디자인패턴(정진홍)

Lee Jun Hyeok (wnsx0000@gmail.com)

December 12, 2025

목차

1. DP

1.1. SOLID

1.1.1. SOLID

SOLID는 2000년에 Rober C. Martin에 의해 정의된 design 원칙으로, object-oriented sw를 견고하고 확장 가능하게 design할 수 있도록 하는 5가지 기본 원칙(SRP, OCP, LSP, ISP, DIP)을 포함하고 있음.

SRP와 ISP는 object가 과하게 커지지 않게 함으로써 한 기능의 변경이 미치는 영향을 최소화함. 즉, 기능 변경을 용이하게 함.

LSP와 DIP는 OCP를 구현하도록 도움. LSP는 polymorphism을, DIP는 abstraction을 효과적으로 활용하도록 함.

design pattern들은 SOLID에 입각해서 design되어 있음.

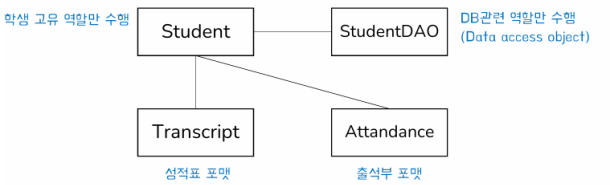
1.1.2. SRP

SRP(Single Responsibility Principle, 단일 책임 원칙)는 모든 class는 단 하나의 책임만을 가져야 한다는 원칙임.

이때 책임은 기능으로 이해할 수 있음. 즉, 하나의 class는 하나의 기능에 집중하고, 수반되는 모든 작업을 스스로 수행할 수 있어야 함. 예를 들어, Student class는 수강 과목을 추가하고 조회하는 책임만을 가져야 하지, 데이터베이스에 object 정보를 저장하고 꺼내오는 등의 책임을 가질 필요는 없음.

하나의 class가 여러 책임을 가지면 해당 class 내에서 서로 다른 기능에 대한 코드가 서로 강하게 결합되어 있으르 수 있으므로, 변경에 유연하게 대처하기 어려움. 예를 들어, 이전 예시에서처럼 design되었다면 데이터베이스 수정이 있을 때 Student class를 수정해야 함.

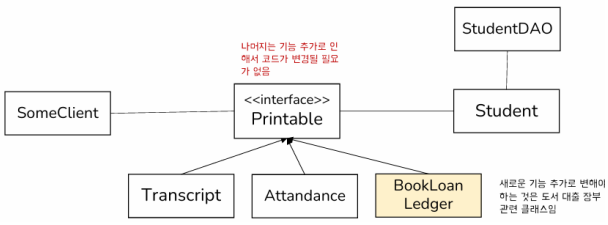
이에 따라 하나의 class에 여러 책임이 있다면 책임 분리를 통해 여러 class로 나누어야 함.



1.1.3. OCP

OCP(Open-Closed Principle, 개방-폐쇄 원칙)는 확장에는 열려 있어야 하고, 수정에는 닫혀 있어야 한다는 원칙임. 즉, 변경 사항에 따른 확장은 용이해야 하지만, 이때 기존 코드의 수정이 없어야 함.

inheritance 및 polymorphism을 적절히 활용해 이를 구현할 수 있음. 특히 변하지 않는 것은 interface 및 abstract class로, 변하는 것은 이를 구현하는 subclass로 design해야 함.

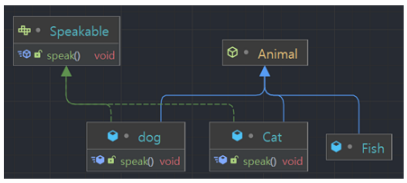


1.1.4. LSP

LSP(Liskov Substitution Principle, 리스코프 치환 원칙)는 상위 타입의 object를 하위 타입의 object로 치환해도 프로그램이 정상적으로 동작해야 한다는 원칙임. 즉, 자식 class가 부모 class의 행동 규약을 지켜야 한다는

것임.

자식 class에서 부모 class의 의도 또는 형식과 다르게 오버라이딩하거나, 잘못된 inheritance 관계를 구성해 특정 method에서 의미 없는 동작을 하도록 design하는 경우 등에서 LSP가 지켜지지 않는 것을 알 수 있음. 다음은 그 예시임.



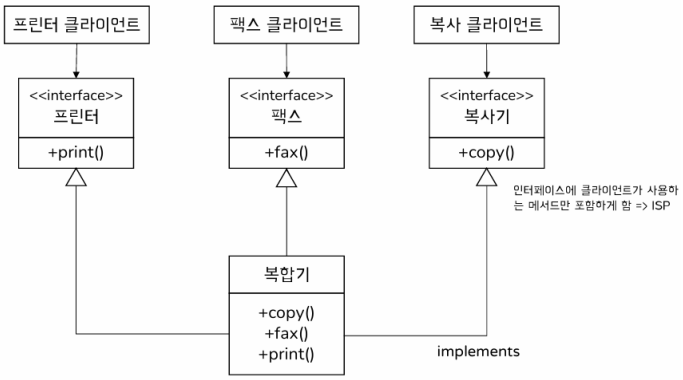

```
List<Speakable> animals = new ArrayList<>();
animals.add(new Cat());
animals.add(new Dog());
animals.add(new Fish());

for(Speakable animal : animals)
    animal.speak();
```

1.1.5. ISP

ISP(Interface Segregation Principle, 인터페이스 분리 원칙)는 시스템을 사용하는 클라이언트가 사용하지 않는 method에 의존하지 않아야 한다는 원칙임. 즉, interface에 클라이언트가 사용하는 method만 포함되어 있어야 하고, interface를 사용에 맞게 분리하여 interface가 단일 책임을 가져야 함.

SRP가 class의 단일 책임에 대한 것이었다면, ISP는 interface의 단일 책임에 대한 것임. 이때 interface의 분리는 클라이언트 기준으로 수행되어야 함. 책임 분리라는 측면에서 SRP와 ISP가 유사하게 보이지만, class 별로 interface를 만들었을 때 클라이언트 별러 하나의 class가 가지는 기능 중 일부만 활용할 수 있음.

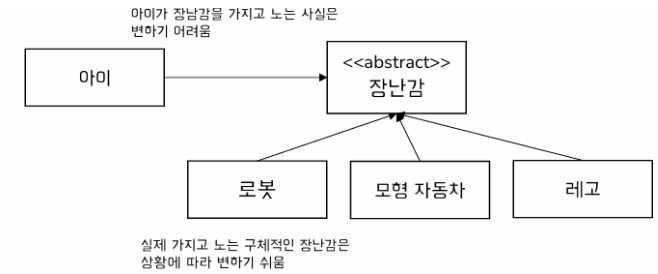


1.1.6. DIP

DIP(Dependency Inversion Principle, 의존 역전 원칙)는 의존 관계는 변화하기 어렵거나, 변화가 존재하지 않는 class에 대해 맺어야 한다는 원칙임. 즉, 의존 관계를 맺을 때는 추상성이 낮은 class보다는, 추상성이 높은 abstract class나 interface를 선택해야 함.

DIP를 잘 지키면 dependency injection을 통해 수정에 용이하도록 코드를 구성할 수 있음. Dependency Injection(의존성 주입)은 어떤 object에 필요로 하는 의존 object를 외부에서 주입하는 것임. 즉, 추상성이 높은 대상에 의존하면 해당 대상을 구체화하는 대상 모두를 주입받을 수 있음.

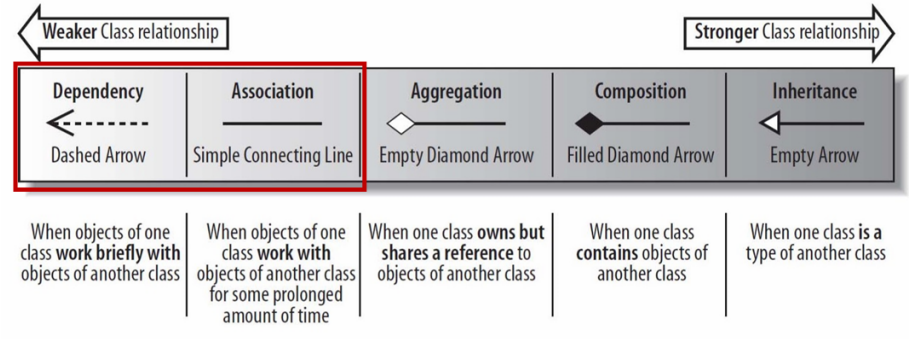
기존에는 추상성이 높은 class가 추상성이 낮은 class에 직접적으로 의존하는 경우가 많았는데, DIP에서는 이를 역전(inversion)하여 추상성이 높던 낮건 추상성이 높은 class에 의존해야 한다고 주장한 것임.



1.2. Class Relationship

1.2.1. Class Diagram Relationship

class diagram에서 class/object 간의 다양한 관계를 표현할 때는 다음과 같이 5가지 분류를 사용함.



- Dependency(의존 관계)는 한 class의 object에서 다른 class의 object를 일시적으로(temporarily) 사용하는 관계로, 점선 화살표로 표기함.
- Association(연관 관계)은 한 class의 object가 다른 class의 object와 장기간 연관되어 동작하는 관계로, 실선으로 표기함.
- Aggregation(집합 관계)은 부분이 전체에 약하게(use) 속하는 관계임. hollow diamond(빈 다이아몬드) 화살표로 표기하는데, 전체에 해당하는 class에 hollow diamond를 그림. 이때 부분은 전체와 상관없이 독립적으로 존재할 수 있음. 즉, 전체 object가 사라져도 부분 object는 존재할 수 있는 관계로, 전체와 부분이 생명주기가 다름.
- Composition(합성 관계)은 부분이 전체에 강하게(own) 속하는 관계임. filled diamond(차 있는 다이아몬드) 화살표로 표기하는데, 전체에 해당하는 class에 filled diamond를 그림. 이때 부분은 전체에 종속되어 독립적으로 존재할 수 없음. 즉, 전체 object가 사라지면 부분 object로 사라지고, 전체와 부분의 생명주기가 같음.
- Inheritance(상속 관계, 일반화 관계)는 subclass들의 공통된 특성을 super class로 일반화한 관계로, 화살표로 표기함. subclass는 private인 것들을 제외하고 super class의 모든 것을 상속받으며, subclass 별로 추가적인 속성을 가질 수 있음. 이에 따라 subclass의 instance는 super class에 대한 간접적인 instance임.

1.3. DP

1.3.1. DP

1. DP

DP(Design Pattern)는 소프트웨어 design 및 개발 과정에서 자주 발생하는 일반적인 문제들에 대해, 지금까지 개발자들이 경험적으로 사용해 온 해결 방법(best practice)을 패턴화한 것임. 이는 특정 코드가 아니라 구조적 아이디어로, 주로 UML로 표기함.

DP는 상황에 대한 맥락인 Context, 문제 상황인 Problem, 문제에 대한 해결책인 Solution으로 나누어 정의될 수 있음.

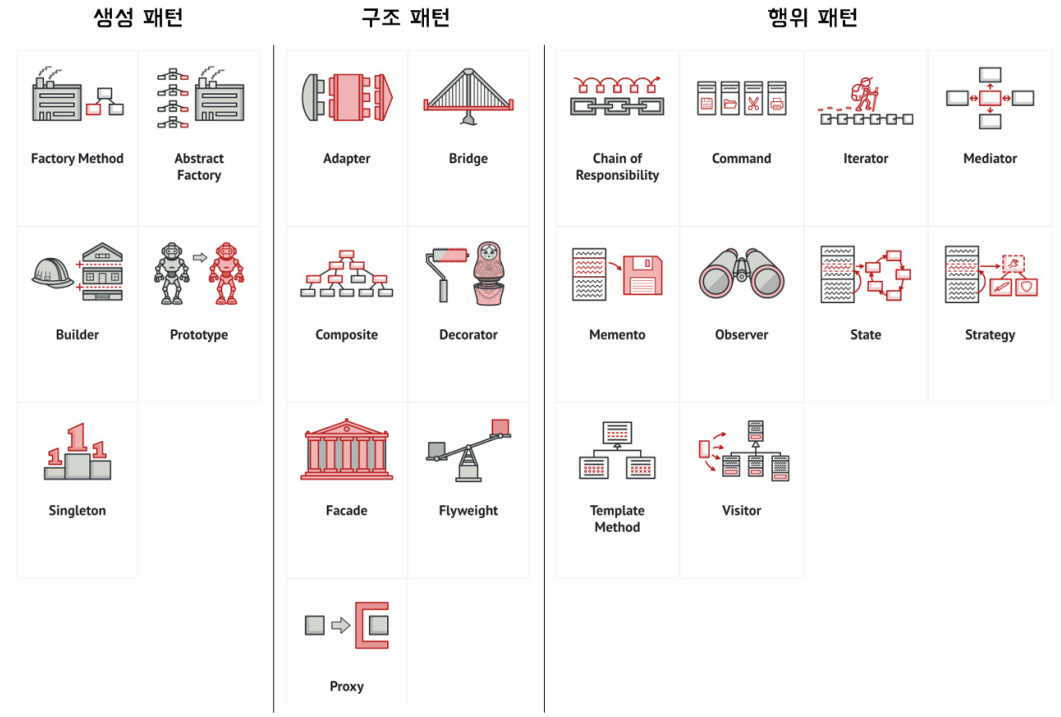
DP를 사용하면 개발자들 간의 소통이 원활해지고, 해당 방법이 경험적으로 최적화된 방식이므로 시행착오를 줄일 수 있음. 하지만 당연하게도 DP를 무작정 남용하는 것은 적절하지 않고, 시스템의 구조와 설계를 잘 고려해서 적용해야 함. DP를 쓰지 않고도 문제를 간단히 해결할 수 있다면 사용할 이유가 없음(오컴의 면도날).

참고로 이제부터 설명하는 DP에 대해서, Client는 해당 DP를 사용하는 코드 또는 사용자임. DP가 SOLID를 잘 지키고 있는 경우에도, client에서 object를 사용할 때는 구체적인 class명을 사용하는 경우가 많음. 특히 필요한 object를 주입하거나 하는 등의 코드가 사용될 수 있음.

2. Categories of DP

초기 DP는 smalltalk, c++ 등으로 설명되다가, GoF(Gang of Four)에 의해 출간된 "Design Patterns: Elements of Reusable Object-Oriented Software"에서 설명하는 23가지 분류가 가장 널리 쓰임. 해당 책에서는 다음과 같이 DP를 분류했고, 본 수업에서도 이 분류를 사용해 DP를 설명함(23개 + enum factory pattern).

1) Creational DP : object 생성에 대한 DP. object 생성이나 변경이 프로그램 구조에 영향을 주지 않도록 함.
2) Structural DP : class나 object를 조합해 더 큰 구조를 만드는 DP.
3) Behavior DP : class나 object 사이의 알고리즘이나 동작, 책임 분배에 대한 DP. 한 object만으로 수행할 수 없는 작업들을, 결합도를 최소화하면서 여러 object들에게 분배하기 위한 방법임.



2. Creational DP

2.1. Singleton Pattern

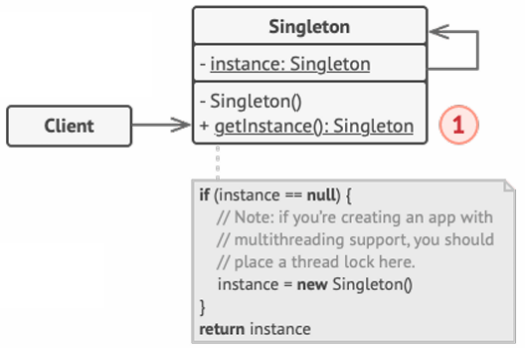
2.1.1. Singleton Pattern

1. Singleton Pattern
Singleton Pattern은 하나의 class가 오직 하나의 instance만 가질 수 있도록 하는 DP임. 이때 singleton은 하나의 원소만 가지는 집합을 말함.

주로 어떤 object에 대한 생성이 리소스를 많이 사용하거나, 전역적으로 하나만 존재해야 하는 object에 대해 사용함. 즉, 하나의 object만 만들고 각 부분에서 해당 object를 가져다가 사용함.

DB 연결(connection pool), 파일 처리, thread pool, logging 등이 사용함. 실제로 java.lang.Runtime과 java.util.logging은 singleton pattern으로 구현되어 있음.

2. Singleton Pattern의 구조
singleton pattern은 기본적으로 다음과 같이 생성자를 private static으로 하고, 해당 class의 object를 반환하는 static 메소드를 가지는 구조를 가짐.



3. Singleton Pattern의 장단점
singleton pattern을 사용하면 object를 하나만 생성해 전역적으로 활용하여 비용과 리소스를 절약하고, 직관적으로 object를 활용할 수 있음.

하지만 singleton pattern의 사용은 SRP(class 자체의 책임, object 생성에 대한 책임 존재), OCP(항상 단일 object만을 생성하고 상속 불가), DIP(interface가 아니라 object 자체와 의존하게 됨)를 위배함. 또한 모듈 간 의존성이 높아지고, 이에 따라 단위 테스트가 어려움.

singleton pattern은 여러 효율성을 확보하는 데에 도움이 될 수 있지만, SOLID를 적절히 준수하지 못할 수 있고 유연성이 떨어짐. singleton pattern을 남용하는 것은 anti pattern이 될 수 있고, 이에 따라 해당 DP를 적용할 때는 충분한 검토가 필요함.

2.1.2. Singleton Pattern의 구현

singleton pattern은 다음과 같이 각각 장단점을 가지는 다양한 구현 방식이 존재함. 특히 single thread의 경우 구현이 굉장히 단순하지만, multi-thread를 고려하면 여러 구현이 존재하게 됨.

일반적으로 권장되는 방법은, 성능이 중요한 환경에서는 lazy holder를 사용하고, 안전성이 중요한 환경에서는 enum method를 사용하는 것임. 물론 상황에 따라 적절한 방법을 선택하면 됨.

1. Lazy Initialization
singleton pattern의 표현을 그대로 구현한 방식임. getInstance()에 대한 첫 호출 시에 object가 생성되므로 lazy initialization이다.

single thread 상황에서는 문제가 없지만, multi thread 상황에서는 synchronization이 적용되지 않아 여러 object가 생성될 위험이 있다.

```
1     public class Settings {
2         private static Settings instance;
3         private Settings();
4         public static Settings getInstance {
5             if (instance == null) {
6                 return new Settings();
7             }
8             return instance;
9         }
10    }
11    ...
12    Settings settings = Settings.getInstance();
```

2. Thread-safe Initialization
lazy initialization에서 getInstance() 메소드에 synchronized를 붙이는 방식임.

해당 메소드에 대한 synchronization을 적용하여 multi-thread 환경에서도 문제가 없지만, synchronization overhead가 존재함.

3. Eager Initialization
참조 변수를 final로 지정해서 선언 시에 object를 생성하는 방식임.

java에서 final은 선언문 또는 생성자에서 딱 한 번만 할당하고 이후에는 할당이 불가능하기 때문에 multi-thread에서도 thead-safe함. 하지만 해당 object에 대한 사용이 시작되기 전에도 자원을 먹고 있으므로 비효율적임.

```
    public class Settings {
        private static final Settings instance = new Settings();
        private Settings();
        public static Settings getInstance {
            return instance;
        }
    }
    ...
    Settings settings = Settings.getInstance();
```

4. Double-checked Locking

getInstance() 메소드가 아니라 instance가 null인 조건문 내부에만 synchronized를 사용해 synchronization overhead를 줄인 방식임. 이때 바깥쪽 null 조건문은 여러 thread가 통과했을 수 있으므로, synchronization이 적용된 내부에서도 null인지 검사함.

이때 참조 변수에 volatile을 지정해야 함. multi thread인 경우 다른 thread의 cache에는 참조 변수의 값이 null로 남아있을 수 있으므로, volatile을 붙여줘야 caching에 조건문 내부 접근을 방지할 수 있음. 또한 synchronized block에 지정하는 object(static이므로 this가 없어서 Settings.class로 지정함)는 락 획득을 시도할 class명임. synchronized block을 사용하면 각 thread는 해당 object에 대한 락 획득을 시도하고, 자신의 차례가 와서 락을 획득하면 해당 block 안에 들어갈 수 있음.

thread-safe하면서도 synchronization overhead를 줄이고, lazy하게 생성해 메모리를 효율적으로 사용할 수 있음. 하지만 코드 구성이 불편하고 이해가 어려움. JVM 1.5부터 동작함.

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

5. Lazy Holder(Bill Pugh's Solution)

nested static class를 정의하고, instance를 static final로 정의해 사용하는 방식. 해당 nested class는 외부 class의 object가 생성될 때 초기화되지 않고, 해당 class에 접근할 때 초기화된다. 또한 nested class의 멤버 변수는 static이어서 첫 사용 시에만 한 번 생성된다.

lazy initialization이고, thread-safe하며, synchronization overhead가 없고, 코드가 간결하다. 하지만 클라이언트가 직렬화/역직렬화 등을 사용하면 임의로 singleton을 파괴할 수 있다. 여기에서 그 방법까지 다루진 않는다.

```
public class Settings {
    private static class SettingsHolder {
        private static final Settings instance = new Settings();
    }
    public static Settings getInstance {
        return SettingsHolder.instance;
    }
}
...
Settings settings = Settings.getInstance();
```

6. Enum Method
enum으로 class를 정의해 해당 class의 object가 자동 할당되도록 하는 방식.

java에서 enum은 class이고, enum 내부에 식별자를 작성하면 식별자의 이름으로 해당 class의 object를 생성해서 할당함. 즉, 다음 코드에서 INSTANCE에 Settings object가 할당됨.

thread-safe하고 synchronization overhead가 없고, 코드가 간결하고, 클라이언트가 임의로 파괴할 수 없음. 하지만 lazy하지 않음.

```
public enum Settings {
    INSTANCE;

    private Settings() {};
    public static Settings getInstance {
        return INSTANCE;
    }
}
...
Settings settings = Settings.getInstance();
```

2.2. Factory Method Pattern

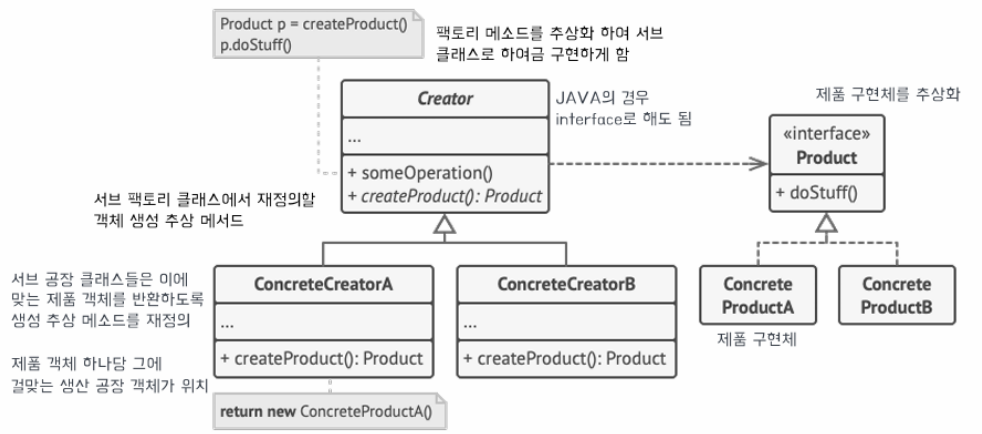
2.2.1. Factory Method Pattern

1. Factory Method Pattern
Factory Method Pattern은 object 생성 및 초기화를 subclass에 위임하도록 하는 DP임. 이를 위해 object 생성을 위한 interface를 사용하고, 구체적으로 class를 생성할지는 subclass에 의해 결정됨.

object 생성을 분리하고 확장 가능성을 높여 OCP를 확보하고자 할 때 사용함. 만약 interface를 사용하지 않고 object를 생성하는 class 하나를 사용한다면, 새로운 object 생성 기능을 추가하려면 기존 코드를 수정해야 하므로 OCP가 위배됨. factory method pattern에서는 interface를 통해 factory object와 product object 간 결합을 느슨하게 함.

XML document parser(여러 파싱 방법 제공), spring의 BeanFactory 등이 factory method pattern으로 구현되어 있음.

2. Factory Method Pattern의 구조
factory method pattern은 다음과 같이 object 생성 abstract 메소드를 가진 Creator(factory) interface(또는 abstract class)를 정의하고, 각 sub factory class들이 자신이 생성할 object에 따라 해당 메소드를 구현하도록 함. 또한 Creator는 각 product에 대한 구현체가 아니라 Product interface에 의존함.



3. Factory Method Pattern의 장단점

object 생성에 대한 interface를 사용하므로 SRP(object 생성에 대한 책임 분리), OCP(코드 변경 없이 새로운 생성 코드 추가)를 확보함.

하지만 각 product의 종류마다 class를 정의해서 사용해야 하므로 코드 복잡도가 높아지고, factory 및 product object 생성에 따른 overhead도 존재함. 뒤에서 다룰 enum factory method(subclass를 각각 구현하는 대신 enum 하나로 구현함.), dynamic factory method(reflection api를 사용해 동적으로 처리함.)는 이런 단점을 개선한 DP임.

2.2.2. Factory Method Pattern의 구현

factory method pattern은 다음과 같이 구현됨. 이후 새로운 factory class, product class 쌍이 추가되어도 기존 코드는 수정할 필요 없음. 또한 client 측에서도 ShipFactory나 Ship을 참조 변수로 해서 코드를 더 간결하게 짤 수 있음.

```
1   public interface ShipFactory {
2       default Ship orderShip(String name) {
3           Ship ship = createShip(name);
4           return ship;
5       }
6       Ship createShip(String name);
7   }
8
9   public interface Ship {
10      void setName(String name);
11  }
12
13  public class WhiteShip implements Ship {
14      private String name = "WhiteShip";
15      public void setName(String name) {
16          this.name = name;
17      }
18  }
19
20  public class WhiteShipFactory implements ShipFactory {
21      public Ship createShip(String name) {
22          WhiteShip whiteShip = new WhiteShip();
23          whiteShip.setName(name);
24          return whiteShip;
25      }
26  }
27  ...
28  Ship whiteShip = new WhiteShipFactory().orderShip("myWhiteShip");
```

2.3. Enum Factory Method Pattern

2.3.1. Enum Factory Method Pattern

1. Enum Factory Method Pattern
Enum Factory Emthod Pattern은 여러 개의 factory class를 정의하고 각각에 대한 object를 생성하는 factory method pattern의 비효율을 해결하기 위해, enum을 사용해 factory class를 정의하는 PD임.

enum을 사용해 singleton object를 생성했던 것처럼, 여러 개의 멤버를 작성해 여러 개의 instance를 관리하는 class인 Multiton을 정의할 수 있음. 이렇게 구현하면 각 factory에 대해 개별적으로 class를 작성하고 object를 생성하는 대신, 하나의 class로 묶어서 한 번만 object를 생성하고 재사용하도록 할 수 있음.

2. Enum Factory Method Pattern의 장단점
enum factory method pattern은 각 factory class를 별도로 정의하고 object를 개별적으로 생성하는 대신, 하나의 factory class를 정의하고, 해당 object를 singleton으로 생성해 활용하도록 함.

이때 새로운 factory/product를 추가하는 경우 enum factory class를 직접 수정해야 함. 즉, OCP를 위반하는데, 이는 구현의 편의성과의 tradeoff임.

또한 java에서 enum은 상속이 불가능하므로 factory가 복잡한 상속 구조를 가진다면 표현에 한계가 존재함.

2.3.2. Enum Factory Method의 구현

1. java의 enum
java에서 enum은 단순한 상수 목록이 아니라 class임. enum class를 정의하면 실제로는 다음과 같이 Enum class를 상속받는 class로 변환되고, 각 상수는 해당 class에 대한 참조 변수가 됨.

```
enum Season {
    SPRING("봄"),
    SUMMER("여름");
    // ...
}
```

실제로는 다음과 같이 변환됨.

```
class Season extends java.lang.Enum {
    public static final Season SPRING = new Season("봄");
    public static final Season SUMMER = new Season("여름");
}
```

또한 enum class는 기본적으로 abstract class이고, 내부에 abstract method를 작성하면 각 상수는 해당 메소드를 구현해야 함.

```
enum Calculator {
    PLUS {
        @Override
        double apply(double x, double y) { return x + y; }
    },
    MINUS {
        @Override
        double apply(double x, double y) { return x - y; }
    };

    // 추상 메서드 선언
    abstract double apply(double x, double y);
}
```

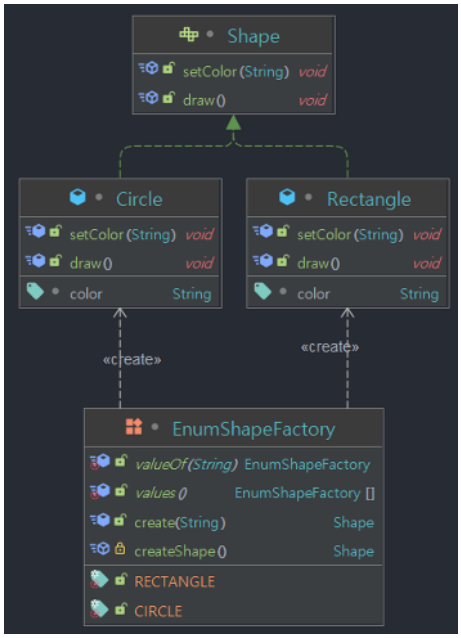
또한 enum은 다른 class를 상속할 수 없고(이미 Enum을 상속받음.), 기본적으로 final class여서 다른 class가 enum을 상속받을 수 없음. 대신 다른 interface를 enum에서 구현하도록 하는 것은 가능함.

2. Enum Factory Method의 구현

enum factory method는 다음과 같이 enum으로 구현한다.

```
    enum EnumShapeFactory {
        RECTANGLE {
            public Shape createShape() {
                return new Rectangle();
            }
        },
        CIRCLE {
            public Shape createShape() {
                return new Circle();
            }
        };

        public Shape create(String color) {
            Shape shape = createShape();
            return shape;
        }
        abstract protected Shape createShape();
    }
    ...
    EnumShapeFactory enumShapeFactory = new EnumShapeFactory();
    enumShapeFactory.RECTANGLE.createShape();
```



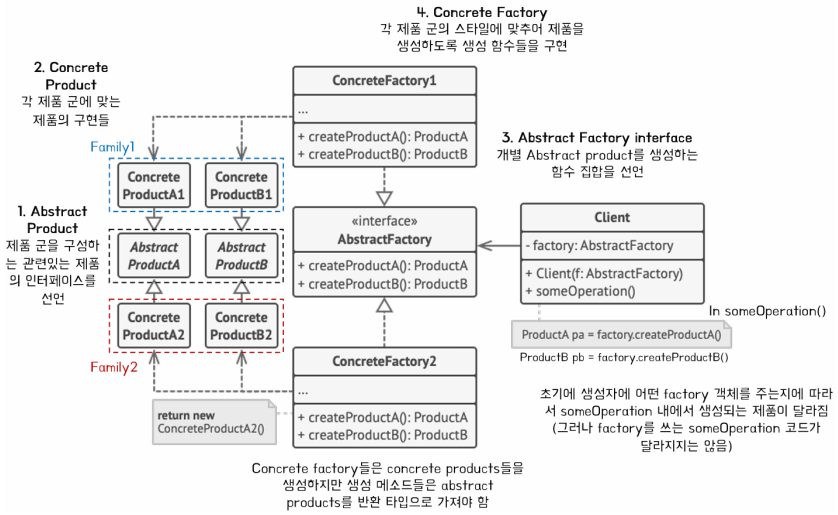
2.4. Abstract Factory Pattern

2.4.1. Abstract Factory Pattern

1. Abstract Factory Pattern
Abstract Factory Pattern은 연관 있는 product family(제품군)의 생성을 추상화한 DP임. 즉, factory method pattern이 단일 product 생성을 추상화했다면, abstract factory pattern은 여러 관련있는 product들(제품군)에 대한 생성을 추상화한 것임.

2. Abstract Factory Pattern의 구조
abstract factory pattern에서는 다음과 같이 각 product와 factory를 interface로 추상화하고, 해당되는 모든 구체적인 product를 생성하는 구체적인 factory를 사용함. factory interface는 각 product를 생성하는 개별 method를 가짐. 이에 따라 새로운 제품군과 그에 대응되는 factory가 추가되어도 기존 코드를 수정할 필요

없음.



3. Abstract Factory Pattern의 장단점

factory method pattern과 마찬가지로 SRP, OCP를 확보함. 즉, client와의 코드 결합도를 낮추고, 새로운 제품군/factory를 쉽게 추가할 수 있음.

factory method pattern과 마찬가지로 product, factory 각각에 대한 class를 모두 구현해야 하므로 코드 복잡성이 높음. 또한 각 product가 제품군을 생성하는데, 제품군에 새로운 product가 추가되면 모든 factory에 해당 product에 대한 구현이 추가되어야 함.

2.4.2. Abstract Factory Pattern의 구현

abstract factory pattern은 다음과 같이 구현할 수 있음.

```
public interface Button { void paint(); }
public interface Checkbox { void paint(); }

public interface GUIFactory {
    Button createButton();
    CheckBox createCheckbox();
}
```

```
// windows 제품군
public class WinButton implements Button {
    @Override
    public void paint() {
        System.out.println("Windows 스타일 버튼을 렌더링합니다.");
    }
}

public class WinCheckbox implements Checkbox {
    @Override
    public void paint() {
        System.out.println("Windows 스타일 체크박스를 렌더링합니다.");
    }
}
```

```java
    // windows factory
    public class WinGUIFactory implements GUIFactory {
        @Override
        public Button createButton() {
            return new WinButton();
        }

        @Override
        public Checkbox createCheckbox() {
            return new WinCheckbox();
        }
    }
    ...
    WinGUIFactory winGUIFactory = new WinGUIFactory();
    Button button = winGUIFactory.createButton();
```

2.5. Builder Pattern

2.5.1. Builder Pattern

1. Builder Pattern

Builder Pattern은 object에 대한 복잡한 생성 과정을 분리하고, client가 다양한 옵션으로 object를 생성할 수 있도록 하는 DP임.

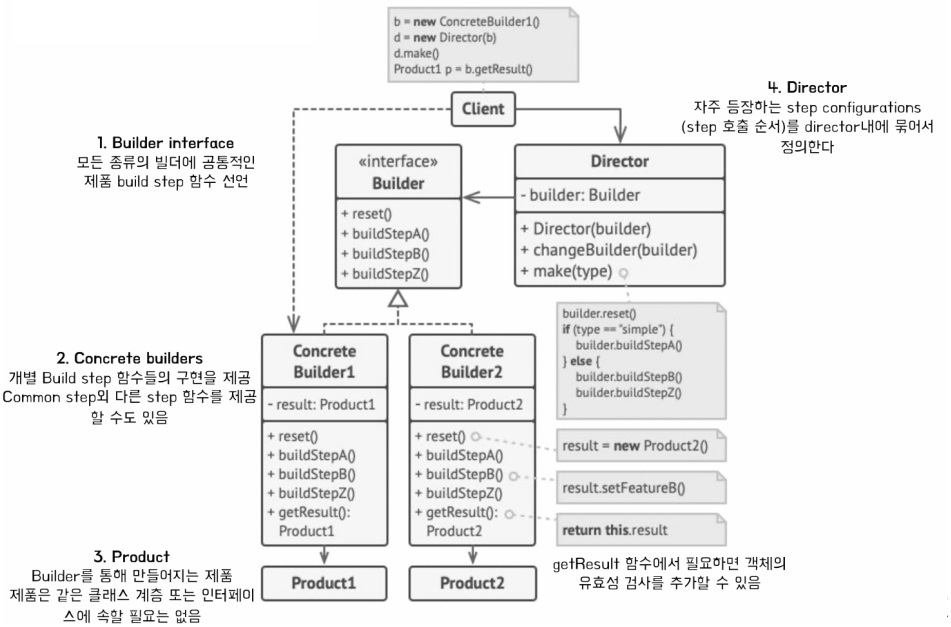
object에 따라서 생성 시에 필요한 값의 종류가 다를 수도 있고, 필요한 모든 값을 생성자에 직접 작성하는 것은 번거롭고 실수할 수 있음. builder pattern을 사용하면 필요한 작업만 직관적으로 수행해서 object를 생성할 수 있음.

java의 StringBuilder, Stream 등이 builder pattern으로 구현되어 있음.

2. Builder Pattern의 구조

builder는 다음과 같이 builder에 대한 interface와, builder 구현체, director로 구성됨. builder 구현체의 각 메소드는 자기 자신(this)을 반환하도록 해서 builder.setRoofs(4).setWalls(4).getResult() 등으로 사용이 가능하도록 함. director는 자주 사용되는 builder의 생성 step 및 config를 자동화해서 메소드로 정의하는 class임.

client는 director를 사용해 어느정도 정해진 step과 config로 object를 생성할 수도 있고, 직접 builder를 사용해 object를 생성할 수도 있음.



3. Builder Pattern의 장단점
builder pattern을 쓰면 object 생성을 분리해 OCP를 확보할 수 있고, 복잡한 생성 과정을 순차적으로 실행할 수 있음. 또한 동일한 프로세스를 통해 여러 방식으로 object를 생성할 수 있고, 불완전한 object를 사용하는 것을 방지할 수 있음.

하지만 builder object를 먼저 만들어야 사용이 가능하고, 코드 구조가 복잡해진다는 단점이 있음. 이에 따라 간단한 생성은 생성자에서 하는 게 좋음.

참고로 method chaining을 구현하는 경우 각 메소드가 적절한 참조를 반환하도록 해야 함.

2.5.2. Builder Pattern의 구현

builder pattern은 다음과 같이 구현됨.

```
public class House {
    private int roofs;
    private int walls;
    private int windows;
    // getters and setters
}

public interface Builder {
    void reset();
    Builder setRoof(int roofs);
    Builder setWalls(int walls);
    Builder setWindows(int windows);
}
```

```
public class HouseBuilder implements Builder {
    private House house;

    HouseBuilder() {
        this.house = new House();
    }

    @Override
    public void reset() {
        this.house = new House();
    }

    @Override
    Builder setRoof(int roofs) {
        this.house.setRoofs(roofs);
    }
    ...
}
```

```
public class Director {
    public void makeSimpleHouse(Bulider builder) {
        builder.reset();
        builder.setRoof(1).setWalls(4).setWindows(4);
    }
}
...
Director director = new Director();
HouseBuilder houseBuilder = new HouseBuilder();
director.makeSimpleHouse(houseBuilder);
House hosue = houseBuilder.getResult();
```

2.6. Prototype Pattern

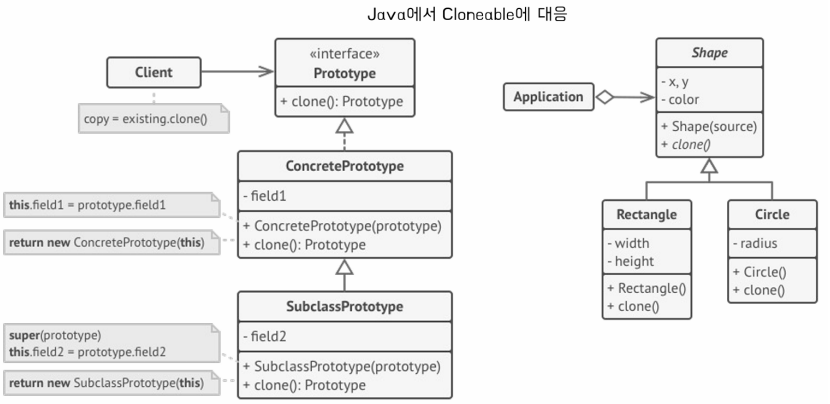
2.6.1. Prototype Pattern

1. Prototype Pattern
Prototype Pattern은 기존의 object를 복사하는 데에 사용되는 DP임.

특히 접근지정자 때문에 외부에서 직접 복사하는 것은 번거로울 수 있으므로, 각 class에서 clone() 메소드를 구현하도록 하는 것임.

java의 Cloneable interface가 prototype pattern임. Cloneable을 implements하고 clone()을 오버라이드 해야 함. 참고로 Cloneable은 멤버를 가지지 않고, clone()은 Object class의 메소드임. 대신 Cloneable은 marker interface로, Cloneable을 implements하지 않고 clone()을 오버라이드하면 CloneNotSupportedException 예외가 발생함.

2. Prototype Pattern의 구조
prototype pattern은 단순히 clone() abstract 메소드를 가지는 interface를 사용하고, clone()을 각 구현체에서 오버라이드하는 구조로 되어 있음.



3. Prototype Pattern의 장단점
prototype pattern을 사용하면 복제에 대한 책임을 분리해 SRP를 확보할 수 있고, shallow copy를 사용하면 object를 새로 생성하는 것보다 효율적임.

하지만 object 구조가 복잡한 경우 clone()을 구현하기 번거로울 수 있고, deep copy 시에 메모리가 낭비되거나 object가 관리되기 어려울 수 있음.

2.6.2. Prototype Pattern의 구현

prototype pattern은 다음과 같이 shallow copy 또는 deep copy로 구현될 수 있음. Shallow Copy에서는 멤버 변수의 값을 비트 단위로 복사하는 것으로, 이에 따라 primitive는 실제로 저장된 값이 복사되고, reference는 참조만 복사됨(동일한 object를 가리키게 됨). Deep Copy에서는 reference가 참조하는 object까지 새롭게 재귀적으로 생성하여 복사하는 것임.

특히 read-only인 경우 shallow copy를 하는 것이 좋음.

super.clone()으로 Object에 구현된 clone()을 그대로 사용하면 shallow copy가 적용되고, deep copy를 하려면 직접 오버라이드해서 구현해야 함.

```java
// shallow copy
public class Person implements Cloneable {
    private String name;
    public Objects clone() throws CloneNotSupportedException {
        return super.clone();
    }
    ...
}
```

```java
// Deep copy
public class Person implements Cloneable {
    private String name;
    public Objects clone() throws CloneNotSupportedException {
        Person person = new Person();
        person.setName(name);
        return person;
    }
    ...
}
```

3. Structural DP

3.1. Adapter Pattern

3.1.1. Adapter Pattern

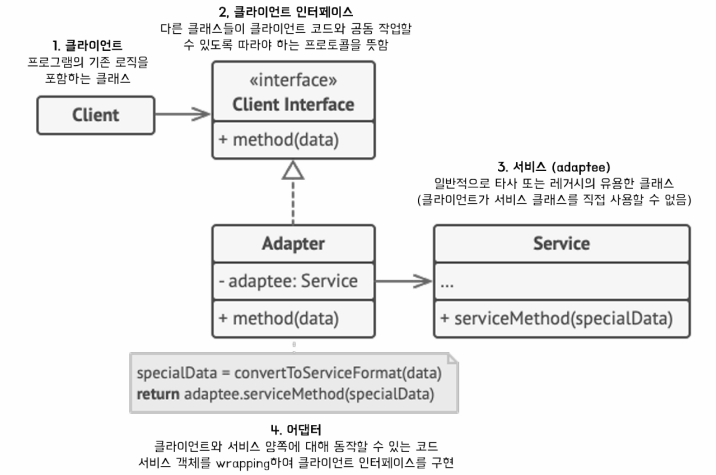
1. Adapter Pattern
Adapter Pattern은 호환되지 않는 interface를 가진 class를 사용할 수 있도록 하는 DP임. 즉, 호환되지 않는 class/interface인 adaptee를 기존 interface로 사용할 수 있도록 함.

주로 새로은 interface가 lagacy 코드와 호환되지 않을 때, 새로운 기능을 추가할 때 사용함.

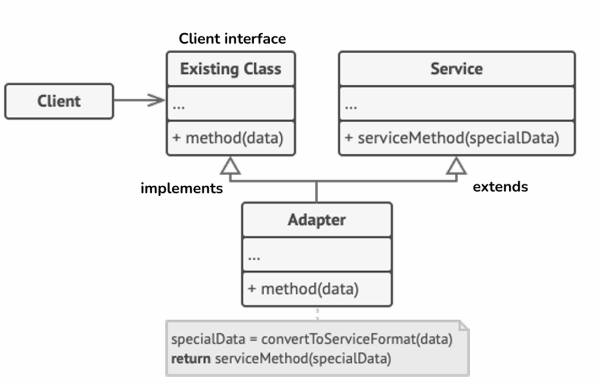
java의 Arrays.asLists(), InputStreamReader 등이 adapter pattern으로 구현되어 있음.

2. Adapter Pattern의 구조
adapter pattern은 다음과 같이 object adapter 구조와 class adapter 구조로 구분됨.

- Object Adapter 구조는 기존 interface를 구현하는 adapter class를 사용하고, adaptee를 adapter class에서 내부적으로 사용하는 composition 구조를 가짐.



- Class Adapter 구조는 기존 interface를 구현하는 adapter class를 사용하고, adaptee를 상속받도록 하는 inheritance 구조를 가짐. 특히 다중 상속을 지원하는 c++에서 사용하기 편리하고, java에서는 interface는 implements로, adaptee는 extends로 상속받는 것으로 구현할 수 있음.



3. Adapter Pattern의 장단점

구체적 로직과 interface를 구분하는 것을 전제로 하고, 기존 코드를 유지하면서 다른 코드를 사용할 수 있도록 하므로 SRP와 OCP를 준수함. 또한 새로운 코드를 추가했을 때 오류가 나면 adapter와 adaptee만 확인하면 됨.

하지만 새로운 class를 정의해야 하므로 코드 복잡성이 조금 높아질 수 있음. 간단한 경우 adaptee의 코드를 수정해서 사용하는 것이 적절할 수 있음.

3.1.2. Adapter Pattern의 구현

object adapter는 다음과 같이 구현됨.

```
    public class Service {
        public void serviceMethod() { ... };
    }

    public interface TargetInterface {
        void targetMethod();
    }

    public class ServiceAdapter implements TargetInterface {
        Service adaptee;
        ServiceAdapter(Service adaptee) {
            this.adaptee = adaptee;
        }

        @Override
        public void targetMethod() {
            adaptee.serviceMethod();
        }
    }
```

class adapter는 다음과 같이 구현됨.

```
    // Service와 TargetInterface는 동일

    public class ServiceAdapter extends Service implements TargetInterface {
        @Override
        public void targetMethod() {
            serviceMethod();
        }
    }
```

3.2. Bridge Pattern

3.2.1. Bridge Pattern

1. Bridge Pattern

Bridge Pattern은 큰 class 또는 밀접하게 연관된 class들의 집합을 두 개의 개별 계층 구조로 나눠 각각 개발하도록 하는 DP임. 즉, inheritance 구조를 composition 구조로 나타내는 것임.

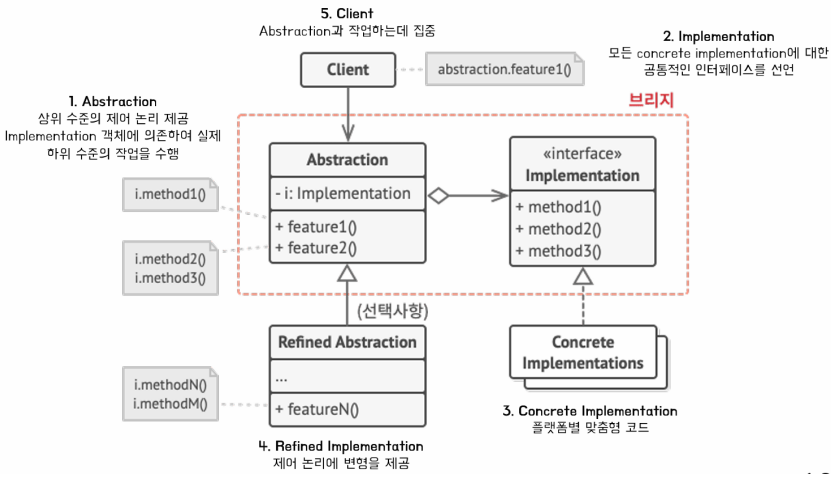
예를 들어 shape과 color라는 개념이 있을 때, 각각에 대한 class를 정의한다면 shape 개수 × color 개수 만큼의 class들이 필요함. 이는 class를 inheritance로 구현하려고 해서인데, 대신 후술할 것처럼 shape을 abstraction으로, color를 implementation으로 정의하고 shape에서 color에 대한 reference field를 사용할 수 있음. 이러면 shape 개수 + color 개수로 class가 줄어듦.

bridge pattern은 adapter pattern과 구조적으로 유사해 보일 수 있지만, bridge pattern은 설계 단계에서 다양한 부분을 독립적으로 개발하기 위해 사용되는 DP이고, adpater pattern은 유지보수 단계에서 호환되지 않는 interface를 사용하려고 할 때 사용되는 DP임.

JDBC 등이 bridge pattern으로 구현되어 있음. 이에 따라 JDBC는 DB 벤더에 상관없이 코드를 작성할 수 있도록 함.

2. Bridge Pattern의 구조

bridge pattern은 다음과 같이 상위 수준의 제어/기능을 포함하고 있는 Abstraction class와, 각 기능에 대한 구현을 포함하고 있는 Implementation class/interface로 구성됨. abstraction은 implementation의 object를 사용해 자신의 기능을 구성함(composition). abstraction은 실제 구현을 implementation에 위임함. 이에 따라 코드 수정이나 버그에 독립적으로 대응할 수 있음.



3. Bridge Pattern의 장단점

bridge pattern은 abstraction과 implementation을 분리하고, 새로운 개념이 추가되었을 때 독립적으로 코드를 추가할 수 있으므로 SRP와 OCP를 준수한다.

하지만 계층적 구조에 의해 코드 복잡도가 증가하고, 설계 구조를 알고 있는 상태가 아니면 코드 분석이 어려울 수 있음.

3.2.2. Bridge Pattern의 구현

bridge pattern은 다음과 같이 구현될 수 있음. 이때 BasicRemote가 abstraction, Device가 implementation, Radio가 implementation의 구현체임.

```
public interface Device {
    void setVolume();
    void getVolume();
}

public class Radio implements Device {
    private int volume = 30;
    @Override
    void setVolume() { ... };

    @Override
    void getVolume() { ... };
}
```

```
public class BasicRemote {
    protected Device device;

    public BasicRemote(Device device) {
        this.device = device;
    }

    public void volumeDown() {
        device.setVolume(device.getVolume - 10);

    }
    public void volumUp() {
        device.setVolume(device.getVolume + 10);
    }
}
...
Radio radio = new Radio();
BasicRemote basicRemote = new BasicRemote(radio);
basicRemote.volumeDown();
```

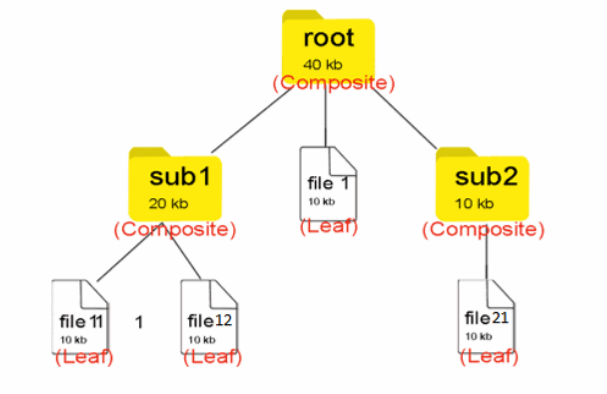
3.3. Composite Pattern

3.3.1. Composite Pattern

1. Composite Pattern

Composite Pattern은 전체-부분 관계의 트리 형태로 표현되는 복합 object들을 단일 object로 묶어서 표현할 수 있도록 하는 DP임. 즉, composite pattern을 사용하면 단일 object와 복합 object를 하나의 interface로 처리할 수 있음. 이때 단일 object를 leaf, 복합 object를 composite이라고 함.

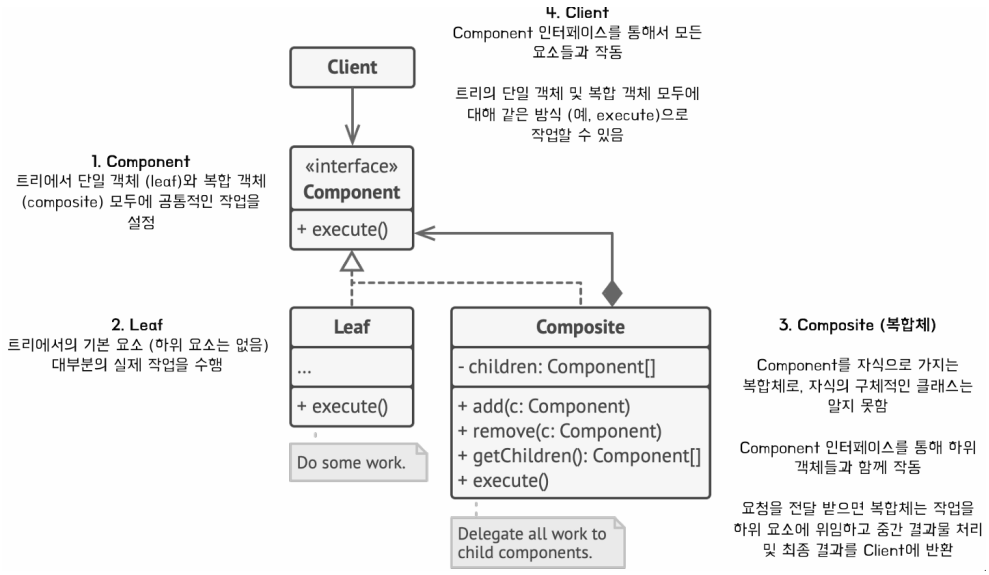
leaf와 composite을 별도의 class로 처리하려고 하면, 트리를 순회하며 노드에 도착할 때마다 타입 검사를 하는 불편함이 생기고, 코드도 깔끔하지 않음. composite pattern을 사용해 재귀적 구조를 편하게 정의하고, 트리를 구성하는 object들의 구체적인 타입을 확인하지 않아도 작업이 가능하도록 할 수 있음. 이에 따라 단일-복합 object의 트리 관계를 단순화해서 균일하게 처리할 수 있음.



java의 Swing 등이 composite pattern을 사용해 구현되어 있다.

2. Composite Pattern의 구조

composite pattern은 다음과 같이 하나의 interface로 leaf와 composite 구현체를 사용할 수 있는 구조를 가짐. 또한 composite는 Component에 대한 리스트를 멤버로 가지고 있어, 다른 composite이나 leaf를 포함할 수 있음.



3. Composite Pattern의 장단점

단일-복합 object의 트리 관계를 단순화할 수 있고, 새로운 leaf class가 추가되더라도 다른 코드를 수정할 필요가 없으므로 OCP를 준수함.

하지만 재귀적인 처리 특성상 트리가 깊어지면 디버깅이 어려움. 또한 기능이 너무 다른 class들 간에는 공통 interface 설계가 어려움.

3.3.2. Composite Pattern의 구현

composite pattern은 다음과 같이 구현할 수 있음.

```java
public interface Component {
    int getPrice();
}

public class Item implements Component {
    private int price;
    public Item(int price) {
        this.price = price;
    }
    @Override
    public int getPrice() {
        return this.price;
    }
}
```

```java
public class Composite implements Component {
    List<Component> components;
    public Composite() {
        this.components = new ArrayList<>();
    }
    public void add(Component component) {
        this.components.add(component);
    }
    @Override
    public int getPrice() {
        int totalPrice;
        for(Component component : components) totalPrice += component.getPrice();
        return totalPrice;
    }
}
```

```java
Item item1 = new Item(10);
Item item2 = new Item(20);
Item item3 = new Item(30);
Composite composite = new Composite();
Composite subComposite = new Composite();
subComposite.add(item1);
subComposite.add(item2);
composite.add(item3);
composite.add(subComposite);
System.out.println(composite.getPrice()); // 60
```

3.4. Decorator Pattern

3.4.1. Decorator Pattern

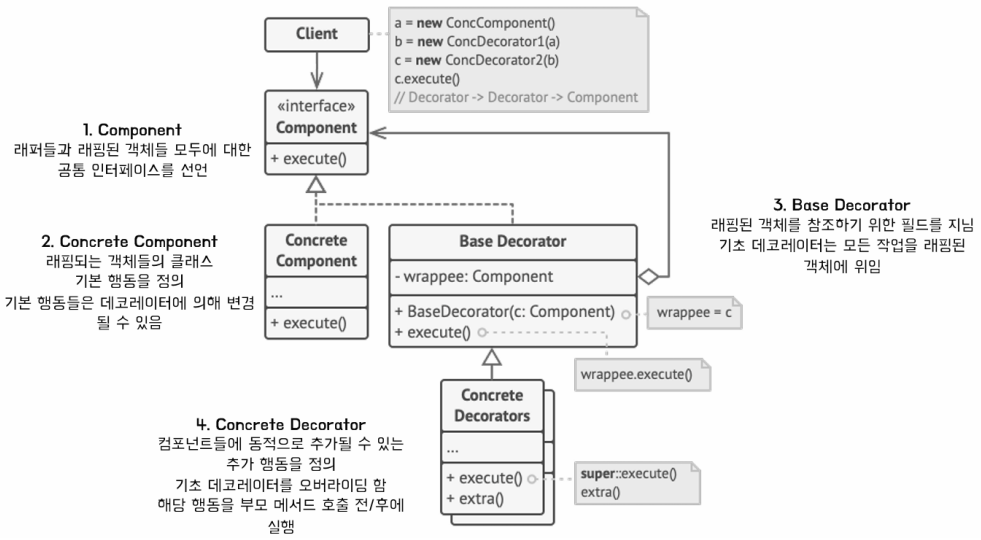
1. Decorator Pattern
Decorator Pattern은 object에 추가적인 기능을 런타임에 동적으로 추가할 수 있도록 하는 DP임. 즉, 기존 object에 추가 기능을 장식하는 것으로 이해할 수 있음.

예를 들어, SNS를 통한 알림 기능을 구현한다고 했을 때 사용자별로 알림을 받고 싶은 SNS가 전부 다르고, 2개 이상일 수도 있다고 해보자. 이를 하나의 상속 구조로 구현하려면 n개의 SNS가 있을 때 $2^n - 1$개의 class를 정의해야 함. 하지만 decorator pattern을 사용하면 하나의 decorator class만 정의해두고 사용자의 선택에 따라 동적으로 기능을 적용할 수 있음.

주로 object의 책임과 기능이 빈번하게 변경되는 경우, 새로운 class를 상속 구조에 모두 추가하는 것이 번거롭거나 불가능할 때 사용함.

java의 Stream, Collection 등이 decorator pattern으로 구현되어 있음.

2. Decorator Pattern의 구조
decorator pattern은 다음과 같이 기본 행동을 정의하고, 해당 행동에 대해 decorate하는 구조를 가짐. decorator는 자신이 wrap하는 object(wrappee)를 가리킬 멤버 변수를 가지고 있음. 이때 base decorator는 모든 작업을 wrapping된 object에 위임하고, base decorator를 상속받는 concrete decorator들이 추가 행동을 정의함. concrete decorator에서는 기존 wrappee의 기능을 그대로 호출하되(super.method()) 추가 기능을 정의함.



3. Decorator Pattern의 장단점
각 decorator class마다 책임을 가지므로 SRP를 준수하고, 새로운 코드 추가가 용이하므로 OCP를 준수함. 또한 구현체가 아니라 interface를 사용하므로 DIP도 준수함.

하지만 코드 복잡도가 높아지고, 여러 decorator로 감싸는 경우 특정 wrapper만 삭제하기 어려움. 또한 여러 decorator로 감쌌을 때 그 순서에 따라 동작에 차이가 있을 수 있고, 순서에 상관없이 동일하게 동작하도록 구현하는 것이 까다로움.

3.4.2. Decorator Pattern의 구현

decorator pattern은 다음과 같이 구현함. base decorator에서는 wrappee의 기능을 그대로 사용하도록 하고, concrete decorator에서는 wrappee의 기능을 사용하되(super.send()), 기능을 추가함.

```java
public interface Notifier {
    void send(String meesage);
}

public class BasicNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println(message);
    }
}
```

```java
    public class BaseDecorator implements Notifier {
        private Notifier wrappee;
        public BaseDecorator(Notifier wrappee) {
            this.wrappee = wrappee;
        }
        @Override
        public void send(String meesage) {
            wrappee.send(message);
        }
    }

    public class FacebookDecorator extends BaseDecorator {
        public FacebookDecorator(Notifier wrappee) {
            super(wrappee);
        }
        @Override
        public void send(String message) {
            System.out.println("[Facebook]");
            super.send(message);
        }
    }
```

3.5. Facade Pattern

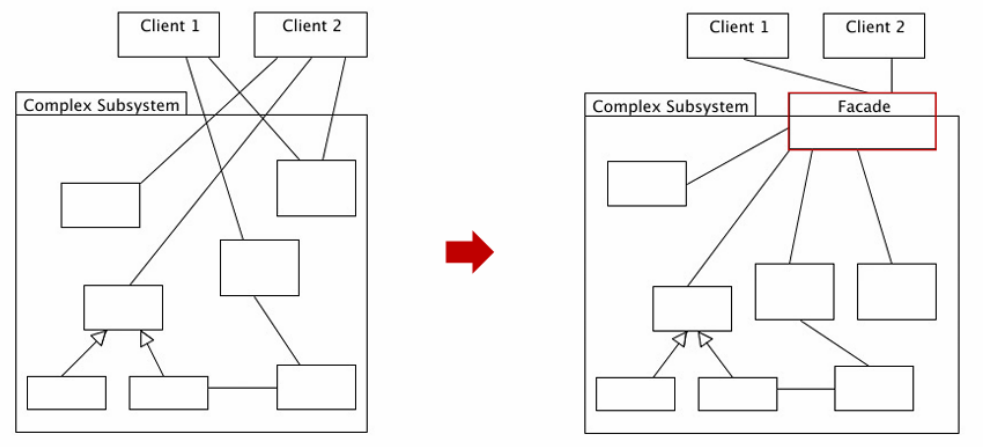
3.5.1. Facade Pattern

1. Facade Pattern

Facade(퍼사드) Pattern은 라이브러리나 서브시스템을 편리하게 사용할 수 있도록 인터페이스로서 기능하는 DP임. 즉, 추가적인 class를 줘서 디테일한 기능을 묶고 사용자가 편리하게 사용할 수 있도록 정리해서 제공하는 것임. 참고로 facade는 건물의 정면을 의미함.

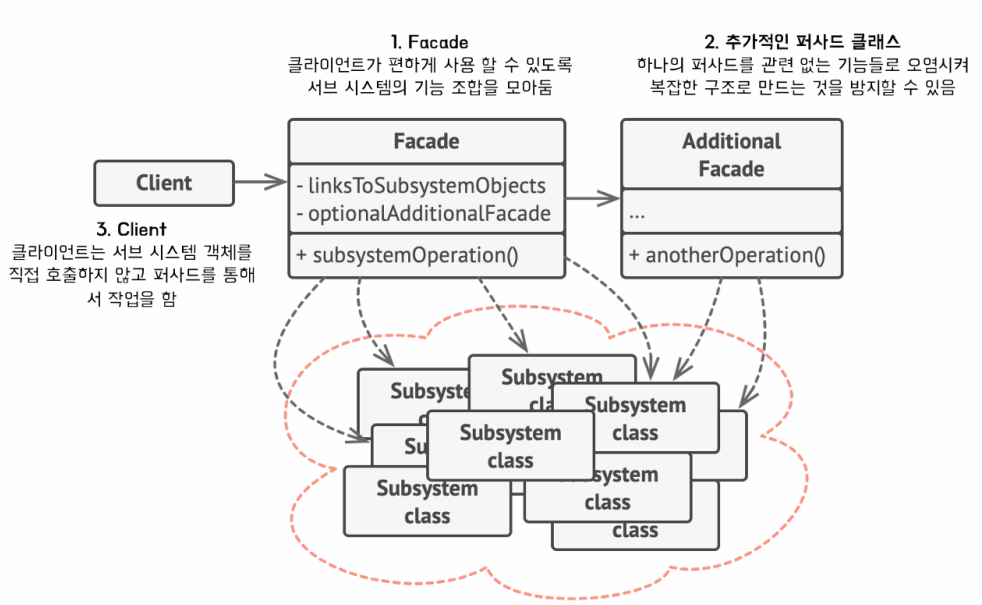
예를 들어, java에서 이메일을 보내는 코드를 작성하는 경우, client가 email API를 복잡하게 조립하며 코드를 길게 작성해야 함. facade pattern으로 각 코드를 묶어서 인터페이스를 제공하면 사용이 더 편리함.

주로 복잡한 서브시스템에 대한 의존도가 높아 이를 낮추려고 할 때, 서브시스템에 대해 제한적이고 간단한 인터페이스가 필요할 때 사용함.



2. Facade Pattern의 구조

facade pattern은 다음과 같이 subsystem class에 의존하는 facade class를 정의하는 구조를 가짐. 이때 facade class 자체도 책임에 따라 여러 class로 분리하는 것이 적절함.



3. Facade Pattern의 장단점

의존성을 줄이거나 한 곳(facade class)로 모을 수 있고, client는 facade class만 이해하면 되는 편리함을 제공할 수 있음.

하지만 facade class가 많은 class와 결합하게 되어 god object가 될 수 있고, 코드 추가에 따라 유지보수 대상이 늘어남.

God Object란 하나의 class나 object가 너무 많은 데이터와 기능을 가져서 신처럼 모든 책임을 떠안는 anti pattern으로, SRP를 위반함.

3.5.2. Facade Pattern의 구현

facade pattern은 다음과 같이, 단순히 복잡한 코드나 여러 기능을 묶은 class로 구현함. client는 EmailSender의 sendEmail(emailMessage);을 호출하는 것으로 이메일을 쉽게 작성할 수 있음.

```
public class EmailSender {
    private EmailSettings emailSettings;

    public EmailSender(EmailSettings emailSettings) {
        this.emailSettings = emailSettings;
    }

    public void sendEmail(EmailMessage emailMessage) {
        Properties properties = System.getProperties();
        properties.setProperty("mail.smtp.host", emailSettings.getHost());

        Session session = Session.getDefaultInstance(properties);

        try {
            MimeMessage message = new MimeMessage(session);
            message.setFrom(new InternetAddress(emailMessage.getFrom()));
            message.addRecipient(Message.RecipientType.TO,
                                 new InternetAddress(emailMessage.getTo()));
            message.setSubject(emailMessage.getSubject());
            message.setText(emailMessage.getText());

            Transport.send(message);
        } catch (MessagingException e) {
            e.printStackTrace();
        }
    }
}
```

복잡한 API 조합은 내부로 숨김

3.6. Flyweight Pattern

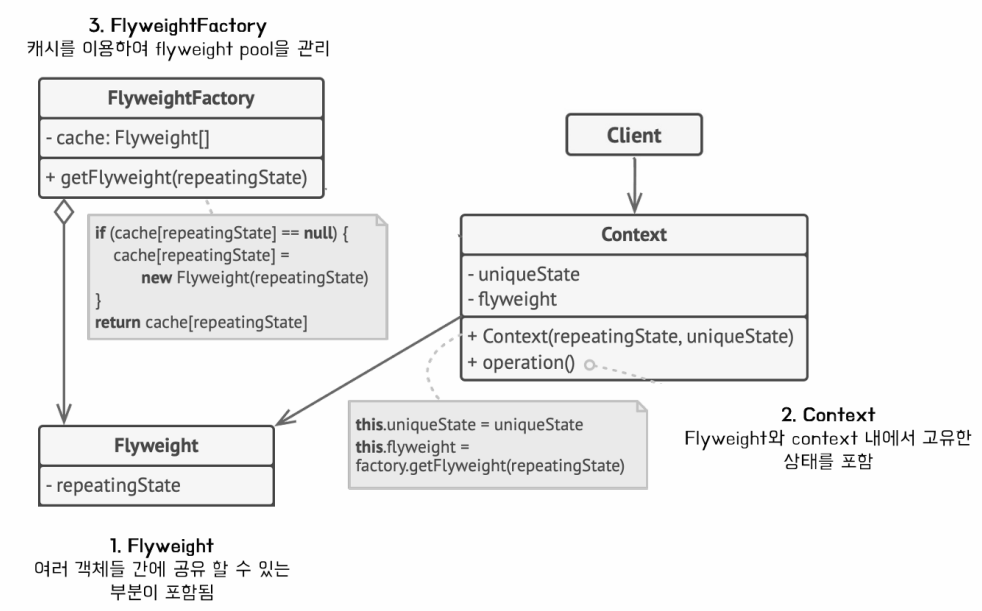
3.6.1. Flyweight Pattern

1. Flyweight Pattern
Flyweight Pattern은 메모리 사용량을 줄이기 위해 가능한 object들을 재사용하는 DP임. 즉, object에 대한 caching을 적용한 것임. 이때 변화하는 속성은 extrinsic state로, 변화하지 않는 속성은 intrinsic state라고 부르고, intrinsic state에 대해 caching을 적용함.

예를 들어, editor에서 chacracer를 표현하는 object가 있고 font는 거의 변화하지 않는다고 할 때, 모든 character object가 font에 대한 정보를 저장하고 있을 필요는 없음. flyweight pattern을 적용하면 font에 대한 부분을 따로 빼서 class를 구성하고, font object는 하나만 생성하여 여러 chrarcter object가 사용하도록 할 수 있음.

주로 메모리에 오래 상주하는 object가 많을 때, object 간에 공통적이면서 변하지 않는 부분이 많을 때 주로 사용함.

2. Flyweight Pattern의 구조
flyweight pattenr은 다음과 같이 flyweight object에 대한 pool을 관리하는(caching을 하는) flyweight factory와 flyweight class로 구성됨. flyweight class를 사용하는 class는 flywieght object를 가리키기 위한 멤버 변수를 가짐.



3. Flyweight Pattern의 장단점
flyweight pattern을 사용하면 메모리 사용량과 데이터 생성 및 저장에 따른 latency를 줄일 수 있음.

하지만 추가적 class 사용으로 코드 복잡성이 높아짐.

3.6.2. Flyweight Pattern의 구현

flyweight pattern은 다음과 같이 구현할 수 있음. 이때 flyweight class는 기본적으로 instrinsic state만을 가지고 있으므로, 멤버 변수를 final로 작성하는 것이 좋음.

```
public class Character {
    private char value;
    private Font font;
    public Character(char value, Font font) {
        this.value = value;
        this.font = font;
    }
}

public class Font {
    final String family;

    final int size;
    public Font(String family, int size) {
        this.family = family;
        this.size = size;
    }
    // getters
}
```

```
public class FontFactory {
    private Map<String, Font> cache = new HashMap<>();
    public Font getFont(String family, int size) {
    String key = family + ":" + size;
    if (!cache.containsKey(key)) {
        Font newFont = new Font(family, size);
        cache.put(key, newFont);
        System.out.println("새로운 폰트 생성: " + key);
    }
    return cache.get(key);
}

...
FontFactory fontFactory = new FontFactory();
Character c1 = new Character('A', factory.getFont("Nanum", 10));
```

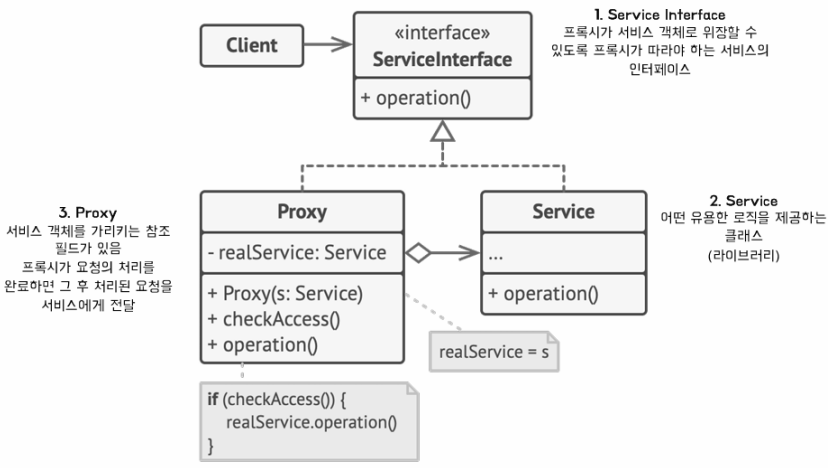
3.7. Proxy Pattern

3.7.1. Proxy Pattern

1. Proxy Pattern
Proxy Pattern은 대상 원본 object에 대한 접근을 대리하거나 제어할 수 있도록 하는 DP임. 즉, client가 원본 object에 직접 접근해서 쓰도록 하는 대신, proxy를 거쳐 사용하도록 하는 것임.

주로 원본 object가 민감한 정보를 가지고 있거나, object 크기가 커서 lazy initialization을 적용하고 싶은 경우 등에 사용함. 또한 proxy class에 cache를 둬서 필요한 정보가 caching되어 있지 않을 때에만 원본 object를 사용하도록 할 수도 있음. 원본 object를 사용하기 전에 proxy를 거치므로 데이터 유효성 검사, 로깅 등을 수행하기에도 적합함. 기능을 추가하고 싶은데 원본 class를 수정하기 어려울 때 사용하기도 함.

2. Proxy Pattern의 구조
proxy pattern은 다음과 같이 원본 object와 동일한 interface를 구현하면서 원본 object를 활용하는 구조를 가짐. 이를 위해 원본 object를 가리키는 참조 변수를 두고 활용함.



3. Proxy Pattern의 장단점

기존 코드와 기능을 유지하면서 새로운 기능을 추가할 수 있으므로 OCP를 준수하고, 기존 기능은 원본 object가 수행하고 새로운 기능은 proxy가 수행하도록 해 SRP를 준수함.

proxy 사용에 따라 코드 복잡성이 높아지고, proxy 자체가 사용하는 리소스가 많으면 비효율적이게 됨.

facade가 새로운 interface를 정의해 복잡한 시스템을 단순하게 포장한다면, proxy는 기존 interface를 사용해 원본 object로 가는 길목을 제어함.

3.7.2. Proxy Pattern의 구현

proxy pattern은 다음과 같이 구현할 수 있음.

```
1   public interface YoutubeLib {
2       Video downloadVideo(int id);
3   }
4
5   public class YoutubeClass implements YoutubeLib {
6       public Video downloadVideo(int id) {
7           Video video = new Video();
8           // 실제 다운로드 수행
9           return video;
10      }
11  }
12
13  public class ProxyYoutubeClass implements YoutubeLib {
14      YoutubeClass youtubeClass;
15      public ProxyYoutubeClass(YoutubeClass youtubeClass) {
16          this.youtubeClass = youtubeClass;
17      }
18      public Video downloadVideo(int id) {
19          // cache된 video가 있으면 해당 video 반환
20          // cache된 video가 없으면 youtubeClass.downloadVideo(...) 호출
21      }
22  }
23  ...
24  YoutubeLib youtubLib = new ProxyYoutubeCLass(new YoutubeClass());
25  youtubLib.downloadVideo(1000);
```

4. Behavior DP

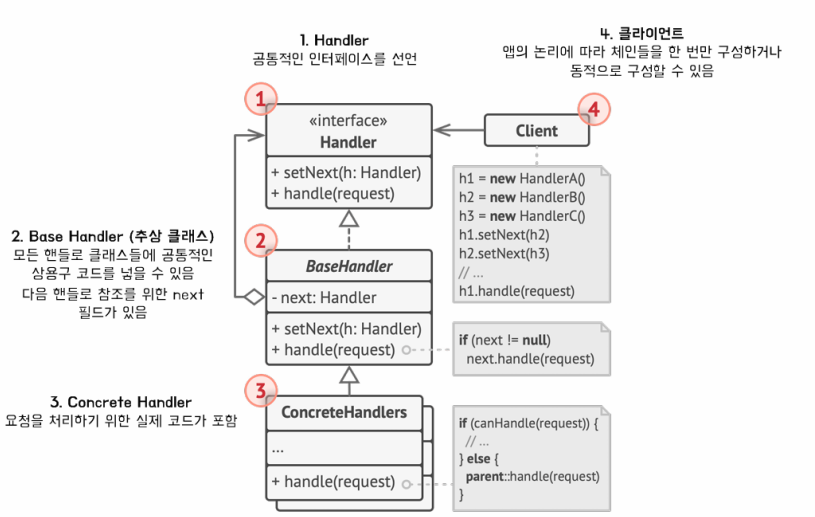
4.1. Chain of Responsibility Pattern

4.1.1. Chain of Responsibility Pattern

1. Chain of Responsibility Pattern
Chain of Responsibility Pattern(CoR)은 어떤 작업이나 client의 요청을 하나의 object에서 모두 하는 대신, 여러 개의 object에서 수행하도록 나누고 chain 형태로 연결해 연쇄적으로 처리하는 DP임. 이때 작업을 수행하는 독립 object들을 handler라고 함. 즉, handler를 chain으로 연결해서 chain을 따라 요청이 처리되도록 함.

어떤 순서에 따라 handler를 실행해야 하거나, 특정 요청을 여러 개의 object에서 처리해야 할 때 사용됨. 특히 요청을 처리할 handler들의 종류나 순서가 동적으로 결정되어야 할 때 사용됨.

2. Chain of Responsibility Pattern의 구조
CoR은 다음과 같은 구조를 가짐. base handler는 handler에 대한 공통 기능과, 다음 handler을 참조하기 위한 멤버 변수를 가지고 있는 class임. concrete handler에서는 처리가 가능하면 처리를 하고, base handler의 멤버 변수를 사용해 다음 handler에게 작업을 넘김.



3. Chain of Responsibility Pattern의 장단점
client는 chain의 내부 구조를 알 필요가 없음. 또한 기존 코드 변경 없이 새로운 handler 코드를 추가해 chain에 넣을 수 있으므로 OCP가 확보됨.

코드 가독성이 떨어지거나 연쇄적인 처리에 따른 latency가 늘어날 수 있고, chain이 잘못 구성되면 무한 루프에 빠질 수 있음.

4.1.2. Chain of Responsibility Pattern의 구현

CoR은 다음과 같이 구현할 수 있음. 이때 interface와 base handler를 별도로 정의하는 대신, Handler를 abstract class로 정의했음.

```java
public abstract class Handler {
    private Handler nextHandler = null;
    public Handler setNext(Handler nextHandler) {
        this.nextHandler = nextHandler;
        return nextHandler;

    }
    protected abstract void process(String url);

    public void run(String url) {
        process(url);

        if(nextHandler != null) {
            nextHandler.run(url);
        }
    }
}
```

```java
public class DomainHandler extends Handler {
    @Override
    protected void process(String url) {
        // url의 domain 파싱 수행
    }

}

public class ProtocolHandler extends Handler {
    @Override
    protected void process(String url) {

        // url의 protocol 파싱 수행
    }
}
```

```java
Handler handler1 = new ProtocolHandler();
Handler handler2 = new DomainHandler();
Handler handler3 = new PortHandler();
handler1.setNext(handler2).setNext(handler3);
String url1 = "http://www.youtube.com:80";

System.out.println("INPUT: " + url1);
handler1.run(url1);
```

4.2. Command Pattern

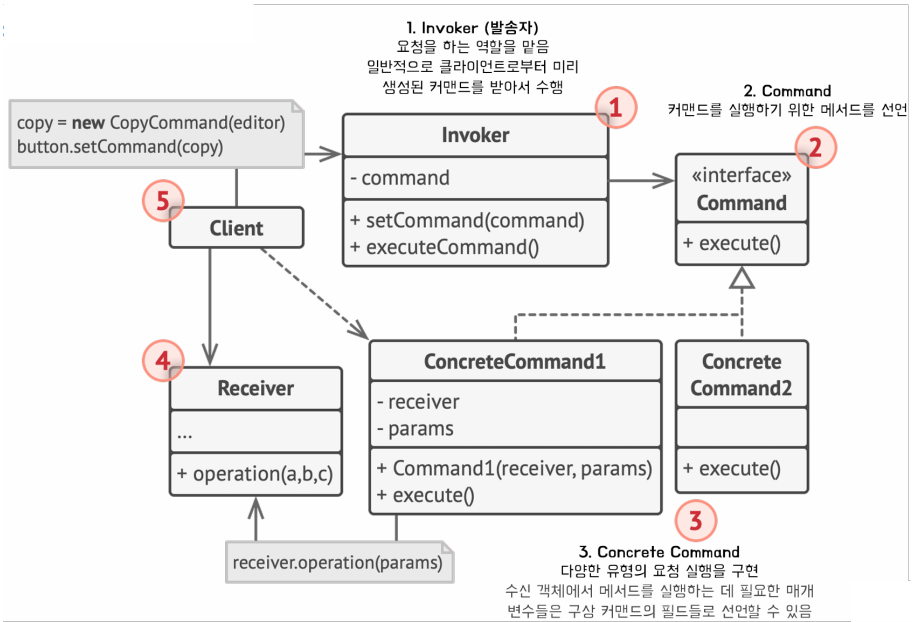
4.2.1. Command Pattern

1. Command Pattern
Command Pattern은 요청 또는 명령을 object로 encapsulate하여 요청에 대한 재사용성을 높이는 DP임. command pattern은 요청을 보내는 object인 invoker와, 요청을 받는 쪽인 receiver로 구성됨. 즉, invoker와 receiver로 class를 분리하고, 요청을 object로 감싸서 전달하는 것임.

예를 들어, 동일하게 생긴 여러 버튼이 각각 다른 작업을 수행해야 하는 경우, 단순히 상속 구조로 구현한다면 버튼별로 class를 하나씩 정의하고 오버라이딩해야 함. 이런 경우 command pattern을 사용하면 각각에 대한 class를 정의하고 메소드를 구현해 요청을 보내는 대신, button을 invoker로, 실제 작업을 수행하는 object를 receiver로 하고, command를 사용해 요청을 보내도록 할 수 있음. 즉, command가 사용자 인터페이스와 비즈니스 로직 사이의 중간 layer로서 기능함.

command pattern을 사용하면 invoker에서 로깅, undo, redo 등을 구현하기에도 편리함.

2. Command Pattern의 구조
command pattern은 다음과 같이 command를 실행시키는 invoker, 요청을 담고 있는 command, 실제로 요청에 따른 작업을 수행하는 receiver로 구성됨.



3. Command Pattern의 장단점
command pattern은 invoker와 receiver를 분리하고, 기존 코드 수정 없이 새로운 코드를 추가할 수 있으므로 SRP를 준수함.

하지만 코드 복잡도가 증가함.

4.2.2. Command Pattern의 구현

command pattern은 다음과 같이 구현할 수 있음.

```
    // receiver
    public class Light {
        private boolean isOn;
        public void on() {
        System.out.println("불을 켭니다.");
            this.isOn = true;
        }
        public void off() {
        System.out.println("불을 끕니다.");
            this.isOn = false;
        }
    }

    // Invoker
    public class Button {
        public void press(Command command) {
            command.execute();
        }
    }
```

```java
// command
public interface Command {
    void execute();
}

public class LightOnCommand implements Command {
    private Light light;
    public LightCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.on();
    }
}
...
Button button = new Button();
Light light = new Light();
button.press(new LightOnCommand(light));
```

4.3. Interpreter Pattern

4.3.1. Interpreter Pattern

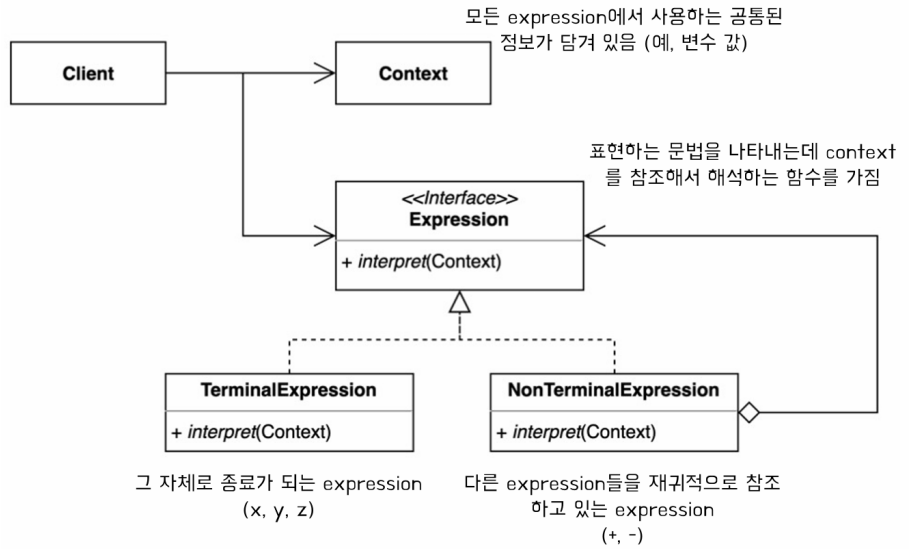
1. Interpreter Pattern
Interpreter Pattern은 특정 언어의 문법 규칙을 class로 나타내고, 해당 언어의 문장을 해석하는 경우 등에 사용되는 DP임.

이는 컴파일러, 인터프리터 등에서 사용될 수 있으며, 특히 언어에 문법을 추가하거나 변경하는 경우 유연하게 확장이 가능함.

예를 들어, postfix expression의 값을 계산하는 경우, 숫자를 terminal expression, 연산자를 nonterminal expression으로 처리해서 계산할 수 있음.

2. Interpreter Pattern의 구조
interpreter pattern은 다음과 같이 expression에서 사용하는 정보(변수의 값 등)를 담고 있는 context, interpret() 메소드(해석하는 메소드)를 포함하는 expression interface와 그 구현체들로 구성됨.

interpreter pattern에서는 언어의 문법을 object들의 tree로 나타내고, 각 object에서 재귀적으로 각각의 interpret() 메소드를 호출해 결과를 반환함. 이때 expression interface의 구현체는 leaf에 해당하는 terminal expression(숫자 등)과 leaf가 아닌 nonterminal expression(연산자 등)으로 구성됨.



3. Interpreter Pattern의 장단점

interpreter pattern은 언어에 새로운 문법 규칙을 추가하기 쉬움. 특히 단순한 문법을 가진다면 효과적임.

하지만 언어의 문법이 복잡해질수록 코드 복잡도도 높아지고, 성능이 저하될 수 있음.

4.3.2. Interpreter Pattern의 구현

다음과 같이 interpreter pattern을 사용해 덧셈과 뺄셈을 포함하는 postfix 연산을 구현할 수 있음. 이때 VariableExpression이 terminal, PlusExpression과 MinusExpression이 nontermainal임.

이때 PostfixParser는 postfix 연산에 따라, 실제로 문자열을 파싱하고 stack을 활용한 연산하는 class임.

```
    public interface PostfixExpression {
        int interpret(Map<Character, Integer> context);
    }

    public class VariableExpression implements PostfixExpression {
        private Character character;
        public VariableExpression(Character character) {
            this.character = character;
        }
        @Override
        public int interpret(Map<Character, Integer> context) {
            return context.get(this.character);
        }
    }
```

```
        public class PlusExpression implements PostfixExpression {
        private PostfixExpression left, right;
        public PlusExpression(PostfixExpression left, PostfixExpression right) {
            this.left = left;
            this.right = right;
        }
        @Override
        public int interpret(Map<Character, Integer> context) {
            return left.interpret(context) + right.interpret(context);
        }
    }
```

```java
public class PostfixParser {
    public static PostfixExpression parse(String expression) {
        Stack<PostfixExpression> stack = new Stack<>();
        for (char c : expression.toCharArray())
            stack.push(getExpression(c, stack));
        return stack.pop();
    }
    private static PostfixExpression getExpression(char c, Stack<PostfixExpression>
    ↪ stack) {
        switch (c) {
            case '+':
                return new PlusExpression(stack.pop(), stack.pop());
            case '-':
                PostfixExpression right = stack.pop();
                PostfixExpression left = stack.pop();
                return new MinusExpression(left, right);
            default:
                return new VariableExpression(c);
            }
    }
}
```

4.4. Iterator Pattern

4.4.1. Iterator Pattern

1. Iterator Pattern

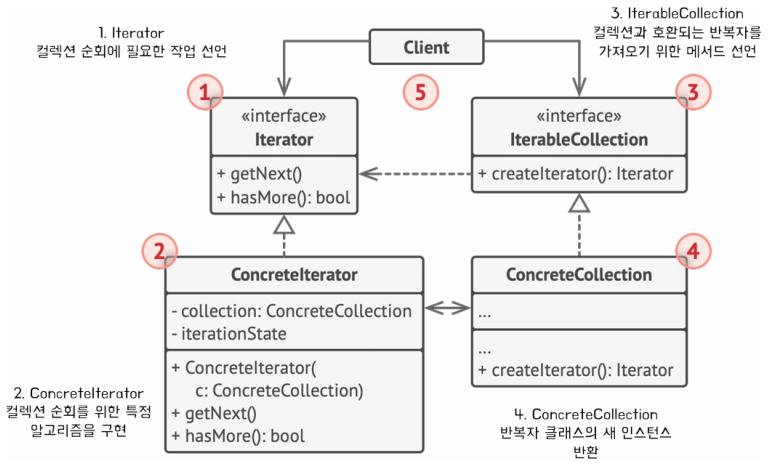
Iterator Pattern은 임의의 collection(list, tree, stack 등)에 대해 collection 내의 요소들을 하나씩 순회할 수 있도록 하는 DP임.

자료구조에 따라 선형적인 것도 있고(list, array 등), 비선형적인 것도 있는데(tree 등) 각 경우에 대해 개별적으로 순회하는 코드를 짜는 것은 불편함. iterator pattern을 사용하면 collection에 상관없이 순회하도록 할 수 있음.

java의 java.util.iterator가 iterator pattern으로 구현되어 있음. hasNext(), next(), remove()(next()로 받았던 요소 삭제), forEachRemaining()(클래스명::메소드명을 작성하면 각 요소에 대해 호출)과 같은 메소드를 지원함.

2. Iterator Pattern의 구조

iterator pattern에서는 collection을 순회하는 별도의 object인 iterator를 정의함. 이 iterator에서는 hasMore()로 다음 요소가 존재하는지 판단하고, getNext()로 다음 요소를 가져오고 index를 +1함. 이에 따라 반복문을 사용하면 쉽게 collection을 순회할 수 있음. iterator를 사용할 collection은 IterableCollection interface을 implements하고, 대응되는 iterator를 구현하면 됨.



3. Iterator Pattern의 장단점
iterator pattern을 사용하면 임의의 collection에 대해 동일한 순회 인터페이스를 제공할 수 있음. 또한 collection과 iterator를 분리하여 SRP를 준수함.

코드 복잡성이 높아짐. 사용하는 collection이 몇 가지 없거나 간단한 경우에는 anti pattern일 수 있음.

4.4.2. Iterator Pattern의 구현

iterator pattern은 다음과 같이 구현할 수 있음.

```
public interface IterableCollection {
    Iterator createIterator();
}

public interface Iterator {
    Object getNext();
    boolean hasNext();
}

public class ArraryCollection implements IterableCollection {
    private Object[] array;
    public ArraryCollection(int size) {
        this.array = new Object[size];
    }
    ...
    @Override
    public Iterator createIterator() {
        return new ArraryIterator(array);
    }
}
```

```
public class ArraryIterator implements Iterator {
    private Object[] array;
    private int index = 0;
    public ArraryIterator(Object[] arrary) {
        this.array = array;
    }
    @Override
    public Object getNext() {
        return arrary[index];
        index++;
    }
    public boolean hasNext() {
        return index < arrary.length;
    }
}

...
Iterator it = arrary.createIterator();
int sum = 0;
while(it.hasNext()) {
    sum += it.getNext();
}
```

4.5. State Pattern

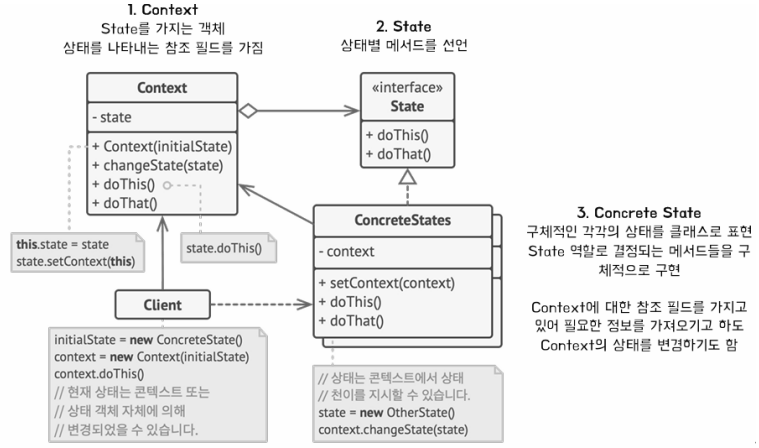
4.5.1. State Pattern

1. State Pattern
State Pattern은 object의 state(상태)를 정의하고, state에 따라 행동을 변경할 수 있도록 하는 DP임. 즉, composition으로 state를 관리하도록 함.

예를 들어, document object가 있고, state가 draft, moderation, published가 있다고 하자. 각 상태에 따라 publish() 메소드의 동작이 달라야 하는데, 이를 단순히 조건문으로 구현할 수도 있지만 가독성이 떨어짐. state pattern을 사용하면 각 state에 대한 class를 만들고, 동작을 정의할 수 있음.

이때 state object를 singleton으로 하는 게 더 좋을 수 있음.

2. State Pattern의 구조
state pattern은 state를 가지는 object를 context라 하고, context가 가지는 각 state에 대한 class를 만들어서 state에 따른 동작을 정의함. context object는 현재 state에 대응되는 state object를 참조하는 변수를 가지고 있고, 해당 object의 메소드를 호출해 작업을 수행함. 또한 state object도 context object를 참조하는 변수를 가지고 있는데, 이는 연산 이후 context의 state를 변화시키도록 할 수 있기 때문임.



3. State Pattern의 장단점
context와 state를 분리해 SRP를 확보할 수 있고, 기존 코드를 적게 수정하면서 상태를 추가할 수 있으므로 OCP가 어느정도 확보됨.

하지만 코드 복잡도가 증가하고, state 별로 class를 생성해야 하므로 state 수가 몇 가지 없거나 state가 잘 변경되지 않는 경우 비효율적임.

4.5.2. State Pattern의 구현

state pattern은 다음과 같이 구현될 수 있음.

```
    public interface PowerState {
        void pushPowerButton(LaptopContext context);
    }

    public OnState implements PowerState {
        @Override
        public void pushPowerButton(LaptopContext context) {
            System.out.println("Laptop power OFF");
            context.changeState(new OffState());
        }
    }

    public OffState implements PowerState {
        @Override
        public void pushPowerButton(LaptopContext context) {
            System.out.println("Laptop power ON");
            context.changeState(new ONState());
        }
    }
```

```
    public class LaptopContext {
        private PowerState = powerState;
        public LaptopContext() {
            this.powerState = new OffState();
        }
        public void changeState(PowerState powerState) {
            this.powerState = powerState;
        }
        public void pushPowerButton() {
            this.powerState.pushPowerButton(this);
        }
    }
    ...
    LaptopContext laptopContext = new LaptopContext();
    laptopContext.pushPowerButton();
    laptopContext.pushPowerButton();
```

4.6. Strategy Pattern

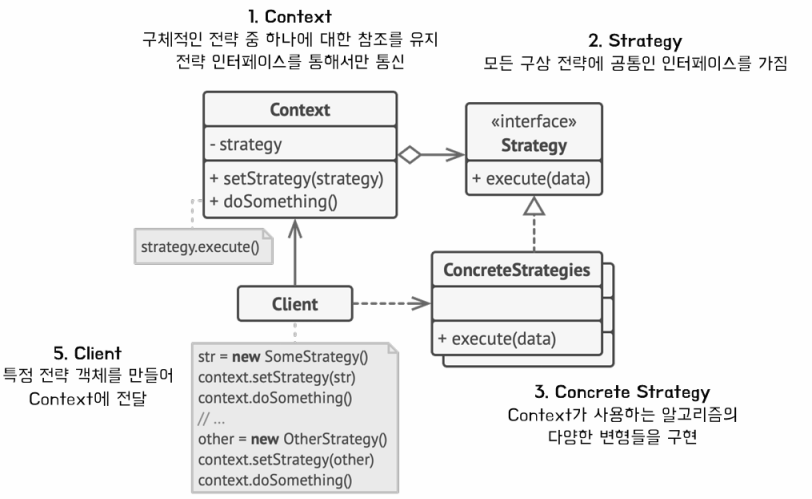
4.6.1. Strategy Pattern

1. Strategy Pattern
Strategy Pattern은 런타임에 알고리즘 전략을 동적으로 선택할 수 있도록 하는 DP임. 즉, composition으로 전략을 관리할 수 있도록 함.

예를 들어, 내비게이션 앱에서 사용자의 설정에 따라 최단경로를 찾아야 할 수도 있고, 차가 가장 안 막히는 경로를 찾아야 할 수 있음. 이때 각 전략을 class에 모두 구현하면 코드가 복잡해지고, 새로운 전략이 추가하기 어려움.

java의 Comparator가 strategy pattern으로 구현되어 있음. sort 등에서 compare하는 전략을 직접 지정하는 경우 등에 사용됨.

2. Strategy Pattern의 구조
strategy pattern은 실제로 strategy 중 하나를 사용해 작업을 수행하는 object인 context와, 각 알고리즘에 대한 object인 strategy로 구성됨. context는 한 stratgy를 참조하는 변수를 가지고, 해당 strategy의 메소드를 호출해 사용함.



3. Strategy Pattern의 장단점

strategy pattern을 사용하면 런타임에 strategy를 결정해서 적용할 수 있음. 이때 새로운 알고리즘을 기존 코드 수정 없이 추가할 수 있으므로 OCP가 준수되고, 각 strategy class는 하나의 알고리즘을 구현하므로 SRP를 준수함.

하지만 코드 복잡도가 높아지고, strategy가 몇 개 없다면 비효율적임. 또한 client가 각 알고리즘 간의 차이를 잘 이해하고 있어야 함.

strategy pattern과 state pattern은 구조가 유사하고 둘 다 composition을 활용하지만, 각각 알고리즘의 동적인 교체와 상태에 따른 행동 변화에 초점을 맞춘다는 점에서 차이가 있음. 즉, 의미적인 차이가 존재함.

4.6.2. Strategy Pattern의 구현

strategy pattern은 다음과 같이 구현할 수 있음.

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

// strategy
public class MasterCardStrategy implements PaymentStrategy {
    @Override
    public void pay(int price) {
        System.out.println(price + " Won paid using MasterCard");
    }
}
```

```java
// context
public class ShoppingCart {
    PaymentStrategy paymentStrategy;
    int price = 0;
    public void addPrice(int price) {
        this.price += price;
    }
    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }
    public void pay(int price) {
        this.paymentStrategy.pay(price);
    }
}
...
ShoppingCart shoppingCart = new ShoppingCart();
shoppingCart.setPaymentStrategy(new MasterCardStrategy());
shoppingCart.pay();
```

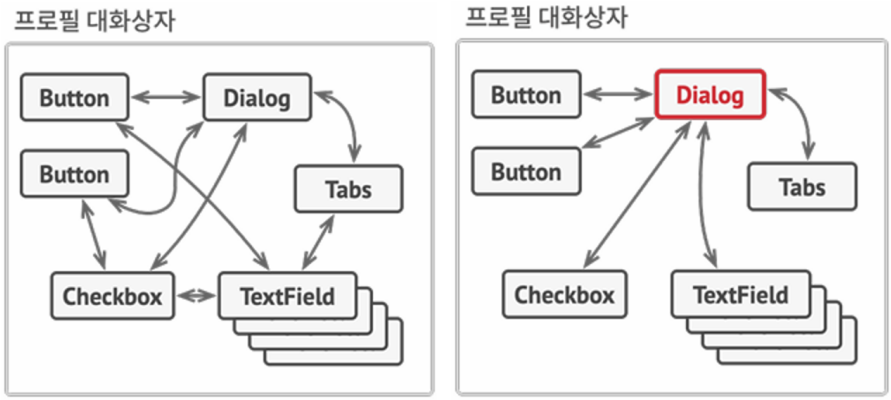
4.7. Mediator Pattern

4.7.1. Mediator Pattern

1. Mediator Pattern

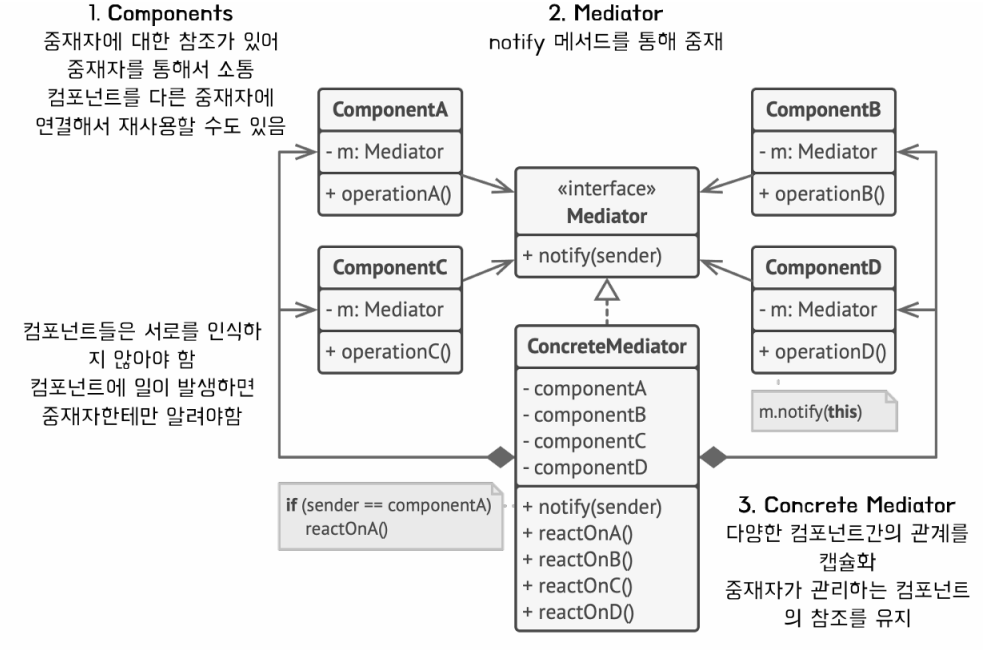
Mediator(중재자) Pattern은 object 간의 통신을 제한하고, mediator object를 통해서만 통신하도록 하는 DP임.

예를 들어, 어떤 다이얼로그에 여러 버튼이나 화면들이 있을 때, 각 버튼 클릭에 따른 화면 변화를 구현하려면 각 object가 필요한 object를 참조하도록 할 수 있음. 하지만 버튼과 화면들이 너무 많아지면 object 간의 참조 또한 너무 많아지고, 결합도가 높아지게 됨. mediator pattern을 사용하면 이런 M:N 구조를 M:1의 관계로 전환할 수 있음.



2. Mediator Pattern의 구조

mediator pattern은 다음과 같이 통신을 주고받아야 하는 object들인 component와, 통신을 중재하는 object인 mediator로 구성됨. component들은 mediator를 참조하는 변수를 가지고 있고, concrete mediator는 각 component를 참조하는 변수를 가지고 있음.



3. Mediator Pattern의 장단점

object 간의 결합도를 낮추고, 코드 이해 및 유지보수를 쉽게 함. 또한 새로운 component 및 mediator에 대한 추가를 비교적 쉽게 할 수 있음.

하지만 mediator가 god object가 될 수 있음.

mediator pattern은 facade pattern과 유사한 부분이 있지만, facade pattern은 서브시스템에 대한 단순화된 인터페이스를 정의하고 서브시스템은 facade에 의한 interface를 인식하지 못함. 반면 mediator pattern은 component 간의 통신을 하나로 묶고, 서브시스템이 mediator를 직접 활용함.

4.7.2. Mediator Pattern의 구현

mediator pattern은 다음과 같이 구현될 수 있음.

```
    public interface Mediator {
        void sendMessage(Component sender, String message);
        void addComponent(Component component);
    }

    public class Component {
        private String name;
        private Mediator mediator;
        public Component(String name, Mediator mediator) {
            this.name = name;
            this.mediator = mediator;
        }
        public void receive(String message) {
            System.out.println("[Received to " + name + "] " + message);
        }
        public void send(String message) {
            this.mediator.sendMessage(this, message);
        }
    }
```

```java
public class ConcreteMediator implements Mediator {
    private List<Component> componentList;
    public ConcreteMediator() {
        this.componentList = new ArrayList<>();
    }
    @Override
    public void addComponent(Component component) {
        this.componentList.add(component);
    }
    @Override
    void sendMessage(Component sender, String message) {
        for(Component c : componentList) {
            if(c != sender) {
                c.receive(message);
            }
        }
    }
}
```

```java
ConcreteMediator concreteMediator = new ConcreteMediator();
Component component1 = new Component("c1", concreteMediator);
Component component2 = new Component("c2", concreteMediator);
concreteMediator.addComponent(component1);
concreteMediator.addComponent(component2);
component1.send("Hello~~ World~~!");
```

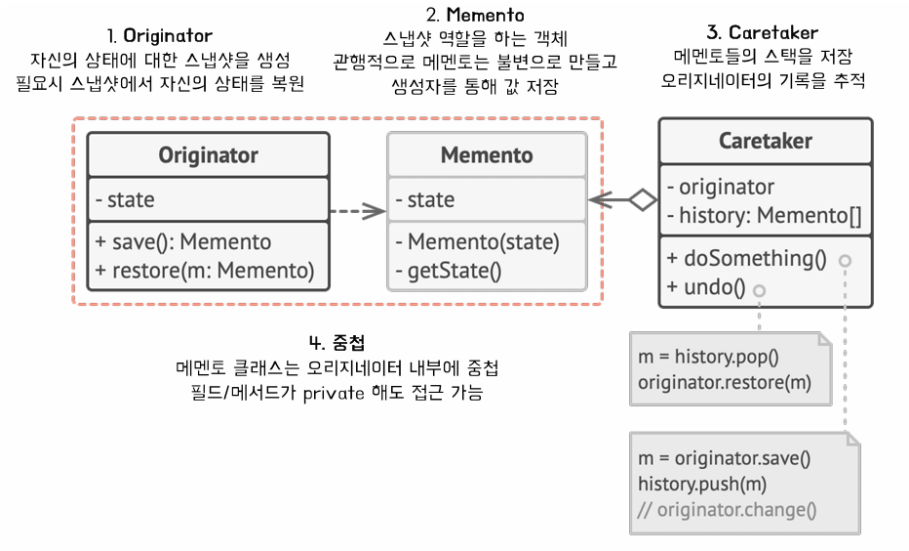
4.8. Memento Pattern

4.8.1. Memento Pattern

1. Memento Pattern
Memento Pattern 또는 Snapshot Pattern은 object의 세부 사항을 공개하지 않으면서 해당 object의 이전 상태 값을 저장하고 복원할 수 있도록 하는 DP임.

예를 들어, 텍스트 편집기에서 텍스트 입력 중에 실행 취소를 하려고 하면, 특점 시점의 object의 모든 상태를 그대로 불러올 수 있어야 함. 하지만 대부분의 object는 중요한 정보를 encapsulation하기도 하고, 해당 class에 변화가 생기면 함께 수정해줘야 하므로, object 외부에서 직접 getter를 사용해 값을 복사하는 것은 적절하지 않음. memento pattern을 사용하면 이런 문제를 해결할 수 있음.

2. Memento Pattern의 구조
memento pattern은 다음과 같이 원본 object에 해당하는 originator와, originator의 상태를 저장하는 object인 memento로 구성됨. 이때 memento class는 originator class의 nested class로, memento object는 originator object의 private 멤버에 접근이 가능함. 또한 memento object들을 관리하는 caretaker object를 사용함.



3. Memento Pattern의 장단점

memento pattern을 사용하면 encapsulation을 유지하면서 snapshot을 저장할 수 있음. 또한 originator와 caretaker를 분리하여 SRP를 달성함.

하지만 snapshot을 너무 많이 생성하면 메모리가 낭비될 수 있음. caretaker에 snapshot 삭제 및 관리 기능을 구현하기도 함.

4.8.2. Memento Pattern의 구현

memento pattern은 다음과 같이 구현할 수 있음. 이때 memento class의 멤버 변수는 final로, 값은 생성자로 지정하는 것이 관행임. 참고로 nested class에서 바깥쪽 멤버 변수는 Editor.this.text와 같이 접근할 수 있음.

```
    public class Editor {
        private String text;
        // setter
        public Snapshot create() {
            return new Snapshot();
        }
        public void restore(Snapeshot snapshot) {
            this.text = snapshot.getText();
        }

        final class Snapshot {
            private final String text;
            public Snapshot() {
                this.text = Editor.this.text;
            }
            // getter
        }
    }
```

```
    public class Command {
        private Stack<Editor.Snapshot> stack;
        public Command() {
            stack = new Stack<>();
        }
        public void makeBackup(Editor editor) {
            Editor.Snapshot snapshot = editor.create();
            stack.push(snapshot);
        }
        public Editor undo(Editor editor) {
            Editor.Snapshot snapshot = stack.pop();
            editor.restore(snapshot);
            return editor;
        }
    }
    ...
    Editor editor = new Editor();
    Command command = new Command();
    editor.setText("1");
    command.makeBackup(editor);
    editor.setText("2");
    Editor oldEditor = command.undo(editior); // 1
```

4.9. Observer Pattern

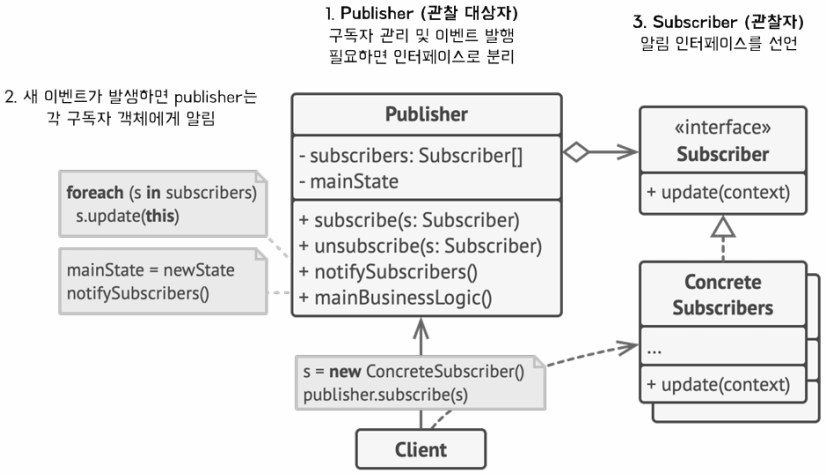
4.9.1. Observer Pattern

1. Observer Pattern

Observer Pattern 또는 발행-구독 모델은 관찰하고 있는 대상의 상태가 변했을 때 모든 subscriber에게 그 정보를 알리는 DP임. 즉, 1:N 관계로 구성되어 subscriber들을 저장해놨다가 정보를 알리도록 하는 메소드가 호출되면 모든 subscriber 각각에 대응되는 메소드를 호출해줌.

2. Observer Pattern의 구조

observer pattern은 다음과 같이 subscriber를 관리하고 정보를 전달하는 publisher와, 정보를 받는 subscriber로 구성됨. publisher는 subscriber에 대한 list를 가지고 있고, notify 시에는 각 subscriber의 update()를 호출함. 또한 publisher의 멤버 변수에 전달할 값을 저장해두고, update 시에 publisher의 참조를 넘기는 것으로 구현할 수 있음.



3. Observer Pattern의 장단점

observer pattern을 사용하면 값의 변경을 쉽게 감시하고 값을 전달할 수 있음. 또한 새로운 subscriber를 기존 코드 수정 없이 추가할 수 있으므로 OCP를 준수함. 런타임에 subscriber를 등록할 수 있다는 장점도 있음.

하지만 subscriber에서 알림을 전달받는 순서를 제어할 수 없고, 단순 통보되는 구조임. 또한 다수의 observer와 subscriber를 등록하고 자주 사용하지 않는 경우 메모리 낭비가 발생할 수 있음.

4.9.2. Observer Pattern의 구현

observer pattern은 다음과 같이 구현됨.

```
public interface Subscriber {
    void display(WeatherAPI api);
}

public class KoreanUser implements Subscriber {
    public void display(WeatherAPI api) {
        System.out.println("Korean user received : " + api.getTemperature);
    }
}

public interface Publisher {
    void register(Subscriber subscriber);
    void remove(Subscriber subscriber);
    void notifySubscribers();
}
```

```
public class WeatherAPI implements Publisher {
    private List<Subscriber> subscriberList;
    private int temperature;
    // getter&setter
    public WeatherAPI() {
        this.subscriberList = new ArrayList<>();
    }
    @Override
    public void register(Subscriber subscriber) {
        this.subscriberList.add(subscriber);
    }
    @Override
    public void remove(Subscriber subscriber) {
        this.subscriberList.remove(subscriber);
    }
    @Override
    public void notifySubscribers() {
        for(Subscriber subscriber : subscriberList) {
            subscriber.display(this);
        }
    }
}
```

```
WeatherAPI api = new WeatherAPI();
KoreanUser user = new KoreanUser();
api.register(user);
api.notifySubscribers();
```

4.10. Template Method Pattern

4.10.1. Template Method Pattern

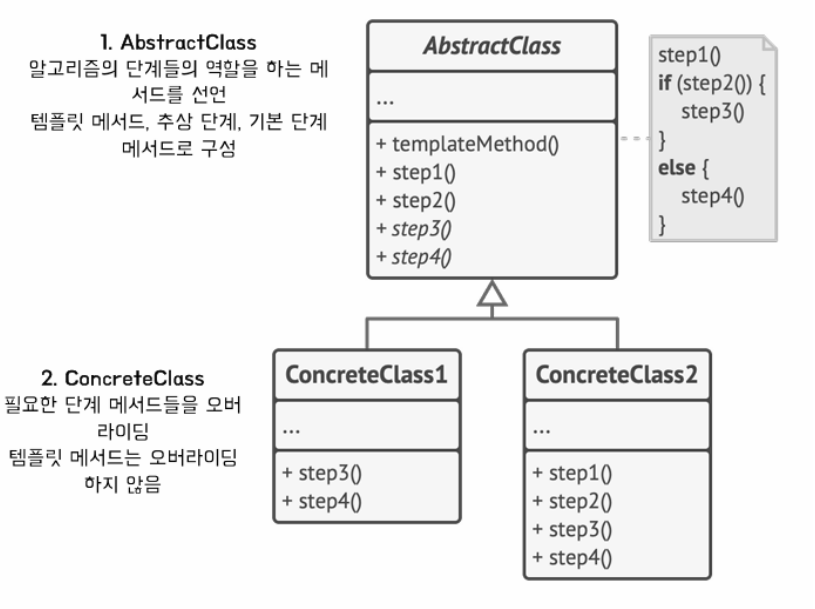
1. Template Method Pattern
Template Method Pattern은 알고리즘의 구조(뼈대)를 단계별로 정의하고, 일부 단계를 subclass에서 오버라이드해서 구현할 수 있도록 하는 DP임. 즉, 여러 class에서 공통으로 사용하는 메소드를 템플릿화하고, 각자

다른 부분만 구현하도록 하는 것임. 이에 따라 알고리즘의 구조는 유지한 채로 일부 동작만 다르게 정의할 수 있음.

예를 들어, 동일한 작업을 수행하는 알고리즘이 있는데, 여러 데이터의 포맷에 호환되도록 하려고 한다고 하자. 이는 데이터 포맷별로 별도의 메소드나 class를 정의해 구현할 수도 있지만, template method pattern을 사용하면 중복되는 부분은 템플릿화해서 사용하고, 다른 부분만 개별적으로 구현할 수 있음.

2. Template Method Pattern의 구조
template method pattern은 다음과 같이 각 단계 메소드를 호출해 알고리즘의 구조를 정의하는 하나의 템플릿 메소드와, 각 단계의 작업을 정의하는 여러 개의 단계 메소드로 구성됨. 이때 단계 메서드 중 abstract 메소드는 추상 단계 메소드, default 메소드는 디폴트 단계 메소드라고 함. 해당 abstract class를 상속받는 class에서는 모든 추상 단계 메소드를 구현해야 하고, 디폴트 단계 메소드는 필요 시 구현하고, 템플릿 메소드는 오버라이드해서는 안 됨(final 처리함.).

또한 abstract class에서 정의하는 body가 비어 있거나 기본 동작만을 가지고 있는 단계 메소드인 Hook 메소드도 사용할 수 있음. 템플릿 메소드는 hook 메소드가 오버라이드 되지 않아도 작동하며, 이는 주로 알고리즘의 전/후로 배치되어 사용자에게 추가 확장 지점을 제공함.



3. Template Method Pattern의 장단점
template method pattern을 사용하면 코드 중복을 줄일 수 있고, client는 필요한 부분만 재정의하여 알고리즘을 사용할 수 있음. 또한 핵심 로직을 상위 class에서만 관리하므로 유지보수가 편리함.

하지만 알고리즘의 구조가 정해져 있으므로 확장이 제한될 수 있음. 또한 client가 상위 class가 가진 템플릿 메소드의 로직을 잘 이해하고 있어야 하며, 상위 class가 변동되는 경우 전체를 수정해야 함.

4.10.2. Template Method Pattern의 구현

Template Method Pattern은 다음과 같이 구현할 수 있음.

```java
public abstract class CalculateNum {
    protected int x, y;
    public final int templateMethod() {
        int result;
        initNum();
        result = calculate();
        result = hook(result);
        return result;
    }
    public void initNum() {
        this.x = 1;
        this.y = 2;
    }
    public abstract int calculate();
    public int hook(int input) {
        return input;
    }
}
```

```java
public class PlusNum extends CalculateNum {
    @Override
    public int calculate() {
        return x + y;
    }
    @Override
    public int hook(int input) {
        System.out.println("plus operation executed!");
        return input;
    }
}
...

PlusNum plusNum = new PlusNum();
plusNum.templateMethod(); // 3
```

4.11. Visitor Pattern

4.11.1. Visitor Pattern

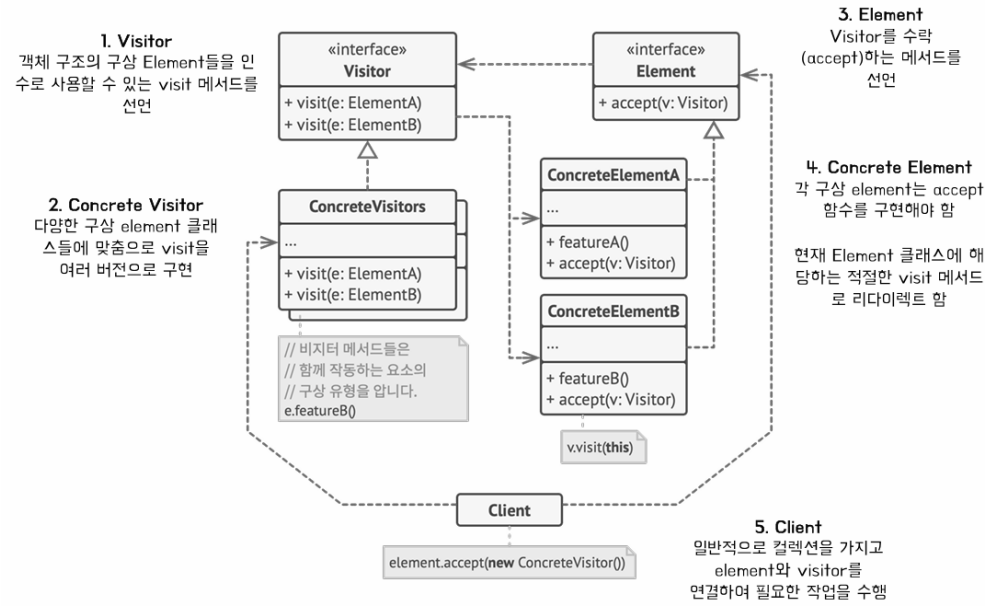
1. Visitor Pattern

Visitor Pattern은 object 구조에서 알고리즘을 visitor라는 object로 분리하고, visitor가 각 object를 순회하며 연산을 수행하도록 하는 DP임. 즉, 알고리즘을 각 object에서 직접 구현해 처리하는 대신, 알고리즘이 구현되어 있는 visitor를 사용해 처리하는 것임. 이는 마치 visitor object가 object 구조의 각 object를 방문하면서 처리하는 것과 같음.

예를 들어, graph 형태로 서로 다른 포맷의 object들이 연결되어 있는 데이터 집합을 생각해보자. 각 object의 정보를 XML 형식으로 내보내려고 할 때, 각 object별로 직접 XML으로 내보내는 메소드를 정의할 수 있지만 이는 SRP, OCP의 관점에서 적절하지 않음. visitor pattern을 사용하면 각 포맷에 대한 메소드를 visitor class에 전부 구현하도록 할 수 있음.

2. Visitor Pattern의 구조

visitor pattern은 다음과 같이 각 element에 대한 알고리즘을 포함하는 visitor와, 연산의 대상인 element로 구성됨. 이때 visitor는 element의 실제 구현체를 인자로 받는 메소드들을 가지며, 이 메소드들은 오버로딩되어 있어 각 element는 동일한 형식으로 visitor를 호출할 수 있음. 각 elements는 구체적인 알고리즘 대신 visitor의 visit()을 호출하는 accept()만 구현하면 됨.



3. Visitor Pattern의 장단점

visitor pattern을 사용하면 새로운 알고리즘을 추가로 도입하거나 기존 알고리즘을 수정할 때 코드 수정을 할 필요가 없으므로 OCP를 준수함. 또한 알고리즘을 별도의 class로 분리하므로 SRP를 준수함. visitor를 다른 element에 재사용하는 것도 가능함.

하지만 element interface가 수정되면 visitor에 대한 수정이 발생할 수 있음.

4.11.2. Visitor Pattern의 구현

Visitor Pattern은 다음과 같이 구현할 수 있음. 앞서 설명한 것처럼 visit()은 오버로딩되어 있고, 이에 따라 element는 visit(this)와 같이 visitor를 사용함.

```java
public interface Shape {
    double accept(ShapeVisitor visitor);
}

public class Circle implements Shape {
    private double radius;
    // getter&setter, 생성자
    @Override
    public double accept(ShapeVisitor visitor) {
        return visitor.visit(this);
    }
}
public class Triangle implements Shape {
    private double sideA;
    private double sideB;
    private double sideC;
    // getter&setter, 생성자
    @Override
    public double accept(ShapeVisitor visitor) {
        return visitor.visit(this);
    }
}
```

```java
public interface ShapeVisitor {
    double visit(Circle circle);
    double visit(Triangle triangle);
}

public class AreaCalculator implements ShapeVisitor {
    @Override
    public double visit(Circle circle) {
        double result = 0.0;
        // circle의 넓이 구하기
        return result;
    }
    @Override
    public double visit(Triangle triangle) {
        double result = 0.0;
        // triangle의 넓이 구하기
        return result;
    }
}
```

```java
List<Shape> shapes = new ArrayList<>();
shapes.add(new Circle(10));
shapes.add(new Triangle(2, 4, 5));
AreaCalculator visitor = new AreaCalculator();
double area = 0;
for(Shape shape : shapes) {
    area = shape.accept(visitor);
    System.out.println("for " + shape + ", area is " + area);
}
```