Composite Pattern 와 Transparency

Composite Pattern 은 SRP 를 희생하여 **Transparency** 를 달성하는 디자인 패턴이다. Transparency 는 클라이언트 코드가 Composite 객체와 Leaf 객체를 동일한 방식으로 처리하도록 하여, 어떤 객체가 복합 객체인지 Leaf 인지 클라이언트 입장에서 투명하게 보이도록 만드는 것을 의미한다. 이를 통해 객체의 내부 구조를 숨기고 클라이언트 로직을 단순화할 수 있다.

Transparency 는 소프트웨어 설계에서 다음과 같은 이점을 제공한다. 클라이언트가 Leaf 와 Composite 객체를 구분할 필요가 없기 때문에 코드가 간결해진다. 둘째, Extensibility(확장성)이다. 새로운 객체 유형(Leaf 또는 Composite)을 추가하더라도 기존 클라이언트 코드를 수정할 필요가 없다. 셋째, Maintainability(유지보수성)이다. 객체 계층 구조의 변경이 클라이언트 코드에 미치는 영향을 최소화하여 유지보수가 용이하다.

Composite Pattern 에서는 Component 인터페이스를 상속받는 객체들이 Leaf 와 Composite 두 가지 역할을 모두 가지도록 설계한다. Composite 객체는 Leaf 뿐만 아니라 다른 Composite 객체도 포함할 수 있다.

예를 들어, 책에 소개된 MenuComponent 클래스는 공통 인터페이스를 제공하여 클라이언트가 Menu 와 MenuItem 객체를 동일하게 처리할 수 있도록 한다. 이를 통해 클라이언트는 개별 객체가 Leaf 인지 Composite 인지 신경 쓰지 않고 print()와 같은 메서드를 호출하여 전체 구조를 탐색할 수 있다. 실제 클라이언트인 Waitress 클래스는 최상위 MenuComponent 만 알고 있어도 내부의 모든 Menu 와 MenuItem 을 재귀적으로 출력할 수 있다.

이러한 설계는 소프트웨어 품질을 높이는데 기여한다. Composite Pattern 은 **Reusability(재사용성)**을 높여 공통 인터페이스를 통해 다양한 객체를 관리할 수 있게 한다. 또한, **Flexibility(유연성)**을 제공하여 Composite 객체가 계층 구조를 유연하게 확장할 수 있게 한다. 마지막으로, **Maintainability(유지보수성)**을 보장하여 클라이언트가 계층 구조의 변경에도 최소한의 수정만으로 대응할 수 있도록 한다.

따라서 설계자는 디자인 원칙을 상황에 맞게 유연하게 적용해야 하며, 원칙이 설계에 미치는 영향을 항상 고려해야 한다. 필요에 따라 일부 원칙을 의도적으로 위배하는 것도 합리적인 선택이 될 수 있다.

Factory Pattern 의 목표는 객체 생성의 복잡성을 캡슐화하여 클라이언트 코드와 객체 생성 과정을 분리하는 것이다. 이를 통해 클라이언트는 객체 생성에 필요한 세부 사항을 몰라도 필요한 객체를 얻을 수 있다. 예를 들어, orderPizza() 메서드는 Pizza 객체를 생성하기 위해 상위 클래스에서 createPizza() 메서드를 호출하며, 구체적인 생성 로직은 서브클래스(NYPizzaStore, ChicagoPizzaStore)에서 구현된다. 이렇게 함으로써 클라이언트는 PizzaStore 의 추상화된 메서드만 호출하면 된다.

Template Method Pattern 의 목표는 알고리즘의 구조를 고정하면서도, 세부적인 구현은 서브클래스에 위임하는 것이다. 이는 알고리즘의 일관성을 유지하면서도 다양한 변형을 허용한다. 예를 들어, prepareRecipe() 메서드는 brew()와 addCondiments() 같은 추상 메서드를 서브클래스(Coffee, Tea)에서 구현하도록 하여 알고리즘의 각 단계를 커스터마이징할 수 있게 한다.

유사점: 두 패턴 모두 "공통적인 작업을 상위 클래스에서 정의하고, 세부 구현은 하위 클래스에서 처리한다"는 점에서 구조적으로 유사하다. Factory Pattern 의 orderPizza()와 Template Method Pattern 의 prepareRecipe()는 상위 클래스에서 정의된 메서드로, 세부 구현에 대한 책임을 서브클래스에 위임한다. 이를 통해 공통 로직의 재사용성을 높이고, 특정 동작의 커스터마이징을 허용하는 구조를 만들어낸다.

왜 비슷하게 보이는가

두 패턴은 **추상화와 캡슐화**를 통해 코드 중복을 줄이고 유연성을 높이려는 공통된 디자인 원칙을 따른다. 특히, 상위 클래스에서 동작의 골격을 정의하고, 세부 구현은 서브클래스에서 다룬다는 구조적 유사성 때문에 비슷하게 보인다.

헐리우드 원칙(Hollywood Principle)은 고수준 컴포넌트가 저수준 컴포넌트를 제어하도록 설계하는 원칙이다. 이는 의존성 부패를 방지하고, 설계의 명확성을 유지하는 데 기여한다.

Template Method Pattern 은 알고리즘의 흐름을 상위 클래스에서 정의하고, 세부 구현은 하위 클래스에 위임한다. 예를 들어, CaffeineBeverage 추상 클래스는 prepareRecipe() 템플릿 메서드를 통해 커피와 차를 만드는 과정을 정의한다. 공통 단계는 상위 클래스에서 구현하고, brew()와 addCondiments() 같은 세부 작업은 하위 클래스에서 구현한다.

이 패턴은 상위 클래스가 알고리즘의 흐름을 제어하면서 하위 클래스는 필요한 부분만 구현하게 하여, 상위 클래스와 하위 클래스 간 결합도를 낮추고, 코드 재사용성을 높이며, 유지보수를 용이하게 만든다. 결과적으로 Template Method Pattern 은 헐리우드 원칙을 활용하여 유연하고 확장 가능한 설계에 기여한다.

Command Pattern 은 "요청을 객체로 캡슐화(encapsulate)"하여 클라이언트와 실행 주체(Receiver)를 분리하고, 유연하고 확장 가능한 소프트웨어 설계를 가능하게 하는 디자인 패턴이다. 이 패턴에서 요청을 캡슐화한다는 것은 특정 작업을 Command 객체로 정의하고, 요청의 세부 사항과 실행 로직을 포함하게 만드는 것을 의미한다.

교과서의 예제에서 LightOnCommand 클래스는 전등을 켜는 요청을 캡슐화한다. execute() 메서드는 요청을 실행하며, undo() 메서드는 이전 요청을 취소할 수 있다. 리모컨(RemoteControl)과 같은 Invoker 는 Command 객체를 호출하여 Receiver(Light)의 동작을 실행하며, 클라이언트는 요청의 구체적인 세부 사항을 몰라도 된다. 이를 통해 클라이언트와 Receiver 간의 결합도가 낮아져 시스템의 유연성이 높아진다. 또한, Command 객체를 저장하거나 조합하여 작업 큐와 같은 복잡한 로직을 구현할 수 있어 reusability 이 높아진다. 새로운 요청이 추가될 때 기존 코드를 수정하지 않아도 되므로 OCP 를 준수할 수 있다.

결론적으로, Command Pattern 은 요청과 실행을 분리하여 소프트웨어의 복잡성을 줄이고, 재사용성과 유지보수성을 극대화하여 고품질 소프트웨어 개발에 기여한다.

DIP 는 상위 수준 모듈이 하위 수준 모듈에 의존하지 않고, 둘 다 추상화에 의존하도록 설계하는 원칙이다. 이는 객체 간 결합도를 낮추고, 시스템의 유연성과 재사용성을 극대화하여 유지보수를 용이하게 한다. DIP 는 상위 수준 로직이 세부 구현에 묶이지 않도록 하여 변화를 쉽게 처리할 수 있도록 돕는다.

즉, DIP 는 OCP 를 구현하기 위한 핵심 메커니즘이다. DIP 를 적용하면 상위 모듈과 하위 모듈이 구체적인 구현이 아닌 추상화에 의존하게 되어, 새로운 기능을 추가하거나 기존 기능을 변경하더라도 기존 코드의 수정 없이 확장이 가능하다. OCP 의 '확장에는 열려 있고, 수정에는 닫혀 있는' 목표를 DIP 로 달성할 수 있다.

Factory Pattern: DIP 를 활용하여 객체 생성 로직을 추상 팩토리로 분리하고, 클라이언트가 특정 구현에 의존하지 않도록 설계한다. 예를 들어, PizzaStore 는 PizzaIngredientFactory 인터페이스를 통해 구체적인 재료 생성 책임을 분리하여 새로운 피자 유형 추가 시 수정 없이 확장 가능하다.

Class Diagram: PizzaStore (상위 클래스)는 createPizza() 메서드로 피자를 생성하며, 재료 생성은 PizzaIngredientFactory 를 활용한다. 하위 클래스(NYPizzaStore, ChicagoPizzaStore)는 createPizza()를 구체적으로 구현하여 DIP 와 OCP 를 만족한다.

Template Method Pattern: 알고리즘 구조를 상위 클래스에서 정의하고, 세부 구현은 하위 클래스에 위임하여 구조 변경 없이 다양한 구현을 가능하게 한다.**Class Diagram**: CaffeineBeverage (상위 클래스)는 prepareRecipe() 템플릿 메서드를 정의하여 알고리즘의 흐름을 제어한다. 하위 클래스(Tea, Coffee)는 brew()와 addCondiments() 메서드를 구체적으로 구현하여 세부 단계를 정의한다. 이 방식은 DIP 를 통해 알고리즘 구조를 고정하고, 하위 클래스에서 세부 구현을 확장 가능하게 한다.

DIP 는 OCP 를 실현하는 데 필수적인 원칙으로, 클라이언트와 구현 클래스 간 결합도를 낮추고, 새로운 기능 추가와 유지보수를 용이하게 한다. DIP 가 적용된 Factory, Template pattern 과 같이 유연하고 확장 가능한 설계를 가능하게 한다.

최소지식의 원칙(Principle of Least Knowledge)은 객체가 다른 객체와 상호작용할 때, 가능한 한 적은 지식만을 가져야 한다는 객체 지향 설계 원칙이다. 이 원칙은 객체가 자신과 직접 연관된 객체들만 참조하며, 불필요하게 많은 객체와 상호작용하지 않도록 제한함으로써 시스템의 복잡도를 낮추고 유지보수를 용이하게 한다. 객체 간의 상호 의존성을 줄이면 하나의 객체가 변경되었을 때 그 영향이 다른 객체들로 확산되는 것을 방지할 수 있어 시스템의 안정성과 예측 가능성이 크게 향상된다. 이는 소프트웨어 설계에서 매우 중요한 요소로, 코드의 유지보수성뿐만 아니라 모듈성과 재사용성을 향상시키며, 결과적으로 소프트웨어 품질을 높이는 데 기여한다.

268 쪽의 두 가지 getTemp() 메서드를 비교하면, 첫 번째 버전은 House 객체가 WeatherStation 뿐만 아니라 그 내부의 Thermometer 객체까지 직접 접근하도록 설계되어 있다. 이는 House 와 Thermometer 사이에 강한 결합이 발생하며, Thermometer 의 내부 구현이 변경될 경우 House 에도 영향을 미치게 된다. 반면 두 번째 버전은 House 가 WeatherStation 과만 상호작용하고, WeatherStation 이 내부적으로 Thermometer 를 관리하도록 설계되어 있다. 이 경우 House 는 Thermometer 의 존재를 알 필요가 없으므로, 의존성이 감소하고 객체 간의 결합도가 낮아진다.

두 번째 버전이 더 좋은 이유는 최소지식의 원칙을 따름으로써 코드의 유지보수성과 안정성을 높이기 때문이다. 객체가 다른 객체의 내부 구조를 알지 않아도 되므로, 내부 구현이 변경되더라도 외부 객체에 영향을 주지 않는다. 이는 객체 간의 관계를 단순하게 만들어 시스템의 유연성을 높이고, 수정과 확장을 용이하게 한다.

Encapsulation 은 알고리즘을 독립적인 클래스에 숨겨 클라이언트가 세부 구현을 알 필요 없게 하며, 예를 들어 QuackBehavior 를 구현한 Quack, MuteQuack, Squeak 클래스는 각각의 알고리즘을 외부로부터 보호하는 역할을 한다. 이는 구현 세부사항의 변경이 클라이언트 코드에 영향을 미치지 않도록 설계할 수 있게 한다.

Abstraction 은 공통적인 행동을 추출해 인터페이스나 추상 클래스로 정의하는 것으로, QuackBehavior 는 울음소리 행동의 추상화를 제공하며, 이를 기반으로 구상클래스에서 각각 구현된다.

Inheritance 는 알고리즘 계층에서 코드 재사용성을 높이는 데 활용되는 개념이다. QuackBehavior 인터페이스를 구현하는 클래스들은 행동의 기본 구조를 상속받아 코드 중복을 줄이고, 필요에 따라 구체적인 세부사항을 재정의할 수 있어 유지보수가 용이하다.

Polymorphism 은 클라이언트가 구체적인 알고리즘 클래스에 의존하지 않고 인터페이스를 통해 알고리즘을 사용할 수 있게 하는 특징이다. 예를 들어, Duck 클래스는 QuackBehavior 타입의 객체를 활용해 행동을 위임받으며, Quack, MuteQuack, Squeak 등 다양한 알고리즘을 런타임에 동적으로 교체할 수 있다. 이를 통해 코드의 변경이나 확장이 매우 유연하게 가능하다.

싱글턴 패턴(Singleton Pattern)은 특정 클래스의 인스턴스를 단 하나만 생성하고, 전역적으로 이 인스턴스에 접근할 수 있도록 보장하는 디자인 패턴이다. 주로 스레드 풀, 캐시, 로그 기록, 설정 관리와 같이 하나의 인스턴스만 필요할 때 사용된다. 싱글턴 패턴은 메모리 낭비를 줄이고 결과의 일관성을 유지하는 장점이 있다. 이를 구현하기 위해 일반적으로 private 생성자를 통해 외부에서 객체 생성을 막고, static 메서드인 getInstance()를 통해 유일한 인스턴스를 제공한다.

그러나 전통적인 구현 방식에서는 멀티스레드 환경에서 여러 스레드가 동시에 getInstance()를 호출해 의도치 않게 여러 인스턴스가 생성될 가능성이 있다. 이를 해결하기 위해 다양한 방법이 제안되었다. 첫 번째로, getInstance() 메서드를 동기화(synchronized)하는 방법이 있지만, 이는 성능 저하를 유발할 수 있다. 두 번째로, 애초에 인스턴스를 정적으로 생성해 클래스 로딩 시점에 생성하는 방법(Eager Initialization)이 있다. 이 방법은 초기화 비용이 들지만, 멀티스레드 문제를 효과적으로 방지한다. 세 번째로, Double Checked Locking(DCL) 기법을 사용하여 최초 생성 시에만 동기화를 적용하는 방식이 있다. 이는 성능과 안전성을 모두 고려한 방법이지만, 변수에 volatile 을 사용해야 하며, Java 1.4 이하에서는 사용할 수 없다.

이 외에도 보다 안정적이고 효율적인 구현 방법이 존재한다. Enum 을 이용한 방법은 리플렉션 공격과 직렬화 문제를 방지하고, JVM 의 기본적인 동작을 활용하여 안전한 싱글턴을 보장한다. 단, Enum 은 클래스 상속이 불가능하므로 제한적인 경우에만 사용할 수 있다. Lazy Holder 방식은 JVM 의 클래스 초기화 과정에서 원자성을 보장하는 특성을 활용해 동기화 없이 안전한 인스턴스 생성을 가능하게 한다. 이는 성능이 중요한 환경에서 가장 많이 사용되는 기법 중 하나이다.