 방문자 (Visitor) 패턴

기존 코드를 변경하지 않고 새로운 기능을 추가하는 방법.

- 더블 디스패치 (Double Dispatch)를 활용할 수 있다.



# 방문자 (Visitor) 패턴

기존 코드를 변경하지 않고 새로운 기능을 추가하는 방법.

# 방문자 (Visitor) 패턴

기존 코드를 변경하지 않고 새로운 기능을 추가하는 방법.

- 장점
  - 기존 코드를 변경하지 않고 새로운 코드를 추가할 수 있다.
  - 추가 기능을 한 곳에 모아둘 수 있다.
- 단점
  - 복잡하다.
  - 새로운 Element를 추가하거나 제거할 때 모든 Visitor 코드를 변경해야 한다.

# 방문자 (Visitor) 패턴

기존 코드를 변경하지 않고 새로운 기능을 추가하는 방법.

- 자바
  - FileVisitor, SimpleFileVisitor
  - AnnotationValueVisitor
  - ElementVisitor
- 스프링
  - BeanDefinitionVisitor

감사합니다.

[whiteship2000@gmail.com](mailto:whiteship2000@gmail.com)  
백기선