Effective Java 읽기 스터디 10min Mini Seminar

1주차 2021.02.22

Design Pattern

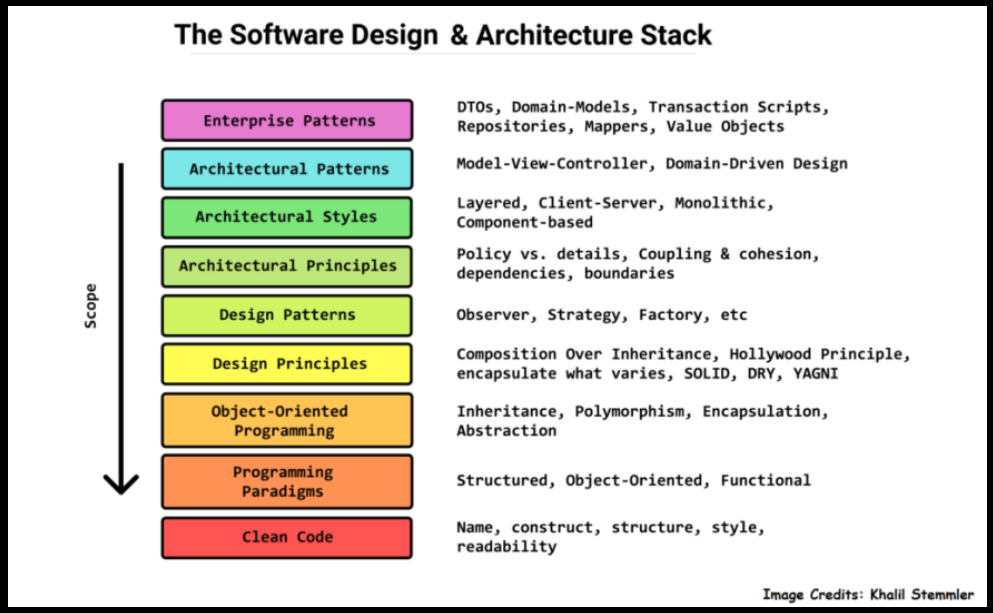
발표자: 김정수

1. 디자인 패턴

: 모듈의 세분화된 역할이나 세부적인 구현 방안을 설계할 때 참조할 수 있는 **전형적인 해결 방식**.

=>기본형 코드들이 포함되어 있음.
=> 패턴에 변형을 가하면 유사한 형태의 다른 패턴으로 변화됨.
⇒ 생성, 구조, 행위 패턴으로 분류할 수 있다.

* 아키텍처 패턴과 차이점
아키텍처 패턴은 전체 시스템의 구조를 설계하기 위한 참조 모델이다.
디자인 패턴은 시스템의 컴포넌트들과 관계를 설계하기 위한 참조 모델.



2. 생성 패턴
: 객체의 생성과 관련된 패턴. (유연성 향상)

추상 팩토리 Abstract Factory
⇒ 인터페이스로 객체를 묶는다!

빌더 Builder
⇒ 객체의 생성 과정과 표현 방법을 분리한다!

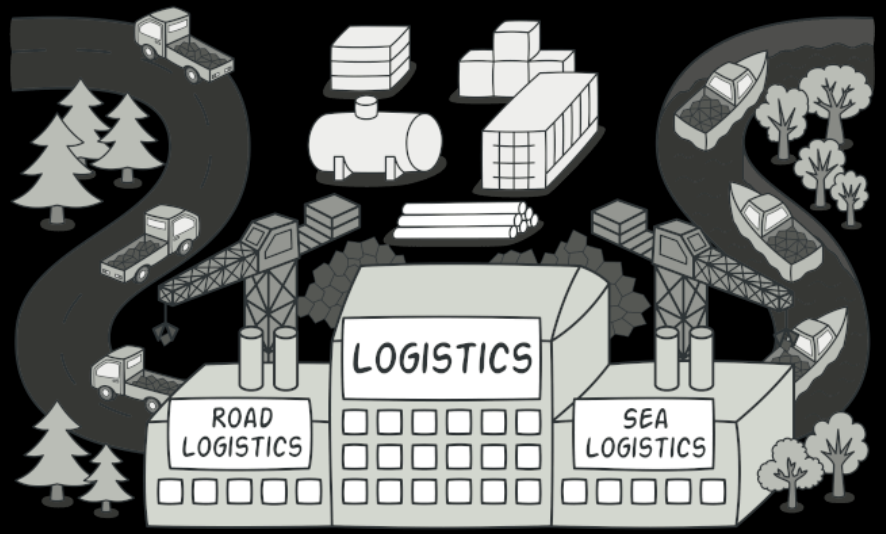
팩토리 메서드 Factory Method
⇒ Superclass의 객체를 subclass에서도 만들 수 있게 한다!

프로토타입 Prototype
=> 원본 객체를 복제한다!

싱글톤 Singleton
=> 생성된 객체가 하나뿐임을 보장한다!

팩토리 메서드 Factory Method

⇒ Superclass의 객체를 subclass에서도 만들 수 있게 한다!



Factory Method patter은 object 생성자 호출을 특별한 'factory method'에서 수행한다.

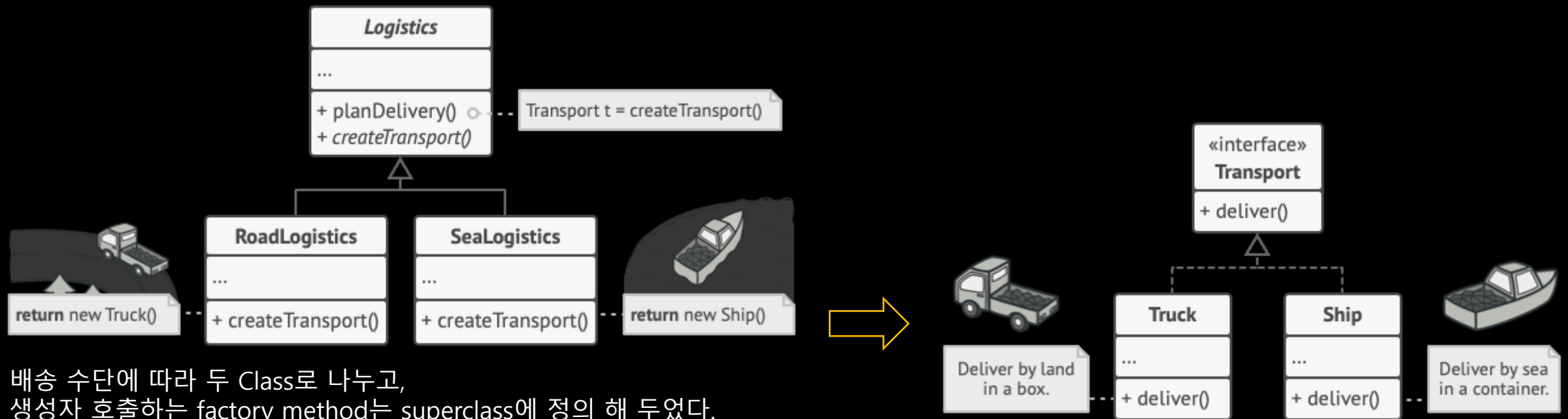
트럭만 운행하던 물류 관리 시스템에서 화물선을 추가한다면 트럭 Class에 있던 모든 내용을 화물선에 맞게 다시 짜야 한다.

⇒ App이 배송 수단의 종류에 따라서 다르게 행동하게 할 수 있어야 한다.

⇒ 배송 수단을 Class로 분리하고 공통된 부분은 superclass로 만들어 보자!

팩토리 메서드 Factory Method

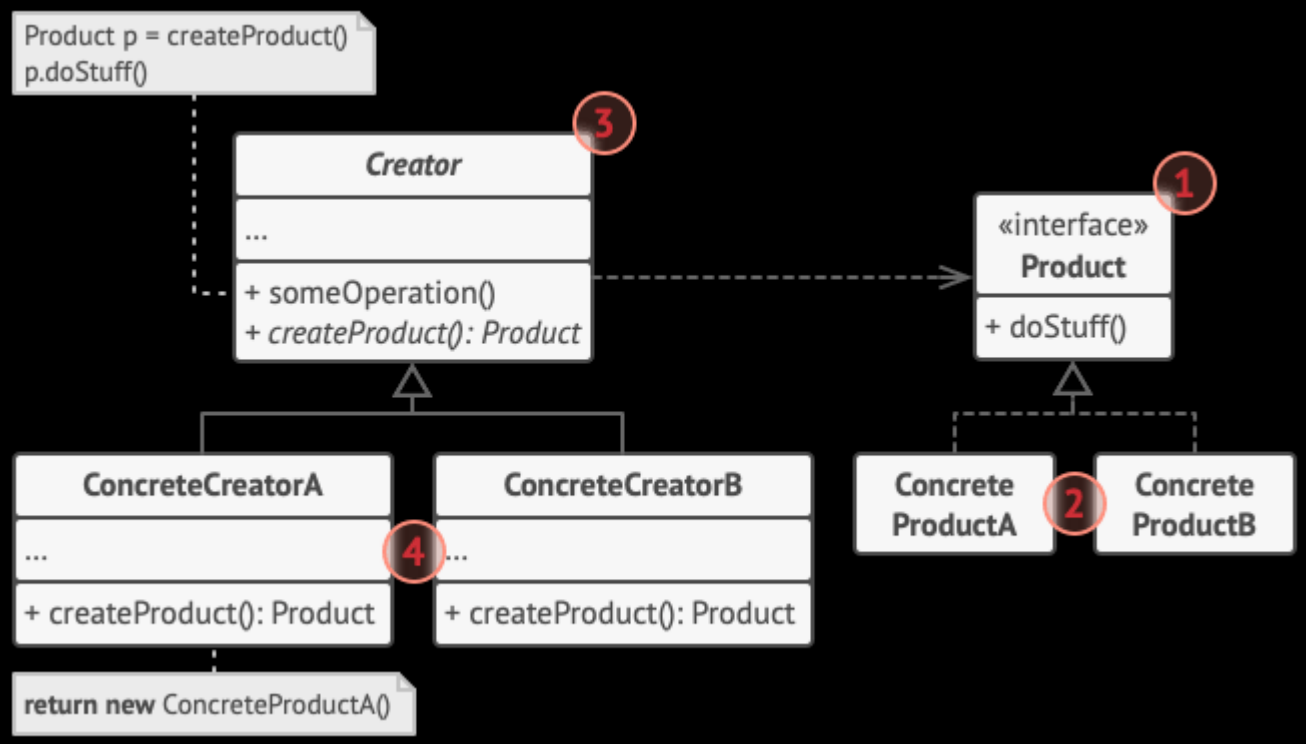
⇒ Superclass의 객체를 subclass에서도 만들 수 있게 한다!



배송 수단에 따라 두 Class로 나누고,
생성자 호출하는 factory method는 superclass에 정의 해 두었다.
⇒ 두 subclass가 서로 다른 타입의 객체(배송 수단)을 반환한다.
⇒ 단, Subclass들은 같은 interface를 구현한 타입만을 반환할 수 있다.

팩토리 메서드 Factory Method

⇒ Superclass의 객체를 subclass에서도 만들 수 있게 한다!



1) Product는 인터페이스를 정의한다.

⇒ Subclass들의 공통 부분을 정의하고, Creator가 이 타입으로 반환한다.

2) Concrete Product는 subclass의 구현부를 정의한다.

3) Creator는 factory method를 정의한다. 이 factory method는 Product type을 반환한다.

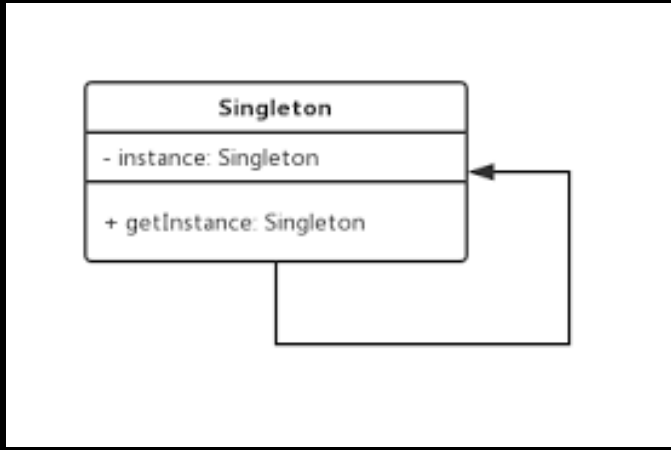
⇒ Abstract로 override를 강제할 수 있다.

4) Concrete Creators는 factory method를 override한다.

- Factory method는 항상 새 인스턴스를 만들지는 않고, 캐시나 pool에서 가져오기도 한다.

싱글톤 Singleton

=> 생성된 객체가 하나뿐임을 보장한다!



1) public static final인 멤버 변수에 인스턴스를 저장하는 방식

```
public class Class{
        public static final Class INSTANCE = new Class();
    private Class(){
        super();
        if(INSTANCE != null){
                throws DuplicatedSingletonInstanceException; // 싱글톤 법칙 위배 익셉션 (직접 만든 익셉션)
        }
    }
}
```

2) public static final인 정적 팩터리 메서드로 접근하는 방식

```
public Class{
        private static final Class INSTANCE = new Class();

    private Class{ ... 생성자는 숨긴다 ...}
    public static Class getInstance(){ return INSTANCE; } // 정적 팩터리 메서드

}
```

3. 구조 패턴

: 클래스나 객체들을 조합해 더 큰 구조로 만들 수 있게 하는 패턴.

어댑터 Adapter
⇒ 인터페이스가 서로 다른 객체들의 호환성 문제를 해결한다

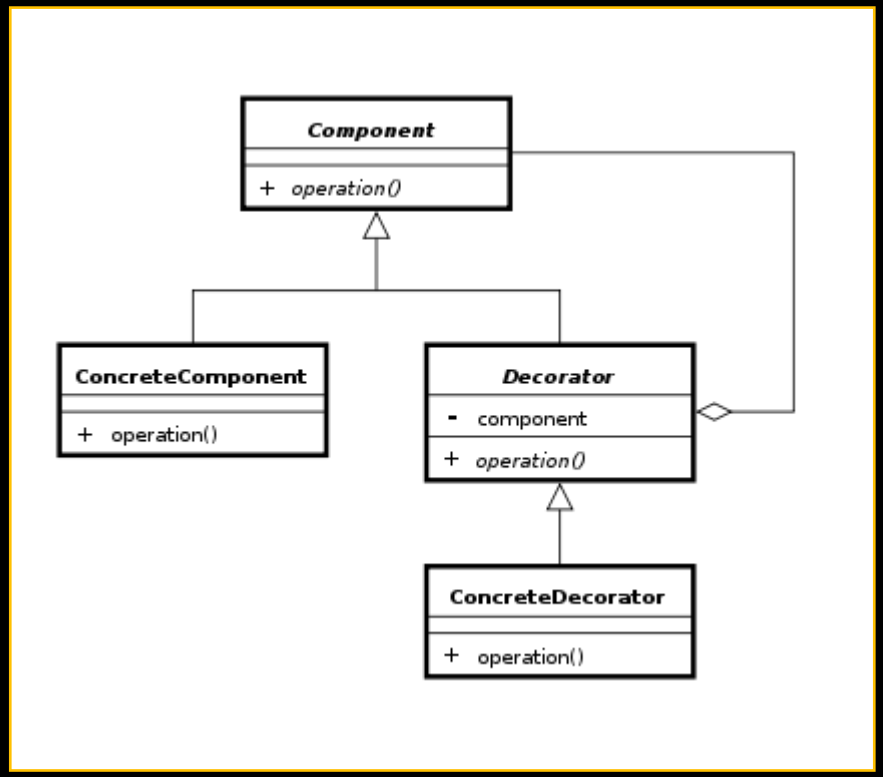
브리지 Bridge
⇒ 구현에서 기능(추상층)을 분리한다

컴포지트 Composite
⇒ 복합 객체와 단일 객체를 구분 없이 다룬다

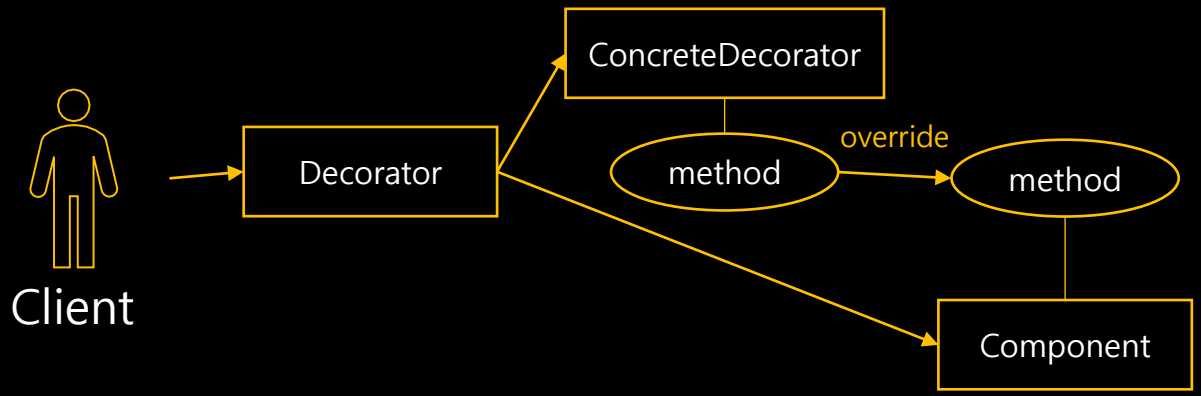
데코레이터 Decorator
=> 객체를 덧붙여서 기능을 확장시킨다

데코레이터 Decorator

=> 객체를 덧붙여서 기능을 확장시킨다



1) 기존 클래스(Component)를 Decorator 클래스의 subclass로 만든다.
2) Decorator 클래스에서는 기존 클래스를 하나의 멤버 필드로 만든다.
3) Decorator 클래스에서 기존 클래스 필드를 Decorator 생성자 파라미터로 받아서 정의한다.
4) Decorator 클래스가 기존 클래스의 모든 메서드를 포워딩하게 한다.
5) ConcreteDecorator 클래스는 기존 클래스의 메서드를 override해서 기능을 추가시킬 수 있다.



3. 구조 패턴

: 클래스나 객체들을 조합해 더 큰 구조로 만들 수 있게 하는 패턴.

퍼싸드 Façade

⇒ 통합 인터페이스를 구성한다.

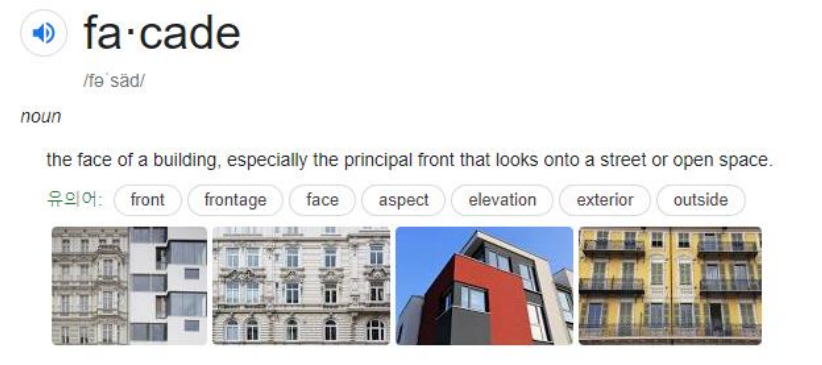
플라이웨이트 Flyweight

⇒ 인스턴스를 공유한다.

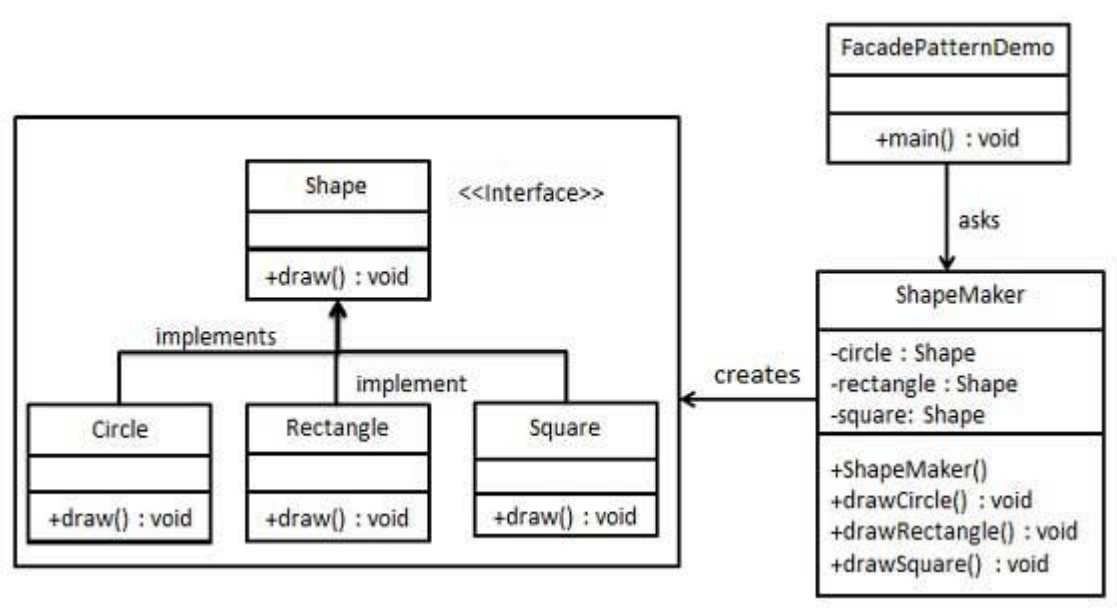
프록시 Proxy

=> 접근이 어려운 객체의 인터페이스 역할을 한다.

퍼싸드 Façade
⇒ 통합 인터페이스를 구성한다.



facade는 건물의 정면을 의미한다.
건물의 정면이 건물 내부를 대표하듯이, facade 패턴은 통합 인터페이스가 건물 정면 역할을 하게 한다. 건물 내부는 캡슐화 되어 건물 바깥에서 바로 접근할 수 없다.
프로그램의 내부가 복잡할수록 간단한 인터페이스를 제공하는 facade는 매우 유용해진다.



1) 인터페이스를 만들고, 행위를 정의한다.
2) 인터페이스를 구현한 클래스를 만든다.
3) 퍼싸드 클래스를 만든다. 퍼싸드 클래스는 클라이언트가 직접 접근할 수 있다.

4. 행위 패턴

: 객체의 상호작용/책임분배 방법을 정의 (결합도 최소화)

책임 연쇄 Chain of Responsibility
⇒ Chain으로 연결된 다른 객체에게 책임을 넘길 수 있다.

커맨드 Command
⇒ 요청을 캡슐화해 저장/로깅한다. (재사용 Good)

인터프리터 Interpreter
⇒ 언어의 문법 표현을 정의한다.

반복자 Iterator
=> 접근이 잦은 객체에 대해 동일 인터페이스를 사용하도록 한다. (캡슐화)

중재자 Mediator
⇒ 객체 간 상호작용을 캡슐화한다. (의존성 낮춤)

4. 행위 패턴

: 객체의 상호작용/책임분배 방법을 정의 (결합도 최소화)

메멘토 Memento
⇒ 특정 시점에서의 객체 내부 상태를 객체화한다.

옵서버 Observer
⇒ 객체 상태가 바뀌면 상속받는 다른 객체에게 상태를 전달한다.

상태 State
⇒ 객체 상태 캡슐화

전략 Strategy
⇒ 알고리즘을 캡슐화

템플릿 메서드 Template Method
⇒ 상위 클래스에서 틀을 정의하고, 하위 클래스에서 행위를 구체화한다.

방문자 Visitor
⇒ 데이터 구조에서 처리 기능을 분리

References

- Factory Method Pattern (https://refactoring.guru/design-patterns/factory-method)
- Decorator Pattern(https://en.wikipedia.org/wiki/Decorator_pattern)
- Facade Pattern(https://www.tutorialspoint.com/design_pattern/facade_pattern.html)