디자인패턴(Design Patterns)

- twenty-three design patterns by Gang of Four -

수강과목	객체지향프로그래밍
담당교수	정문성 교수님
제출일자	2023.11.20
학 과	컴퓨터공학과
학 번	20211476
이 름	홍준표

목 차

⑥ 상태 패턴(State Pattern)
⑦ 메멘토 패턴(Memento Pattern)
⑧ 책임 연쇄 패턴(Chain-of Responsibility Pattern)
⑨ 중재자 패턴(Mediator Pattern)
⑩ 방문자 패턴(Visitor Pattern)
⑪ 인터프리터 패턴(Interpreter Pattern)

Ⅰ. 서론

1. 연구의 계기와 목적

가. 연구 계기

디자인 패턴은 개발자들 사이에서의 일종의 ‘교과서’ 이다. 개발의 경험이 축적되다 보면 유사한 기능을 구현하기 위해 이전에 사용되었던 코드들을 들여다보는 일들이 종종 발생한다. 이러한 반복되는 형식들을 일종의 패턴으로 기억한다면, 보다 올바른 방식의 설계를 빠르게 적용할 수 있을 것이다. 이렇듯 여러 디자인 패턴들을 익히는 것은 매우 중요한 일이다. 프로그래밍의 높은 효율성 증대를 원하는 개발자들이라면 필수로 알아야 할 개념임이 분명하기에 디자인 패턴에 관한 내용을 공부해보고자 연구를 진행하였다.

나. 연구 목적

디자인 패턴이 익숙하지 않은 개발자라면 빠른 시간에 적재적소에 맞는 패턴들을 찾아 적용시키는 것은 매우 까다로운 일일 것이다. 패턴들마다의 특징과 이해하기 쉬운 예시들을 바탕으로 내용을 정리해 놓는다면, 디자인 패턴을 익히고 적용함에 있어 매우 수월해질 것이다. 본 연구는 디자인 패턴이 익숙지 않은 사람들을 위한 일종의 가이드라인을 제공할 목적으로 작성하였다.

Ⅱ. 본론

1. 생성 패턴(Creational Pattern)

객체의 생성을 사용자에게 모두 처리하도록 놔두는 것은 자칫 클래스의 설계 목적과 맞지 않을 수 있다. 예를 들자면 클래스 설계자가 단 하나만의 객체가 인스턴스화 되기를 기대하고 설계한 클래스가 있다고 가정했을 때 사용자가 여러 개의 객체를 인스턴스화 하는 것은 문제를 발생시킬 여지가 있다. 이러한 객체의 생성과 관련되어 예기치 못한 문제를 미연에 방지하고자 여러 개의 생성 패턴이 만들어졌다.

Gang of Four 의 생성 패턴
1.Abstract Factory
2.Singleton
3.Factory Method
4.Builder
5.Prototype

① 추상 팩토리 패턴(Abstract Factory Pattern)

추상 팩토리 패턴은 관련된 객체들이 서로 유기적으로 동작할 때 관련된 객체들의 집합을 생성하기 위한 디자인 패턴이다. 각 집합마다의 객체를 생성하기 위한 Factory 가 존재하여 사용자가 객체를 생성할 때 보다 직관적으로 이해할 수 있다.

예를 들어보자면, 삼성에서 컴퓨터를 판매할 때는 삼성 제품의 마우스, 키보드, 모니터 등을 함께 판매할 것이다. 마찬가지로 애플에서 컴퓨터를 판매할 때는 자사 제품의 마우스, 키보드, 모니터 등을 함께 판매할 것이다. 삼성 제품을 판매할 때는 삼성과 관련된 집합의 객체를, 애플 제품을 판매할 때는 애플과 관련된 집합의 객체를 특정한 클래스에 구애받지 않고 유연하게 생성할 수 있게 된다. 이는 추상 팩토리 패턴의 장점이다.

<예시 1-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 1-1>

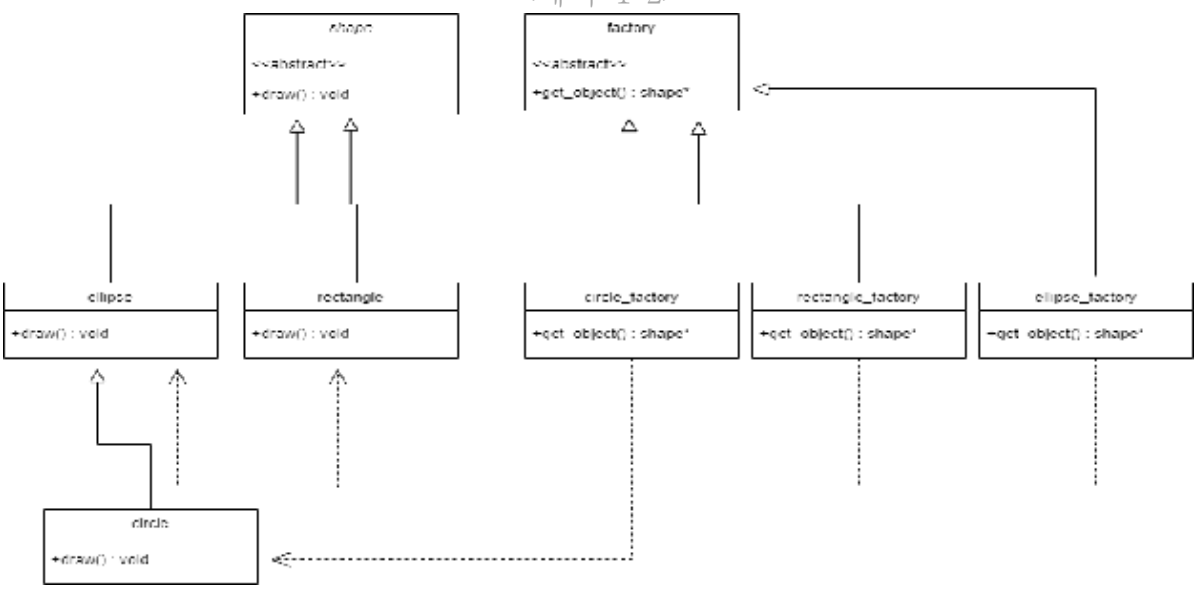
<예시 1-1> 예시 코드는 원과 사각형을 그리는 프로그램이다. 추상 팩토리 패턴에 알맞게 shape 추상 클래스와 객체를 생성하는 factory 추상 클래스가 존재한다. ellipse 클래스와 rectangle 클래스는 각각 shape 클래스를 상속받고 있으며 circle 클래스는 ellipse 클래스를 상속받고 있다. 각각은 shape 클래스에 존재하는 순수가상함수 draw()를 구현하고 있다. 마찬가지로 도형 객체를 생성하기 위해 factory 클래스를 ellipse_factory 클래스와 circle_factory 클래스 rectangle_factory 클래스가 상속받아 객체 생성 함수인 get_object 를 구현하고있다.

하지만 이러한 디자인 패턴의 단점이 존재한다. 만약 개발자가 circle 과 rectangle 이외의 여러가지 도형을 추가하고자 할 때에는 관련된 객체의 클래스를 계속해서 추가하여 구현해주어야 하기 때문에 코드가 매우 복잡해진다

필자는 코드를 직접 작성하여 여러 테스트를 진행해보았다. 객체의 생성이 직관적이어서 어떤 도형의 객체를 생성하는지는 이해하기 쉬웠다 다만 위에서 언급한 코드의 복잡성 문제, 만약 도형을 20 가지 정도를 추가하고자 할 때에는 각각 shape 클래스를 상속받아 구현하고 factory 클래스를 상속받아 구현해야 하기 때문에 20 가지의 도형을 추가하기 위해서는 이러한 반복 작업을 40 번이나 반복해야 한다. 이를 직접 코드로 작성하게 되면 코드 자체가 무한히 길어지고 효율적이지 못하다는 생각이 들었다.
두번째 문제는 코드의 수정 문제이다. 만약 draw() 함수의 기능을 변경하고자 할 때에는 이를 상속받은 모든 도형 클래스의 함수를 수정해주어야 했다. 상속받는 클래스가 적다면 충분히 가능한 일이지만, 클래스가 많아질 경우 단순 반복되는 작업들이 너무나도 많아져 이 또한 비효율 적이라는 생각이 들었다.

<예시 1-2> <예시 1-1> 의 클래스 다이어그램이다.

<예시 1-2>



② 싱글톤 패턴(Singleton Pattern)

싱글톤 패턴은 특정 클래스의 객체가 단 하나만 생성되도록 하는 디자인 패턴이다. 예를 들면 스마트폰에서 비밀번호나 알람 등을 설정한다고 가정했을 때 스마트폰의 설정이 여러 개일 수는 없다. 하나의 객체에서만 수정을 해야하는 작업이기에 이런 경우 싱글톤 패턴으로 구현하는 것이 바람직할 것이다. 물론 이러한 디자인 패턴을 따르지 않는다고 해서 프로그램 개발이 불가능한 것은 아니지만, 다만 그럴 경우 클래스 개발자가 의도하지 않은 동작을 사용자가 행할 여지가 생기기 때문에 객체의 생성을 분리하고 생성자를 private 으로 외부에서의 접근을 제한한다면 이러한 문제를 근본적으로 막는 데에 좋은 방법이 될 것이다.

이러한 싱글톤 패턴의 장점은 하나의 객체를 여럿이 공유해야 하는 경우에 하나의 객체가 메모리에 고정되므로 무분별한 객체 생성을 막을 수 있으며, 불필요한 메모리 낭비를 방지할 수 있다.

하지만 여럿이 하나의 객체를 공유한다는 특징이 오히려 단점이 되기도 하다. 싱글톤 패턴으로 생성된 객체를 다른 클래스에서 참조하게 되면 클래스의 의존성이 높아져 자칫 클래스의 수정이 원치 않은 결과를 발생시킬 수 있다

<예시 2-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 2-1>

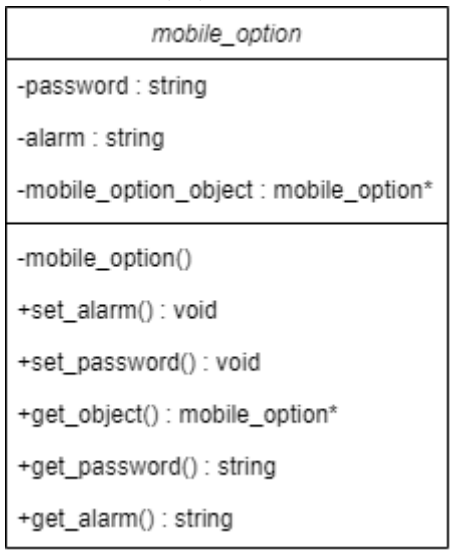
위 코드는 스마트폰에서 비밀번호와 알람을 설정하는 코드이다. 싱글톤 디자인 패턴은 해당 클래스를 사용하는 사용자에게서 객체의 생성을 제한하는 것이 특징이기 때문에 위 코드에서 mobile_option 클래스의 생성자가 private 안에 작성되어 객체의 생성을 제한하고 있는 것을 볼 수 있다. 이렇게 생성자를 private 안에 작성하게 되면 해당 생성자를 호출하여 객체를 만들 수 있는 것은 같은 클래스 안에 존재하는 멤버함수 뿐이다. 따라서 객체 생성을 담당하는 get_object 함수가 존재한다. 또 하나의 특징이 있다면, 객체를 생성하는 get_object 함수는 static 으로 선언된다. 이는 전역함수로써 클래스의 객체를 하나만 생성하도록 보장받는 의미도 있지만 생성자를 접근 제한하였기 때문에, main 함수에서 객체를 생성하여 해당 함수에 접근할 수 없기 때문이다.

싱글톤 디자인 패턴은 생성자의 접근을 제한하여 객체의 생성을 제한할 수 있다는 흥미로운 아이디어로 받아들여졌다. 클래스 사용자의 무분별한 객체 생성을 막을 수 있다면 클래스 개발자의 의도를 사용자에게 정확히 전달함과 동시에 불필요한 메모리 낭비를 줄일 수 있는 일석 이조의 방법인 셈이다.

하지만 장점만이 있어 보이진 않았다. 결국 하나의 클래스에 단 하나의 인스턴스 만을 갖기 때문에 해당 객체는 전역적으로 공유하는 인스턴스이고 이것은 클래스의 활용성 면에서는 큰 단점으로 작용할 것 같았다. 해당 클래스를 사용하여 복잡한 의존관계로 프로그램을 작성하는 것은 알 수 없는 오류를 발생시킬 여지가 있어 좋지 않은 방식으로 생각되었다.

<예시 2-2> <예시 2-1> 의 클래스 다이어그램이다.

<예시 2-2>



③ 팩토리 메소드 패턴(Factory Method Pattern)

팩토리 메소드 패턴은 추상 팩토리 패턴과 매우 유사하다. 사실상 관련된 클래스를 묶어 캡슐화하고 팩토리를 통해 객체를 생성하여 결과값을 얻는 일련의 과정은 동일하다. 그러나 사용 목적에 있어 차이를 보이는데, 팩토리 메소드 패턴은 주로 한 종류의 집합을 생성하기 위해 사용하는 반면 추상 팩토리 패턴은 여러 종류의 집합을 생성하는데 사용된다. 이러한 특징으로 인해 팩토리 메소드 패턴보다 추상 팩토리 패턴의 캡슐화가 더 세밀한 부분까지 되어있는 것을 알 수 있다.

기본적으로 추상 팩토리 패턴과 유사하기 때문에 이것이 갖는 장점들을 일부 동일하게 갖고 있다. 먼저 객체를 생성하는 팩토리가 해당 클래스와 분리되어 있기 때문에 사용자의 사용 편의성 증가와 코드의 유지 보수가 용이해진다. 두번째는 이러한 캡슐화로 인해 기존의 코드를 수정하더라도 다른 곳에 큰 영향을 주지 못한다는 것에 있다.

단점 또한 비슷하다 추상 팩토리 패턴과 마찬가지로 새로운 집합을 추가할 때마다 새로운 클래스를 만들어야 하는데 이것은 무한히 늘어나는 코드로 인해 유지보수와 가독성에 악영향을 미칠 수 있다. 또한 계속해서 늘어나는 클래스는 코드의 복잡성을 증가시킬 수 있다.

<예시 3-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

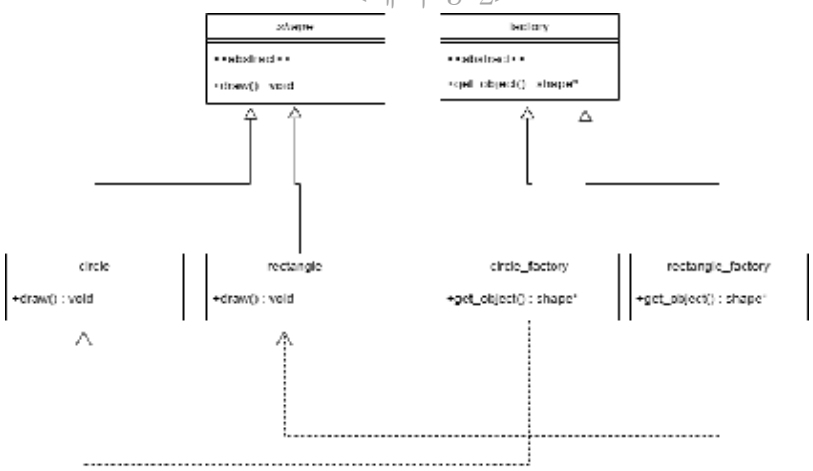
<예시 3-1>

위 코드는 추상 팩토리 패턴과 같은 코드이다. 다만 한가지 다른 점은 ellipse 클래스가 존재하지 않고 circle 클래스만이 존재한다는 점이다. 팩토리 메소드 패턴은 파생 클래스가 다시 한번 여러가지로 파생해야 할 때는 유용하지 않다 ellipse 클래스에서 circle 클래스가 다시 확장되어야 한다면 추상 팩토리 패턴이 적합할 것이다.

추상 팩토리 패턴과 팩토리 메서드 패턴의 차이를 이해하는 것이 정말 어려운 일이었다. 필자가 내린 결론은 공통된 요소가 모인 집단이 여러가지가 존재하고 집단의 요소마다 여러가지 갈래가 나뉘어질 때는 좀 더 세밀한 추상화가 필요하기에 추상 팩토리 패턴을 사용한다. 반면에 하나의 집단에 단일 요소만이 존재하는 경우 팩토리 메소드 패턴을 사용한다는 점으로 이해하였다.

<예시 3-2> <예시 3-1> 의 클래스 다이어그램이다.

<예시 3-2>



④ 빌더 패턴(Builder Pattern)

빌더 패턴은 복잡한 객체의 요소들을 builder 클래스로 설정하고 생성하여 관리한다. 예를 들면 computer 라는 클래스가 있고 컴퓨터에 필요한 부품들이 여러가지 있다고 가정했을 때 이들 전부를 하나의 computer 클래스에서 초기화하고 관리하기에는 클래스의 덩치가 매우 커질뿐더러 생성자 또한 요소가

많아짐에 따라 코드의 길이가 무한적으로 늘어날 것이다. 이는 코드의 가독성을 떨어트리고 직관적이지 못한 방법이다. 만약 이를 builder 클래스로 나누어 역할을 분담시킨다면 복잡한 객체의 요소들을 단계별로 원하는 것만 설정하여 객체를 유연하게 생성시킬 수 있을 것이다.

즉, 빌더 패턴의 장점은 복잡한 객체의 생성에 용이하다는 것이다. 패스트푸드점을 예로 들자면, 손님의 식사에는 햄버거와, 사이드메뉴 그리고 음료수가 준비될 수 있다. 때로는 사이드메뉴 없이 햄버거와 음료수만이 필요할 수도 있을 것이다. 시시각각 변하는 요소들로 구성된 객체를 생성할 때 객체의 생성과 구성이 분리 되어있는 builder 패턴은 유용할 것이다.

다만, 빌더 패턴으로 구현된 코드의 경우 새로운 클래스 추가 시 새로운 또 하나의 빌더 클래스를 생성해야 하므로 코드의 양이 늘어나 오히려 가독성이 떨어질 수도 있다.

<예시 4-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

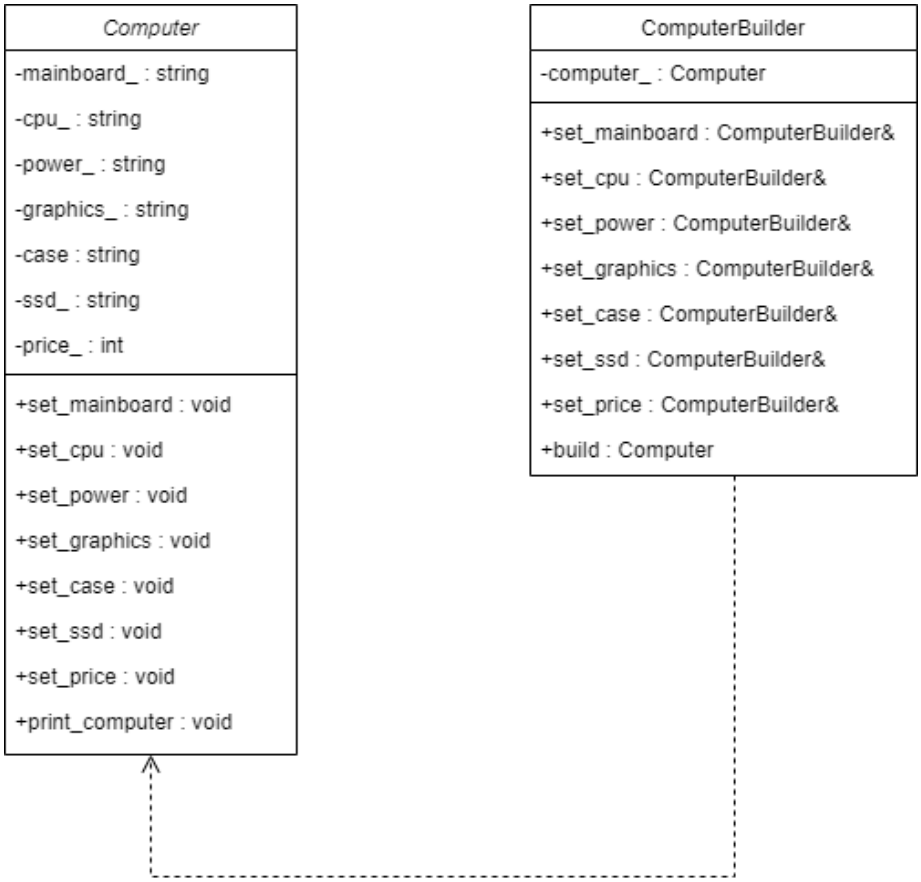
<예시 4-1>

위 코드는 사용자의 입맛에 맞게 컴퓨터의 구성 부품들을 결정하고 나만의 컴퓨터 객체를 만드는 코드이다. 빌더 패턴의 특징인 builder 클래스가 존재하며 computer 클래스의 객체를 만들고 가장 마지막으로 완성된 객체를 build()함수를 통해 생성된 객체를 반환한다.

코드를 작성하며 이와 같은 원리가 적용된 실생활 사례들을 떠올려보았다. 그 중 하나가 바로 앞서 소개한 조립 pc 에 관한 코드인데, 아마도 필자의 생각이지만 '다나와 와 같은 조립 pc 의 견적을 맞추는 사이트들의 경우가 빌더 패턴을 사용하여 만들지 않았을까 하는 생각을 해보았다.

<예시 4-2> <예시 4-1> 의 클래스 다이어그램이다.

<예시 4-2>



⑤ 프로토타입 패턴(Prototype Pattern)

프로토타입 패턴은 new 연산자를 사용하여 새로운 객체를 생성하는 것이 비효율적일 때나 사용자가 객체의 복사본을 많이 생성해야 할 때 사용된다. 한 객체에서 다른 객체로의 복사본을 만드는 것은 복사 생성자의 활용과 유사하지만, 복사 생성자의 경우 다형성이 적용된 환경에서는 활용할 수가 없다.

프로토타입 패턴의 장점은 다른 생성패턴들과 마찬가지로 캡슐화를 통해 구체적인 객체를 만드는 과정을 숨길 수 있다. 또한 기존 객체를 복제하는 과정이 새 객체를 생성하는 것 보다 비용적인 측면에서 이득일 수 있다.

하지만 적절한 상황(객체 생성 비용이 큰 경우 등)이 아닐 때 사용하게 된다면 프로토타입 패턴 또한 새로운 객체를 생성하는 것이므로 오히려 메모리 낭비로 이어질 수 있다. 상황에 따라 적절한 패턴을 활용하는 것이 중요하다

필자는, 프로토타입 패턴이 clone()이라는 직관적인 함수로 복사를 생성하니 코드의 목적을 파악하는데 (가독성) 매우 유용했다. 또한 복사 생성자 부분을 미리 구현해 놓고 함수로 호출하는 방식은 shallow copy 에서 오는 문제를 미연에 방지하는 효과도 있어 단점보다 장점이 큰 디자인 패턴이라 생각되었다.

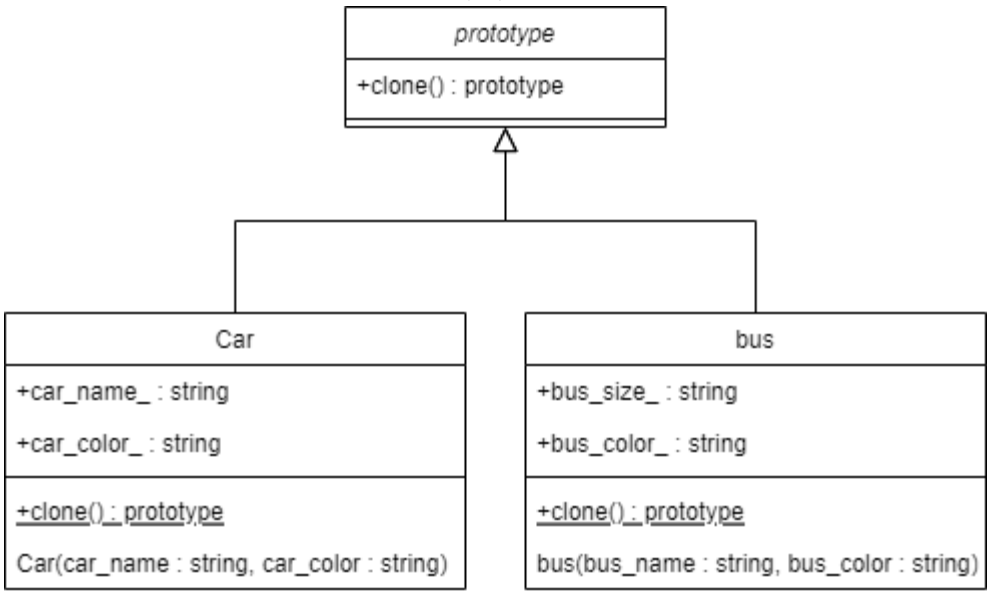
<예시 5-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 5-1>

위 코드는 『포르잔 C++ 바이블』 책에 수록된 코드를 가져왔다. 약간의 다른 점이라면 스마트 포인터를 사용하여 scope 를 벗어나면 클래스마다의 소멸자가 자동으로 호출된다는 점이다. 위 코드는 자동차 객체와 버스 객체를 생성한다. 각각의 객체 생성자에 모델명 또는 크기와 색상을 인자 값으로 넣고 그 객체를 clone()함수를 통해 복사하고 출력해준다. 이런 식의 객체의 복사 과정이 많이 일어나야 하는 상황이라면 프로토타입 패턴으로 코드를 구현하게 되었을 때 객체의 복사가 매우 수월 해진다.

<예시 5-2> <예시 5-1> 의 클래스 다이어그램이다.

<예시 5-2>



2. 구조 패턴(Structural Pattern)

구조 패턴은 클래스의 상속과 합성을 통해 더 큰 구조로 만드는 방식을 의미한다. 예를 들면 다른 인터페이스를 갖는 2개의 객체가 묶여 하나의 인터페이스를 제공하거나 여러 객체들을 하나로 묶어 새로운 기능을 제공한다, 구조패턴을 사용하면 독립적인 클래스를 하나의 클래스인 것처럼 사용할 수 있다.

Gang of Four의 구조 패턴

1.Bridge Pattern
2.Adapter Pattern
3.Facade Pattern
4.Flyweight Pattern
5.Proxy Pattern
6.Decorator Pattern
7.Composite Pattern

① 브리지 패턴(Abstract Factory Pattern)

'브리지'는 단어 그대로 서로를 잇는 다리를 의미한다. 브리지 패턴에서의 다리는 어떠한 기능 클래스 계층과 구현 클래스 계층의 연결을 의미하며 예를 들자면, 도형에는 '구' 와 '정육면체' 가 있고 색상에는 '빨강' 과 '파랑' 이 있을 때 이를 상속으로 구현하기 위해서는 'shape' 클래스를 각각의 'RedCircle' ' RedSquare' ' BlueCircle' ' BlueSquare' 등의 색상별로 모양별로 자식클래스를 만들어야 하는 매우 비효율 적인 상황이 생긴다. 만약 이 상황에서 다른 색상이 하나 추가된다 거나 새로운 도형이 추가된다면 엄청난 양의 클래스를 다시 만들어야 하는 상황에 놓일 것이다. 이를 해결하기 위한 것이 브리지 패턴으로, shape를 종류별로 클래스들이 상속하는 것이 아닌 shape와 color 인터페이스로부터 각각 circle, square / Red, Blue 등의 클래스를 만들어 상속한다면 새로운 도형이나 색상이 추가되었을 때 각각의 클래스에 하나씩만 추가해 주면 되기 때문에 이전의 방식보다 훨씬 간편하면서도 직관적인 코드를 작성할 수 있게 된다.

브리지 패턴의 장점으로는 위의 설명에서 언급했듯이 하나의 shape 인터페이스가 아닌 shape와 color를 분리함으로써 각각을 독립적으로 확장할 수 있게 되었다는 것이다. 이는 코드의 재사용성을 높이며 코드의 유지보수를 용이하게 만들어 준다.

브리지 패턴의 단점으로는 추상화로 코드를 분리할 경우 코드의 설계가 다소 복잡해질 여지가 있다.

2020년도 쯔음, 교내 행사 출품작으로 간단한 슈팅게임을 만든 적이 있었다. 이 게임은 장애물이 날아오고 그것을 플레이어가 피하여 스테이지를 넘어가는 방식으로 진행되는 게임이다. 게임을 개발하면서 장애물과 장애물에 부딪히면 발생

하는 디버프 들이 워낙 다양하다 보니 이를 한곳에 묶어 '장애물' 구조체와 '디버프' 구조체로 나누어 관리하면 어떨까? 라는 생각을 잠깐 해본적이 있었다. 실제로는 구현하지 않았기 때문에 많은 수고로움이 있었지만 당시에 브리지 패턴을 알고 있었다면 좀 더 수월하게 게임을 만들 수 있지 않았을까 하는 생각이 들었다.

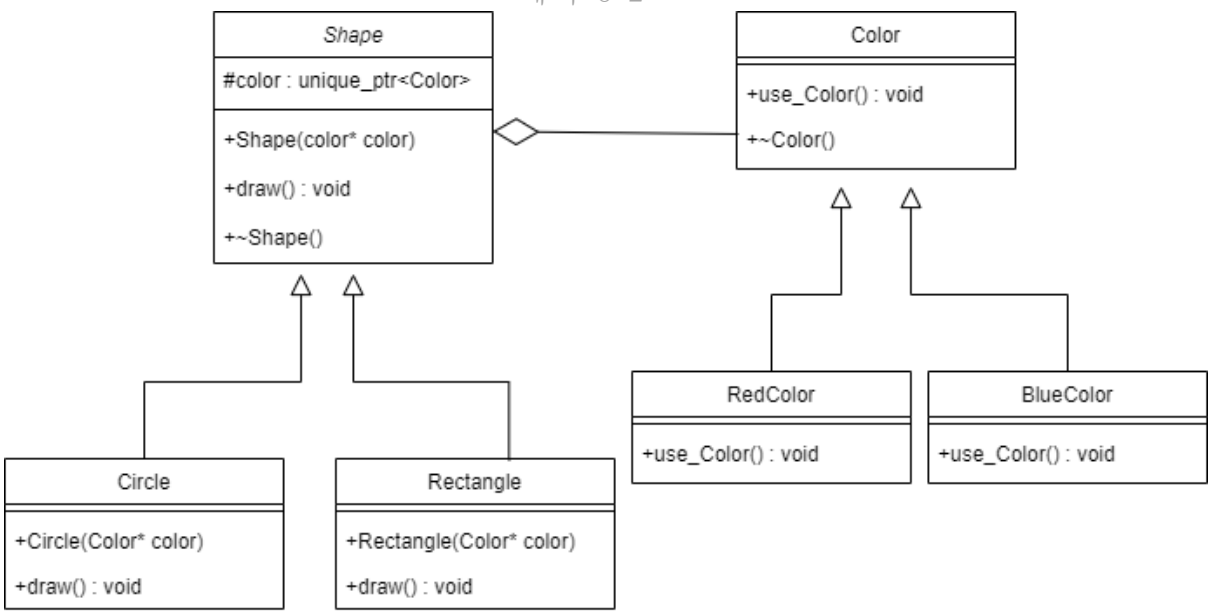
<예시 6-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 6-1>

위 코드는 도형과 그에 맞는 색상을 선택하여 출력하는 코드이다. 브리지 패턴을 적용하여 도형 부분의 shape 클래스와 색상을 결정하는 color 클래스로 분리하여 각각 shape, rectangle 그리고 RedColor, BlueColor 가 존재한다. shape 에서는 color 의 인스턴스 하나를 갖고 인스턴스로부터 색상 결정 함수인 use_Color() 를 호출하다. 이러한 방식의 장점은 도형을 추가로 늘리거나 색상을 추가할 때 클래스 하나만을 더 만들어 주면 되기 때문에 매우 간편하다.

<예시 6-2> <예시 6-1> 의 클래스 다이어그램이다.

<예시 6-2>



② 어댑터 패턴(Adapter Pattern)

어댑터 패턴은 마치 플러그와 소켓 간의 연결시켜주는 연결자와 같은 역할을 하는 패턴을 의미한다. 외국에서 콘센트에 전원을 꽂을 때 우리나라의 표준 규격인 220v 와 맞지 않아 그냥 사용하려고 하면 제품이 망가질 수도 있다. 이때 서로 다른 규격을 호환시켜 주는 변압기가 존재하는데 이를 어댑터라고 볼 수 있다. 같은 맥락으로 이러한 중간자 역할의 어댑터는 클래스로 존재할 수 있으며 기존의 사용하던 인터페이스를 사용하고자 하는 원하는 형태의 인터페이스로 어댑터를 사용하여 바꿔줄 수 있다

이러한 패턴의 장점이자 목적은 호환성이 없는 기존 클래스의 인터페이스를 사용자가 원하는 형태로 바꾸어 사용할 수 있게 만들어 코드의 재활용성을 높인다. 또한 코드 전체를 전부 다시 구현할 필요가 없기 때문에 이러한 면에서는 매우 효율적이라고도 볼 수 있다.

하지만 단점으로는 기존 코드에서 새롭게 어댑터 클래스를 만들어 주어야 하기 때문에 전반적인 코드의 양이 늘어나게 되고 이는 가독성이 떨어지고 유지 보수가 힘들어질 수 있다

필자는 어댑터 패턴이 유용하지 못하다고 느낀다. 만약 기존의 기능에서 어댑터를 통해 몇 가지 다른 기능을 추가하여 사용하고자 한다면 어댑터 패턴을 사용해야 하겠지만, 이것의 사용을 위해서 새로운 어댑터 클래스를 만들고 구현해야 하는 것은 비효율적이라 느꼈다. 가능하면 기존 클래스를 수정하여 추가된 기능을 사용하는 것이 바람직해 보였다. 서로 다른 독립적인 클래스의 호환성을 위한 어댑터 패턴의 사용은 괜찮아 보였지만 이 패턴의 사용이 근본적인 원인을 해결하는 데에 초점을 맞춘 것이라 생각되지는 않았다.

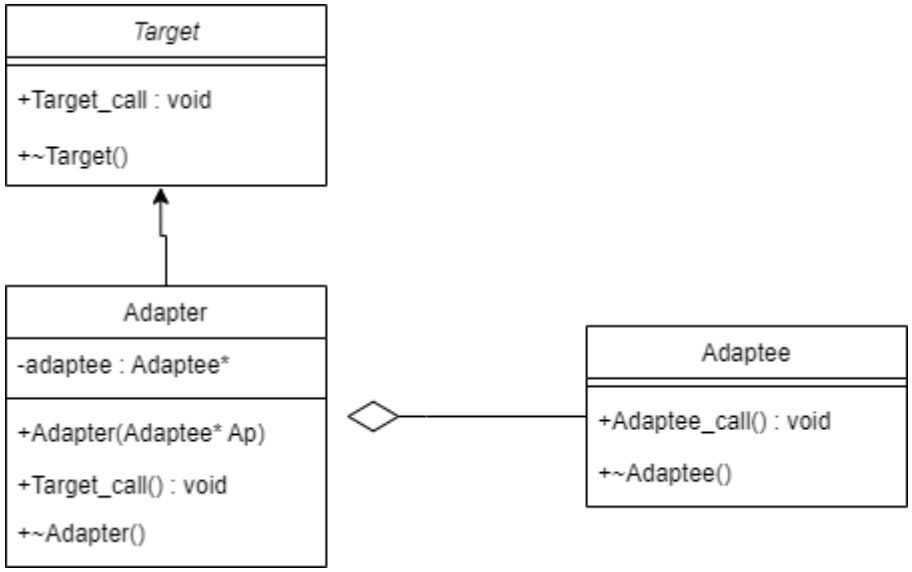
<예시 7-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 7-1>

위 코드는 기존의 클래스인 Target 클래스를 Adapter 클래스를 통해 Adaptee 클래스의 형태를 사용하려 하는 코드이다. 어댑터 클래스는 기존 클래스인 Target 을 상속받아 Target_call()함수를 재정의하여 Adaptee_call()함수를 호출하도록 한다. 이렇게 하면 기존의 Target_call()함수를 호출하더라도 어댑터를 통해 오버라이딩 된 새로운 Target_call()함수가 호출되어 궁극적으로 Adaptee_call()함수가 호출되어진다.

<예시 7-2> <예시 7-1> 의 클래스 다이어그램이다.

<예시 7-2>



③ 퍼사드 패턴(Façade Pattern)

퍼사드 패턴은 다른 클래스들의 집합에 대한 하나의 통합된 인터페이스를 제공하는 패턴이다. 하나의 목적을 위해 많은 클래스의 객체들을 직접 하나하나 수정하고 작동시키기에는 매우 어렵고 난해한 작업일 것이다. 예를 들어 집의 구조를 그리는 프로그램이라고 가정한다면 집에는 안방, 부엌, 거실, 화장실 등 많은 방들이 존재할 것이다. 집을 그릴때마다 각각의 방의 객체를 생성하여 초기화 시켜주고 방의 크기를 계산하여 그리는 것은 매우 비효율적인 방법이므로 퍼사드 패턴을 사용하여 하나의 클래스를 만들고 해당 클래스 안에서 객체의 생성과 값의 초기화 계산의 결과 까지를 한 번에 담당한다면 사용자는 하나의 객체 생성만으로 집의 구조도를 그릴 수 있을 것이다.

퍼사드 패턴의 장점은, 기존의 코드는 사용자가 많은 클래스들을 직접 호출하여 사용하였기 때문에 코드가 난해해지고 사용함에 있어 불편함을 초래했다. 퍼사드 패턴의 사용으로 시스템에 대한 의존을 한곳으로 모은다면 여러 클래스들을 한 번에 다루기 편해지고 코드의 가독성이 좋아진다. 또한 퍼사드 클래스를 사용하여 요청하고 처리하기 때문에 사용자와 해당 기능을 담당하는 클래스와의 결합도를 낮출 수 있다.

퍼사드 패턴의 단점은 시스템에 대한 의존을 한곳으로 모은다는 점으로부터 발생한다. 어찌되었든 시스템 자체가 퍼사드 클래스에 대한 의존성이 강해지므로 해당 클래스를 수정하거나 재사용하기에 불리할 수 있다. 따라서 퍼사드 패턴을 사용하더라도 클래스에 대한 의존성을 적절히 분배하는 것 또한 중요할 것이다.

퍼사드 패턴을 접하기 전까지는 일괄적인 작업들을 함수 내에서만 해결하려 고민하였다. 이러한 방식은 많은 부분에서 한계가 있었고, 서로다른 원하는 기능의 묶음이 여럿 필요로 할 때는 함수 내부에서만 조작하기에는 가독성도 떨어지고 직관적이지 못했다. 이러한 패턴처럼 기능들을 담당하는 함수들을 한곳에 묶어

클래스 안에서 적절히 활용된다면 원하는 기능의 묶음 객체를 여러 개 생성하여 서로 다른 값을 도출해내는 것도 가능하기에 자주 활용할 것 같다.

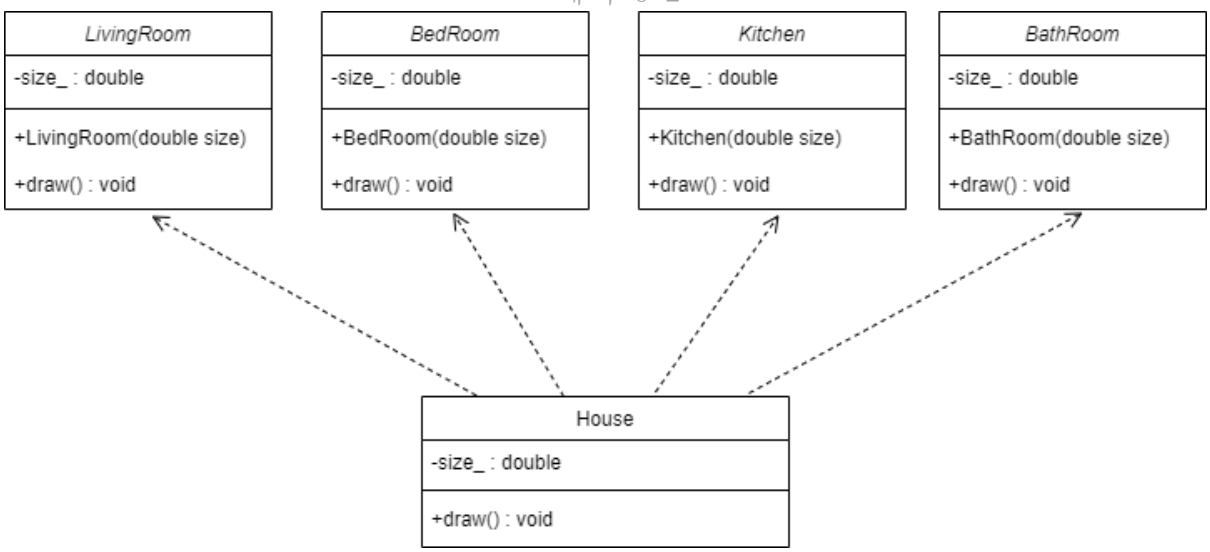
<예시 8-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 8-1>

위 코드는 사용자가 사이즈를 지정하면 욕실과 거실, 침실과 부엌의 크기를 출력하면서 평방 피트로 그림을 그리는 프로그램이다. 코드에는 방마다의 클래스가 존재하는데, 이것을 하나하나 호출하기에는 코드가 난잡해지고 사용성이 떨어지므로 House 라는 퍼사드 클래스는 사용자가 입력한 size 의 값이 일괄적으로 대입되고 계산되어 적절한 함수를 호출해준다. 사용자는 여러 함수를 호출할 필요 없이 퍼사드 size 값을 대입하여 퍼사드 객체를 생성하고 객체로부터 draw()함수만을 호출하면 일괄적으로 모든 방의 사이즈를 출력하고, 그림을 그려준다.

<예시 8-2> <예시 8-1> 의 클래스 다이어그램이다.

<예시 6-2>



④ 플라이웨이트 패턴(Flyweight Pattern)

플라이웨이트 패턴은 어떤 클래스에 대한 객체를 여러 개 만들어야 할 때 사용된다. 만약 클래스 내부에 존재하는 멤버변수의 값을 객체들이 전부 같은 값을 갖고 있다고 가정해보면 해당 클래스의 객체를 생성할 때 마다 새로운 메모리에 같은 값이 계속해서 할당되는 상황이 벌어질 것이다. 작은 데이터의 크기라면 프

로그램 실행에 있어 큰 문제가 되지 않겠지만, 중복되는 데이터의 크기가 객체 하나당 100mb 만 차지한다 하더라도 1000 개의 객체를 만들게 되면 100gb 이상의 메모리를 차지하기 때문에 실질적으로 실행되기 어려운 프로그램이 되어버린다.

플라이웨이트 패턴의 가장 큰 장점은 많은 객체 생성이 필요할 때 중복되는 객체들은 하나만을 생성하여 그것을 공유함으로써 메모리 낭비를 효과적으로 줄일 수 있다.

하지만 중복되는 객체를 하나의 메모리로 공유하다 보니 객체마다의 속성을 다르게 적용하여 사용하는 것은 불가능하다. 이것이 플라이웨이트 패턴의 단점이다.

플라이웨이트 패턴은 메모리를 공유한다는 점에서 앞서 서술한 싱글톤 패턴과 굉장히 유사한 느낌을 받았다. 싱글톤 패턴의 경우 생성자를 private 으로 접근을 제한함으로서 하나의 객체만을 생성시켜 메모리를 공유하게 만드는 패턴었다면 플라이웨이트 패턴은 객체가 중복될 경우 해당 객체를 한 번만 생성하게 만드는 것이니 싱글톤 패턴보다 활용도가 훨씬 좋을 것이란 생각이 들었다.

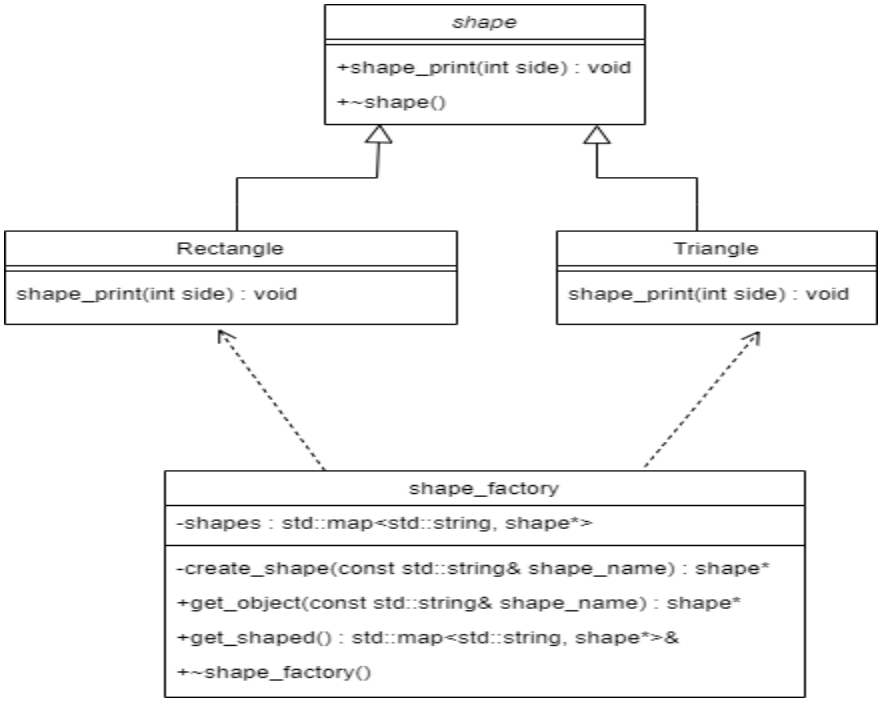
<예시 9-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 9-1>

위 코드는 사용자가 도형을 정한 후 한 변의 길이를 입력하여 함수를 호출하면 도형의 둘레를 출력하는 프로그램이다. 코드에서 get_object() 함수의 인자값에 원하는 도형의 이름을 입력하면, factory 클래스 내부에서 해당 이름으로 생성된 객체가 있는지 판단 후 있으면 기존 객체를 반환, 없으면 새로운 객체를 만들어 반환시킨다. 이렇게 중복된 이름을 갖는 rectangle 의 경우 한 변의 길이가 달라졌다고 해서 새로운 객체를 만드는 것 보다는 기존의 객체를 반환하여 같은 연산을 수행한다면 메모리를 절약할 수 있을 것이다.

<예시 9-2> <예시 9-1> 의 클래스 다이어그램이다.

<예시 9-2>



⑤ 프록시 패턴(Proxy Pattern)

프록시는 대리자 또는 대리인의 의미를 갖고 있다. 어떠한 객체가 해야하는일을 다른 곳으로 위임하여 다른 객체가 이를 대신 처리하는 방식을 프록시 패턴이라 한다. 이러한 프록시의 특성을 이용한 것이 바로 '스마트 포인터' 인데 이 스마트 포인터는 하나의 객체로서, 포인터의 역할을 함과 동시에 메모리 누수를 막는 delete 의 기능까지 가지고 있는 일종의 프록시라 볼 수 있다.

이러한 프록시 패턴의 장점은 사용자가 원본 객체에 대한 접근을 직접적으로 접근하지 않고 프록시 객체를 통해 접근하기 때문에 프록시 객체를 사용하여 원본 객체에 대한 접근을 제어할 수 있어 보안성이 그만큼 향상된다.

프록시 패턴의 단점은 사용자가 원본 객체의 기능을 사용하고자 할 때 프록시 객체를 통한 접근을 해야 하기 때문에 프로그램의 성능이 저하될 수 있다. 뿐만 아니라 객체를 여러 개 생성할 경우 원본객체 또한 다수가 계속해서 생성되기 때문에 메모리 효율이 저하될 우려가 있다.

프록시 패턴은 유용하긴 하지만 이미 있는 기능을 대신 처리해준다는 특징이 그닥 매력 있게 다가오진 않았다. 어댑터 패턴처럼 이미 만들어진 코드에 코드를 붙여 사용해야 하는 방식 중 하나라는 생각에 설계에 문제를 보완하려는 목적으로 만든 패턴이 아닐까 라는 추측을 해보았다.

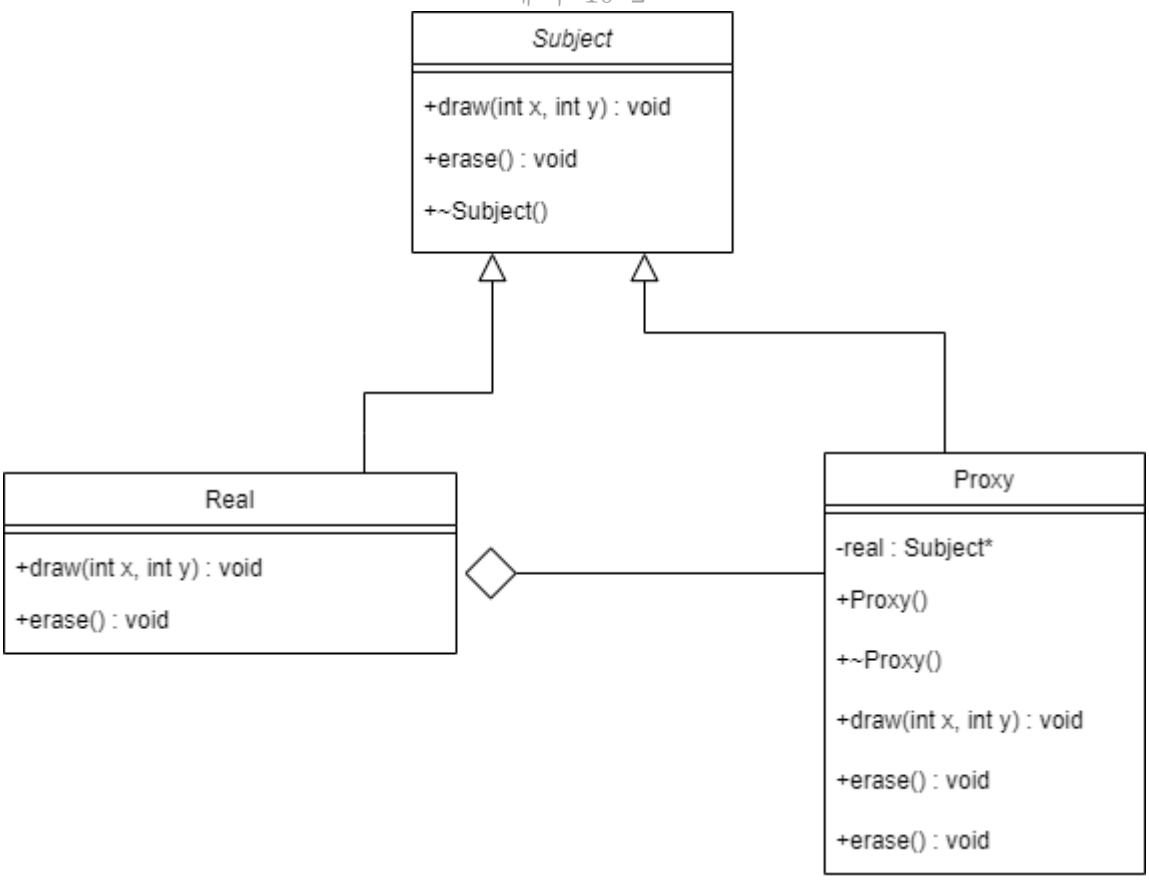
<예시 10-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 10-1>

위 코드는 real 클래스에 존재하는 draw()함수를 통해 사용자가 지정한 좌표값에 그림을 그리고 지우는 역할을 하는 프로그램이다. 코드 내에는 프록시 클래스가 존재하며 사용자는 real 클래스의 객체를 직접 사용하지 않고 프록시 클래스의 객체를 통해 간접적으로 객체를 생성하고 draw()함수를 호출하게 된다.

<예시 10-2> <예시 10-1> 의 클래스 다이어그램이다.

<예시 10-2>



⑥ 데코레이터 패턴(Decorator Pattern)

데코레이터 패턴은 단어 그대로 객체를 꾸미는 객체가 존재하는 패턴이다. 꾸민다는 의미는 어떠한 기능적인 것들을 추가하여 확장이 가능하다는 의미이다.

데코레이터 패턴은 꾸미고자 하는 클래스를 상속 없이 한번 감싼 후에 기능을 확장하는 도구로써 작동하여 서브클래스보다 더욱 유동적인 기능 확장이 가능하다. 이러한 패턴은 객체의 기능을 동적으로 확장하거나 변경이 필요한 상황에서 사용할 수 있다.

데코레이터 패턴의 장점은 런타임 때 객체의 기능을 추가하여 동적으로 생성할 수 있다. 앞서 서술한 대로 데코레이터 객체가 원본을 감싸는 형태로 특수한 특징을 추가하여 구현하기 때문에 확장이 용이하다.

이러한 패턴은 특정한 기능을 하는 클래스를 겹겹이 둘러싸고 있기 때문에 이것이 어떠한 기능을 하는지 정확히 알기가 힘들어진다. 코드가 복잡해지게 되면 데코레이터 패턴을 적용한 클래스가 어떠한 일들을 하는지 원본 클래스가 무엇인지 명확하지 않아 코드의 가독성이 떨어질 수 있다.

이러한 패턴으로 프로그램을 짜는 것은 득보다 실이 더 많아보였다. 데코레이터 패턴 특성상 클래스의 확장을 상속을 이용하지 않고 새로운 클래스를 만들어 기능을 위임하는 형태로 구성되는데 이러한 점 때문에 코드를 이해하기가 매우 어려워졌다. 복잡한 코드 속에서 만약 데코레이터 클래스의 이름과 원본 클래스의 이름이 전혀 관련성이 없다면 혹은 연관성을 찾을 수 없다면 원본 클래스를 찾을 수 없기 때문에 이 클래스가 어떠한 기능을 하는지 전혀 알 길이 없다. 오히려 장점을 포기하고 상속으로 클래스를 확장하는 편이 가독성 측면에서는 더 좋아보였다.

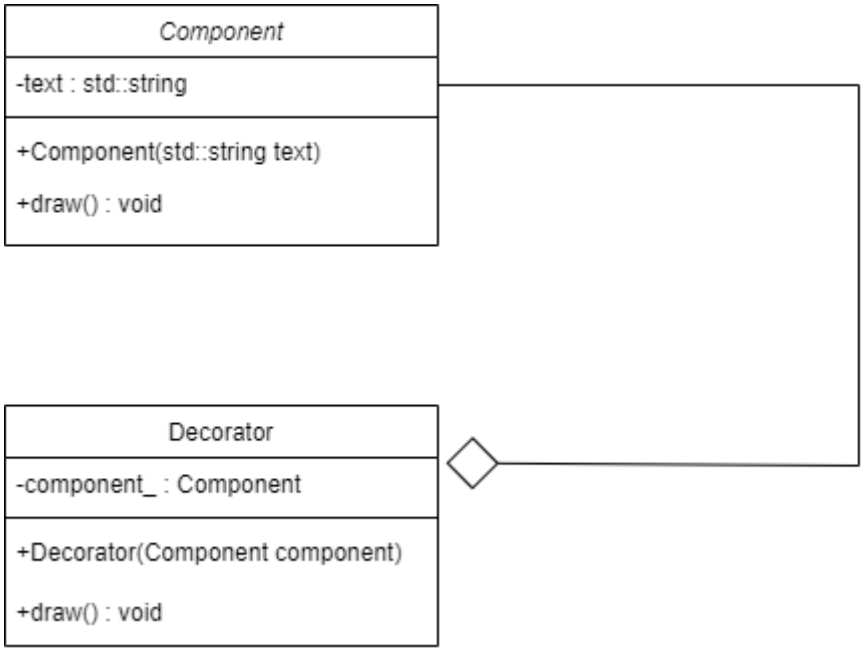
<예시 11-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 11-1>

위 코드는 Component 의 기능을 Decorator 클래스가 위임하여 대신 수행하는 코드로 볼 수 있다. 원본 클래스인 Component 클래스는 생성자에서 사용자가 입력한 문자열을 초기화하고 draw()함수를 통해 출력한다. 여기서 Decorator 는 “*****..” 들을 먼저 출력하고 Component 의 객체로 draw()함수를 호출함으로써 밑밑했던 문자열에 꾸밈을 더하는 작업을 하다. 사용자는 이러한 기능을 사용하기 위해서는 Component 의 객체가 아닌 Decorator 의 객체를 만들어 draw()함수를 호출하면 일련의 과정들이 자동화되어 꾸며진 문자열이 출력되게 된다

<예시 11-2> <예시 11-1> 의 클래스 다이어그램이다.

<예시 10-2>



⑦ 컴포짓 패턴(Composite Pattern)

컴포짓 패턴은 그룹 전체와 개별의 객체를 동일하게 처리할 수 있는 패턴을 뜻한다. 여러 객체로 구성된 객체를 하나의 객체처럼 취급해야할 때 컴포짓 패턴을 사용한다. 예를 들면 그래픽 시슽템이나 문서 편집기, 혹은 gui 에서 사용되는데, 그래픽 시스템의 경우 요소들이 모여 하나의 큰 물체를 만들기 때문에 각각의 요소들을 하나의 객체로 취급한다면 물체는 복합 객체가 되는 것이다

이러한 패턴의 장점은 사용자가 복합 객체를 사용하나 단일로 객체를 사용하나 동일하게 취급하기 때문에 사용성 면에서 매우 유용하다. 또한 이러한 동일하게 취급하는 특징 덕에 설계의 범용성이 조아지고 새로운 클래스를 추가하기에 용이하다.

하지만 묶음 클래스 속에서 각각의 객체를 처리해주어야 하기 때문에 프로그램의 효율이 저하될 수 있으며, 새로운 하위 leaf 클래스가 많아질수록 디버깅의 어려움이 생긴다

앞서 과거 학창시절 출품작으로 만들었던 슈팅게임에 관련한 얘기를 꺼낸 적이 있었다. 이 슈팅게임의 개발 과정은 절차지향에 바탕을 둔 맹목적인 코드 덩어리였다. 여러 구조체들이 필요할 때마다 구성해서 사용하다 보니 비슷한 유형의 구조체들이 여기저기 난잡하게 작성되어 있어 오류를 찾기도, 객체를 다루기도 여간 힘든 일이 아니었다. 이러한 디자인 패턴을 미리 알았더라면, 동일하게 작동하는 구조체를 한곳에 묶어 처리했더라면, 오류를 찾느라 몇 날 며칠을 밤새며 코드 하나하나 해석해보는 일은 없었을 것이다.

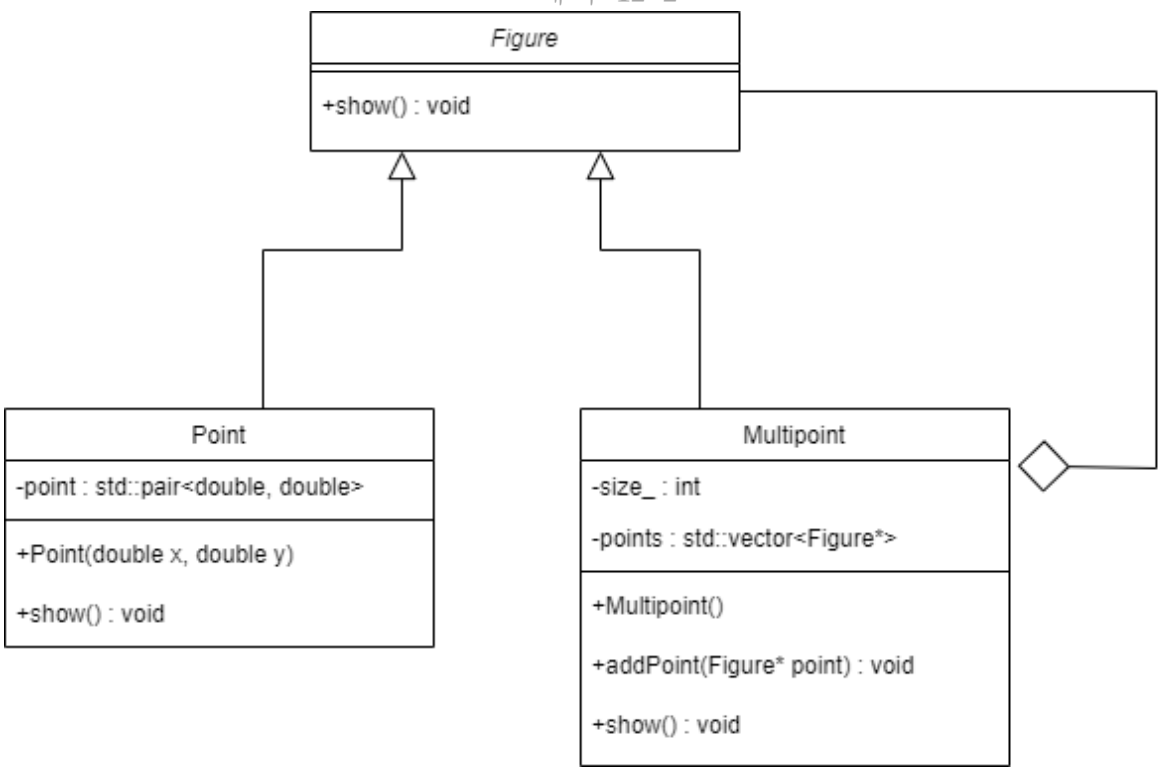
<예시 12-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 12-1>

위 코드는 점과 그 점들의 집합을 다루는 코드이다. 코드가 매우 단순하고 그래픽적인 요소가 없어 점 하나만을 좌표로 표현하고, 복합객체에서 점 여러 개의 좌표를 표현하는 한계가 있다. 다만 위 코드의 핵심은 하나의 점과 점 여러 개를 뭉친 것은 결국 이동과 표현을 똑같이 할 수 있다는 점에서 서로 비슷한 객체로서 취급하고 사용이 가능하다는 것이다

<예시 12-2> <예시 12-1> 의 클래스 다이어그램이다.

<예시 12-2>



3. 행동 패턴(Behavior Pattern)

만약 복잡한 구성을 갖는 코드가 있다고 가정해보면 그곳에는 상속 관계의 객체, 의존 관계의 객체 등 여러 방식으로 얽히고 설켜 있을 것이다, 복잡한 구성을 갖는 프로그램을 어떠한 구조적인 계획 없이 무차별적으로 필요에 의해 사용하다 보면 어느 한 클래스의 의존도가 높아져 예상치 못한 오류를 만날 수 있을 것이다. 이러한 문제를 해결하고자 행동 패턴은 객체나 클래스 사이의 책임 분배와 관련된 패턴들을 제시한다. Gof 의 행동 패턴은 총 11 개로 구성되어 각각의 패턴들은 상황에 따라 책임을 적당히 분배하는 방법을 제시해준다.

Gang of Four 의 행동 패턴

1.Iterator Pattern
2.Template Method Pattern
3.Strategy Pattern
4.Command Pattern
5.Observer Pattern
6.State Pattern
7.Memento Pattern
8.Chain-of Responsibility Pattern
9.Mediator Pattern
10.Visitor Pattern
11.Interpreter Pattern

① 반복자 패턴(Iterator Pattern)

반복자 패턴은 컨테이너(배열이나 리스트 등)의 구현을 숨기면서도 컨테이너의 요소를 접근할 수 있게 하는 반복자 클래스를 제공한다. 이름이 반복자인 이유는 컨테이너의 데이터 형이 연속되어진 데이터의 묶음이기 때문에 이를 반복하여 꺼내온다는 의미로 반복자 패턴이라 불린다.

해당 패턴의 장점은 연속된 자료의 내부 구조를 사용자에게 감춤으로써 구조가 바뀌더라도 사용자의 코드에는 큰 영향을 미치지 못한다. 이것은 컨테이너와 데이터를 읽어오는 부분을 분리함으로써 단일 책임 원칙을 준수하여 코드의 유지보수를 향상시킨다.

반복자 패턴의 단점은 만약 데이터 종류가 바뀐다면 반복자 코드를 다시 수정해주어야 해서 불편함이 있다. 또한 간단한 배열만으로 구성된 프로그램에서 반복자 패턴을 적용하게 된다면 오히려 복잡도만 증가할 뿐 전혀 효율적이지 못하게 된다. 따라서 적절한 상황에서의 적용이 필요하다.

반복자 패턴은 연속적인 데이터형(배열, 리스트, 연속된 객체 등)을 하나의 반복자 클래스로 가져오는 데에 매우 편리함이 있지만 실질적으로 배열 형태의 다른 데이터형을 활용해본 기억이 거의 없다. 큰 규모의 프로젝트를 개발할 때에는

필요한 패턴으로 보이지만, 적절하지 않은 곳에 사용하게 되면 가독성만 떨어질 뿐 효율적이지 못할 것이 분명하기에 당분간은 위 패턴을 사용할 일은 없을 것 같다.

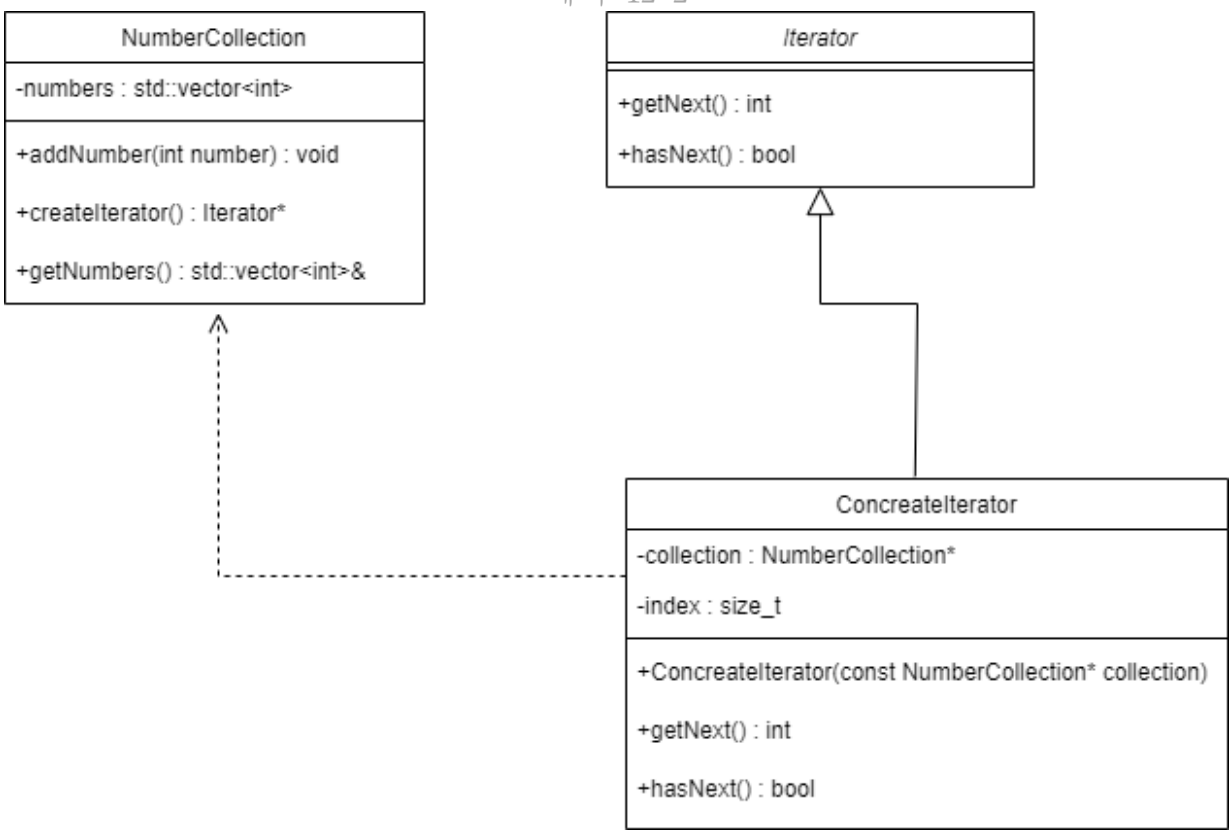
<예시 13-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 13-1>

위 코드는 사용자가 addNumber()함수를 통해 저장한 숫자를 push_back()함수를로 차례로 저장한 뒤 iterator를 거치며 순서대로 출력하는 코드이다. 반복자 패턴의 핵심인 배열의 값 저장 부분과 배열의 출력 부분이 나뉘어져 존재한다.

<예시 13-2> <예시 13-1> 의 클래스 다이어그램이다.

<예시 12-2>



② 템플릿 메소드 패턴(Template Method Pattern)

템플릿 메소드 패턴은 코드를 단계별로 나누어 이것을 함수나 또는 그러한 메서드들로 만든 뒤 템플릿 메서드에 함수의 호출을 순서에 맞게 정의하는 방식으로 구성된다. 즉 어떠한 처리 과정을 세분화하여 메서드로 만든 뒤 기반 클래스에서 순서를 정의한다는 의미이다. 또한 공통으로 사용되는 함수를 상위 클래스에 순수 가상 함수로 정의하고 이를 서브 클래스에 구현함으로써 코드의 확장성이 좋아진다.

템플릿 메소드 패턴의 장점은 추상 클래스를 통해 하위 클래스에서 자세한 구현을 하기 때문에 기반 클래스만을 참고하더라도 전반적인 코드의 흐름을 알기 쉽고 가독성이 뛰어나다. 또한 이러한 자세한 구현을 위임하는 형태로 인해 코드의 중복을 최소화할 수 있으며 확장성이 뛰어나다.

이와 반대로 템플릿 메소드 패턴의 단점은 클래스를 상속하여 기능을 위임받는 형태로 인해 코드를 유연하게 작성하기에는 불편함이 생길 수 있으며, 추상 메소드들이 많아질 경우 프로그램의 유지관리가 어려워질 수 있다.

프로그램을 개발할 때 특정한 기능을 하는 함수를 여러 개 만들곤 한다. 필요에 따라서 즉각즉각 만들기 때문에 구조화나 체계화가 제대로 되지 않는 경우가 많았다. 템플릿 메소드 패턴의 경우 사용하고자 하는 함수를 기반 클래스에 정의하고 서브 클래스로 위임하기 때문에 개발자의 목적과 함수의 기능을 유추하는 데에 편리함을 느꼈다.

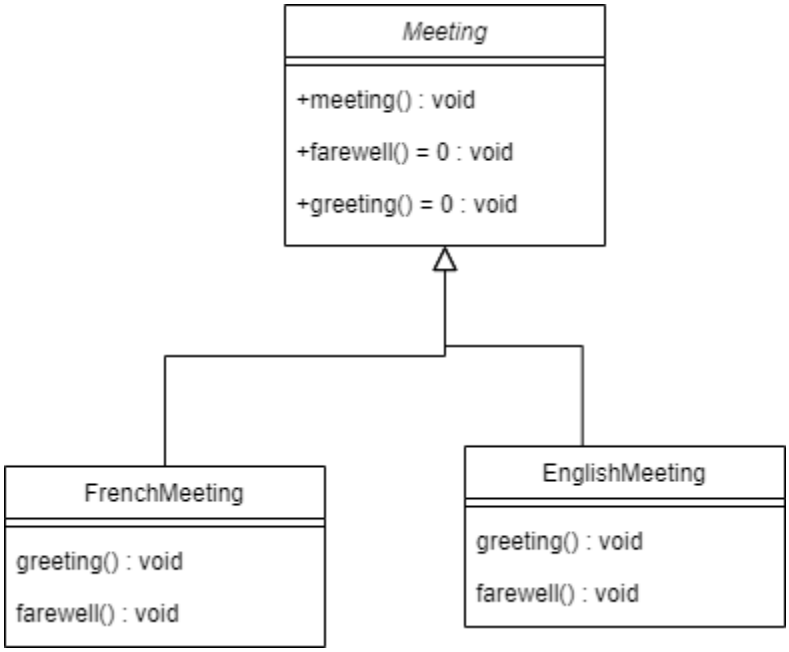
<예시 14-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 14-1>

위 코드는 누군가를 만났을 때 인사와 헤어졌을 때의 인사를 언어별로 출력하는 프로그램이다. 누군가를 만났을 때 “반가워”라는 인사와 헤어질 때의 “잘 가” 라는 인사는 정해진 순서대로 출력되어야 하기 때문에 Meeting()클래스에서 각각의 greeting()함수와 farewell()함수를 순서대로 호출해주는 meeting()함수가 존재한다. 언어마다의 파생 클래스는 Meeting클래스를 상속받으므로 파생클래스에서의 meeting()함수를 호출하게 되면 각 언어에 맞는 인사말이 순서대로 출력되게 된다.

<예시 14-2> <예시 14-1> 의 클래스 다이어그램이다.

<예시 14-2>



③ 스트래티지 패턴(Strategy Pattern)

스트래티지 패턴이란 전략을 쉽게 바꾸어 적용할 수 있게끔 하는 패턴이다. 전략이라 함은 어떠한 기능을 클래스로 캡슐화 한 그러한 것들이라 할 수 있다. 예를 들어 이러한 패턴은 게임 내의 점수 시스템에서 적용될 수 있는데, 플레이어가 체크포인트에서 목표물에 발사체를 맞추어야 점수를 획득 하는 게임에서는 플레이어에 따라 서로 다른 점수 시스템이 사용될 수 있다. 이러한 상황에서는 체크포인트에서 목표물에 발사체를 맞춰야 하는 시스템은 동일하지만 플레이어에 따라 다른 알고리즘을 전략적으로 선택하고 적용되어야 한다. 이러한 상황에서 스트래티지 패턴을 사용하면 런타임 중에 유연하게 해당 알고리즘을 선택할 수 있을 것이다.

스트래티지 패턴의 장점은 상황에 따라 알맞은 알고리즘을 선택해야 하는 경우 매우 유용하게 사용할 수 있는 패턴이다. 디자인 특성상 알고리즘을 분리하여 캡슐화 하였기 때문에 새로운 기능의 확장이 용이하다는 장점 또한 지닌다. 추가적으로 이와 비슷하게 캡슐화 하여 분리하는 패턴들의 장점들을 갖고 있다고 할 수 있다. (코드의 재사용성, 유연성, 유지보수가 유용해진다 등)

스트래티지 패턴의 경우 상황에 따라 유연하게 전략을 선택할 수 있지만, 이러한 전략은 사용자가 상황에 따라 직접 선택하여야 하기 때문에 알고리즘의 전반적인 내용을 알고 있어야 한다. 따라서 스트래티지 패턴은 사용자 편의성 면에서는 좋지 못한 부분이 있는 것이다.

스트래티지 패턴의 경우 패턴의 특성을 이해하기가 까다로웠다. 패턴의 구조가 명확히 정의 되어있지 않아 사람에 따라 상황에 따라 다르게 적용되다 보니 예시 코드만을 보고는 패턴의 특징들을 유추하기란 까다로운 일이었다. 다만 스트래티

지 패턴의 공통적인 특징은 사용자가 상황에 따라 유연하게 전략을 수정하고 적용할 수 있게끔 하는 패턴임을 인지하는 데에는 큰 어려움이 없었다.

<예시 15-1> 코드는『**포르잔 C++ 바이블**』책에서 소개된 코드이다. 코드가 매우 간결하여 이것을 클래스 다이어그램으로 그릴 경우 스트래티지 패턴의 특징을 전혀 알 수가 없다. 따라서 뒤에 소개할 <예시 15-2> 의 클래스 다이어그램은 코드의 클래스 다이어그램이 아닌 책에 소개된 스트래티지 패턴의 구조에 관한 클래스 다이어그램이다.

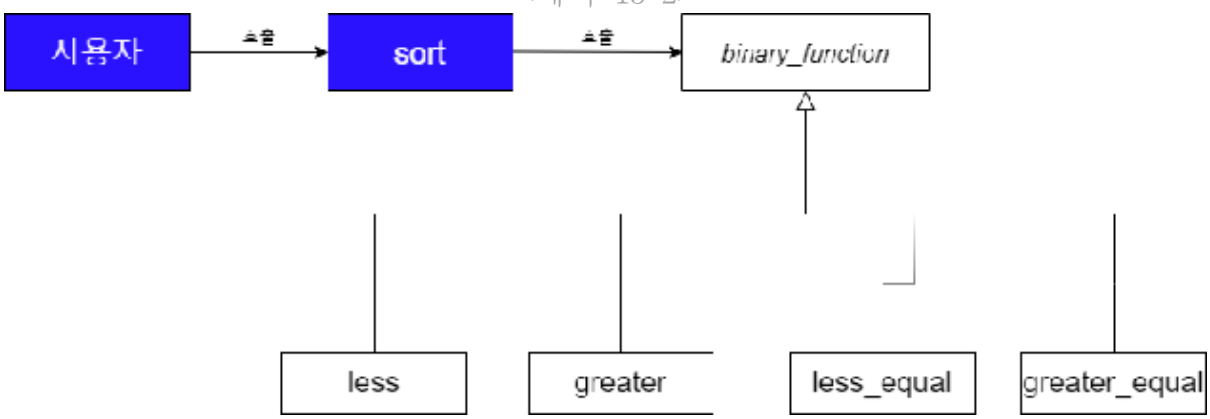
<예시 15-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 15-1>

위 코드는 여러 개의 숫자 데이터를 사용자의 입맛에 맞게 sort()함수를 사용하여 오름차순 또는 내림차순을 선택하여 그에 맞게 출력해주는 코드이다. 코드가 매우 단순해 보이지만 상황에 따라 함수를 통해 오름차순 또는 내림차순으로 결정하는 부분이 스트래티지 패턴의 아이디어가 적용된 모습이다.

<예시 15-2> 스트래티지 패턴에 관한 클래스 다이어그램이다.

<예시 15-2>



④ 커맨드 패턴(Command Pattern)

커맨드 패턴은 단어 그대로 명령을 캡슐화 하여 주어진 여러가지 기능들을 실행할 수 있게끔 하는 재사용성이 높은 디자인 패턴이다. 주로 버튼을 클릭하는 등의 GUI 프로그래밍에서 많이 사용된다.

이러한 커맨드 패턴의 장점으로는 명령을 캡슐 화하여 독립적으로 관리할 수 있기 때문에 명령을 수정하거나 추가적인 기능을 넣기에 매우 용이해진다 또한

원하는 기능을 실행하거나 다시 실행할 수도 있고, 원하지 않을 때는 실행을 취소할 수도 있다. 이러한 경우 명령이 캡슐화 되어 있기 때문에 사용성 측면에서 많은 이점이 생긴다.

커맨드 패턴의 단점으로는 클래스의 수가 너무 많아진다는 데에 있다. 명령을 받아 수행하는 리시버의 기능을 추가하고 싶다면 이 기능에 대한 클래스를 새롭게 만들어야 하는데 이렇게 되면 잡다한 클래스들이 많아져 코드의 복잡도가 증가한다.

커맨드 패턴의 의도는 사용자가 해당 기능을 사용하기 위해 캡슐화된 객체를 사용하여 명령을 내림으로써 하나의 클래스에 대한 의존에서 벗어나 원하는 명령을 사용할 수 있게 하는 것이다. 다만 필자는 해당 패턴을 조사하면서 유지보수에 관한 문제점이 분명 생길 것이다 라는 생각이 들었다. 호출을 보내는 부분과 받는 부분, 그리고 실행되는 부분까지 서로 유기적으로 동작하기 때문에 복잡성이 증가하여 해당 오류를 찾기도 힘들뿐더러 코드를 이해하기에도 매우 불리할 것 같아 자주 사용하지는 않을 것 같다.

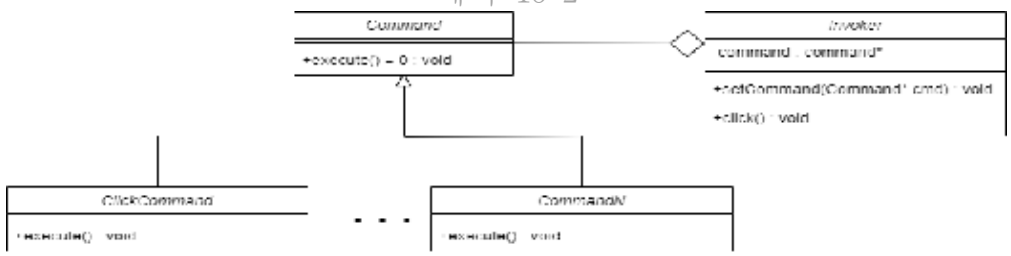
<예시 16-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 16-1>

위 코드는 명령을 호출하는(Invoker) 클래스와 명령을 받고 수행하는 ClickCommand 클래스가 존재하며 인터페이스인 Command 가 존재한다. 버튼을 누르는 명령을 ClickCommand 로 캡슐화하여 실질적인 호출부와 명령을 처리하는 부분이 나뉘어져 의존성을 낮추었다. 사용자는 ClickCommand 의 객체를 만들고 Invoker 에서의 setCommand 함수의 인자로 객체를 넘긴다. 준비가 완료되면 click()함수로 명령을 호출하면 “버튼이 클릭됨!” 이라는 메시지가 출력된다. (이 코드에서는 ClickCommand 클래스가 Receiver 의 역할까지 수행하고 있다.)

<예시 16-2> <예시 16-1> 코드의 클래스 다이어그램이다.

<예시 16-2>



⑤ 옵저버 패턴(Observer Pattern)

옵저버는 관찰자를 뜻한다. 객체의 상태 변화를 관찰하여 상태가 변화할 경우 메서드를 통해 observer 객체에 메시지를 보내 이를 통지하는 디자인 패턴이다.

만약 증권상품들 같이 실시간으로 변하는 가격에 대해서 내가 원하는 가격이 왔을 때 알림을 받는 것이 그의 예가 될 수 있다. 이러한 알림을 모든 사항에 대해 받는 것이 아니기 때문에 구독 형태(subscribe)로 원하는 객체를 구독하여 객체의 변경이 발생할 때 알림을 받을 수 있다. 만약 알림을 받고 싶지 않다면 구독 취소(unsubscribe)를 하여 알림을 받지 않을 수 있다.

이러한 디자인 방식의 장점은 실시간으로 한 객체에 대한 변경 사항을 다른 객체로 전파할 수 있으며 객체의 상태 변화를 별도의 함수 호출 없이 실시간으로 알 수 있기 때문에 특정한 이벤트에 관련된 처리가 많은 프로그램에서 유용하게 사용할 수 있다.

이러한 패턴 단점으로는 자주 사용하게 되면 코드가 복잡 해져 가독성이 저하될 우려가 있으며 유지보수의 어려움이 발생할 수 있다.

일반적으로 게임에서나 여러 프로그램에서 자주 사용되는 요소가 랜덤 요소이다. 특정한 확률로 해당 이벤트를 발생시키는 역할을 할 때 사용하는데 이러한 방식의 코드를 디자인할 때 옵저버 패턴이 매우 유용하게 쓰일 듯하다. 하지만 비슷한 시스템을 여러 개를 만들어야 한다면 가독성이 떨어지기 때문에 필자는 이러한 이벤트들이 많은 상황에서의 사용은 꺼려진다.

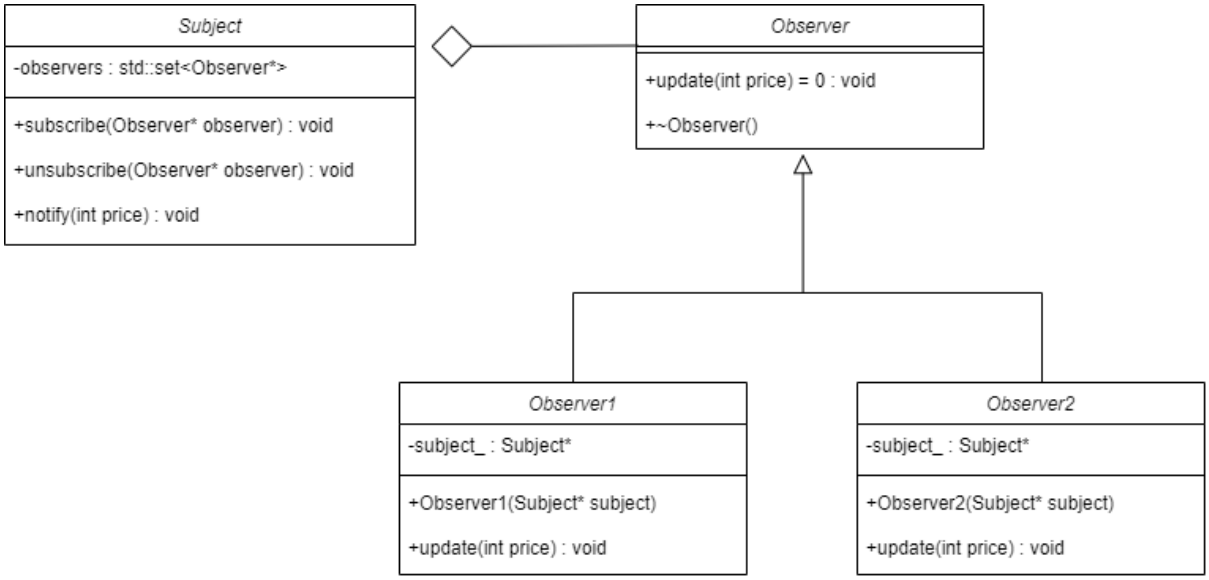
<예시 17-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 17-1>

위 코드는 소비자가 물건을 구매하려고 할 때 물건의 가격이 원하는 가격 이하로 떨어지게 되면 이것을 구매자에게 알리는 코드이다. 소비자가 물건의 판매자에게 구독을 요청하게 되면 물건의 판매자는 값의 변화를 실시간으로 모니터링하다가 조건에 해당되면 구매자에게 알림을 발송한다. 위 코드에서의 판매자는 Subject 클래스이고 구매자는 여러 명이 될 수 있기 때문에 각각 Observer1, Observer2 클래스이다. 실시간으로 변하는 데이터를 표현하기 위해 while 반복문을 사용하여 rand()함수의 시드 값을 초기화 시켜주면서 물건의 값을 변경한다. 만약 물건의 값인 price 변수가 40 보다 작다면 반복을 중단하고 suject 클래스의 notify 함수를 호출하여 소비자에게 이 사실을 알린다.

<예시 17-2> <예시 17-1> 코드의 클래스 다이어그램이다.

<예시 17-2>



⑥ 상태 패턴(State Pattern)

상태 패턴은 어떠한 상태가 변경되었을 때 그에 맞게 다르게 변해야하는 상황이 있다. 예를 들자면 전구의 스위치를 off 상태로 바꾸었을 때 불은 꺼져야 하고, 스위치를 off 상태로 바꾸었을 때는 불은 켜져야 한다. 이렇듯 이렇듯 어떠한 상태가 변경되었을 때 그에 맞는 기능을 하는 디자인 패턴이 상태 패턴이다. 꼭 상태 패턴을 사용하지 않아도 구현이 가능하다. 가장 원초적인 방식으로 어떠한 조건을 검사하여 참이나 거짓일 때의 행동을 다르게 하는 것은 이미 과거에 많이 해왔던 방식들이었다.

상태 패턴의 장점은 하나의 객체에 대한 동작을 구현할 때 상태 객체만을 수정하므로 코드의 확장성과 유지보수가 간편해진다. 또한 과거의 방식처럼 조건문을 무분별하게 사용해야 하는 방식은 가독성이 매우 떨어지고 코드가 간결하지 못하기 때문에 이 또한 상태 패턴의 장점이라고 할 수 있다.

상태 패턴은 조건문 대신 state 객체의 수를 증가시키기 때문에 관리해야 할 클래스의 수가 증가한다. 조건문을 마구자비로 사용한 코드 보다는 가독성이 좋겠지만 관리해야 할 클래스의 수가 계속 증가하는 것은 결코 좋을 수는 없을 것이다. 조건문과 상태 패턴을 적절한 상황에 알맞게 사용하는 것이 매우 중요한 개발자의 역량일 것이다.

필자는 상태 패턴을 조사하면서 이러한 디자인 패턴이 '패턴' 으로서 만들어질 정도로 실용적인 것인지 의문이 들었다. 어떤 상태에 대한 것을 캡슐화 하여 관리하는 것이 코드를 활용하는 데에 있어서는 장점일 수도 있겠지만, 결국 이러한 패턴도 조건들이 많아지고 다양해지다 보면 필연적으로 조건문을 사용해야하는 순간이 올 것이라는 생각이 들었다. 객체의 상태마다 모두 정의하여 클래스를 만드는 것은 코드가 복잡해지기만 하는, 득보다 실이 많은 방식이 아닐까 고민을 해보았다.

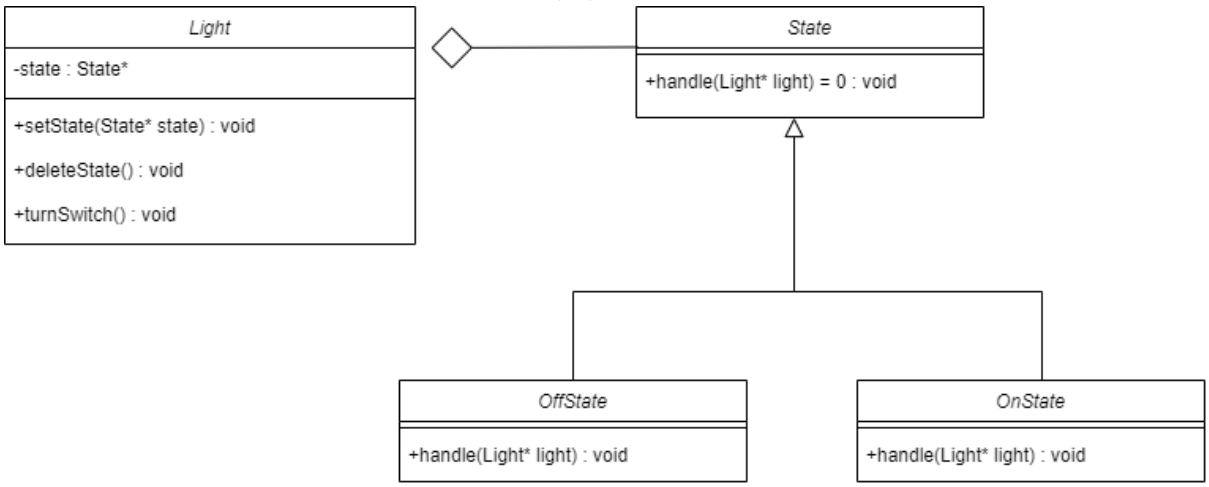
<예시 18-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 18-1>

위 코드는 조건문 대신 state 객체들을 활용하여 현재 불이 켜져 있으면 “불을 끕니다” 불이 꺼져 있으면 “불을 켭니다”를 출력하는 코드이다. 조건문을 사용하지 않기 때문에 현재의 상태를 상태 객체를 생성하는 방식으로 판단하게 된다. 만약 불이 꺼져 있다면, 해당 상태 객체를 delete 시키고 새롭게 켜져 있는 상태 객체인 OnState 를 할당한다.

<예시 18-2> <예시 18-1> 코드의 클래스 다이어그램이다.

<예시 18-2>



⑦ 메멘토 패턴(Memento Pattern)

메멘토 패턴은 변경된 기능이 다시 원래 상태로 돌아가야 할 때 사용하는 패턴이다. 예를 들면 특정한 객체의 상태가 변경되었는데, 변경되기 전의 객체의 상태로 돌아가야 할 때 이러한 패턴을 적용할 수 있다. 이와 비슷하게 복사생성자를 활용하여 객체를 복사시키고 만드는 방법이 있지만, 객체가 변경될 때마다 복사를 해야하는 번거로움이 존재한다. 이러한 일련의 과정들을 자동화시킨다면 사용자는 매우 편리하게 객체를 수정하고 다시 되돌릴 수 있을 것이다.

메멘토 패턴의 장점은 직접 원본 객체를 복사시켜놓고 변경하지 않아도 객체가 수정될 때 마다 이전의 객체가 저장된다는 점에 있다. 또한 원할 때 마다 함수를 호출하여 매우 편하게 원래의 객체로 되돌릴 수 있다는 점에서 큰 장점을 갖는다.

하지만 이러한 방식의 큰 단점이 있는데, 무조건적으로 객체의 이전 데이터가 저장되는 형식이기 때문에 사용자가 원하지 않아도 이전의 객체가 계속해서 저장된 상태로 존재한다. 따라서 매우 복잡하고 데이터의 양이 많은 객체를 이러한 패턴으로 디자인하게 되면 여러 객체를 만들고 수정했을 때 2 배의 메모리 공간을 차지하게 되기 때문에 이는 메모리 낭비로 이어질 수 있다.

메멘토 패턴은 잘만 활용한다면 매우 편리한 디자인 패턴으로 활용될 수 있을 것 같다. 항상 복사생성자를 호출하여 사용하였기 때문에 복사시켜놓는걸 간혹 깜빡하여 뒤로 가기 기능으로 이전 코드를 복사, 가져오는 형태로 사용 하다 보니 매우 비효율적이였다. 이러한 패턴을 사용하면 바로바로 이전의 객체 데이터가 저장되기 때문에 프로그램을 구성하는데 한결 수월해질 듯하다. 하지만 필자의 주관적인 패턴의 단점은 원하는 시점의 객체 데이터를 가져오거나 저장할 수는 없는 것이 아쉬운 점이었다. 한번 객체의 데이터가 변경되면 바로 저장되는 형태로 구성되어서 이러한 구조를 다층구조로 만들어 시점을 2 개 ~ 3 개로 늘리거나 또는 원하는 시점을 함수로 저장하게끔 하는 방식도 좋지 않을까 하는 생각을 해보았다.

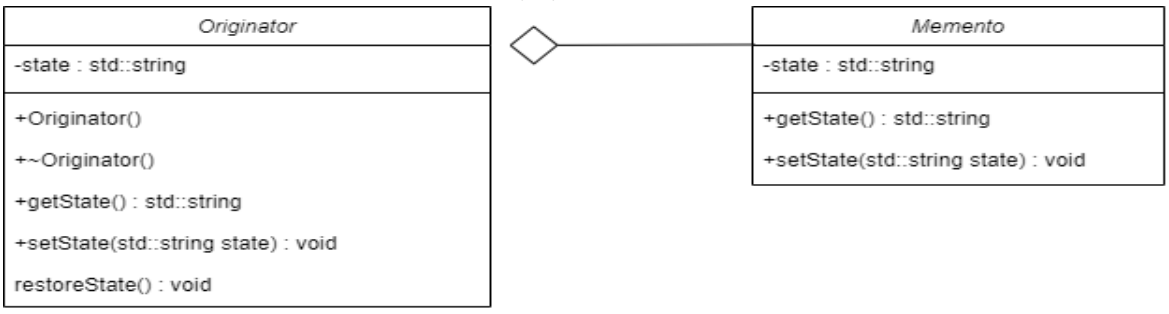
<예시 19-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 19-1>

위 코드는 originator 객체 안에 “안녕하세요”라는 문자열을 저장하고, 다시 한번 “안녕” 이라는 문자열을 저장한다. 그러면 객체의 문자열에는 “안녕하세요” 라는 이전의 문자열 데이터가 지워지고 새로운 “안녕” 이라는 문자열이 덮어씌워지게 될 것이다. 여기서 “안녕하세요” 라는 문자열은 새로 할당된 객체 안에 문자열에 저장되게 되고 다시 restoreState()함수를 호출하게 되면 저장되어있던 객체의 문자열을 불러와 현재의 객체 문자열에 넣어주고 출력한다.

<예시 19-2> <예시 19-1> 코드의 클래스 다이어그램이다.

<예시 19-2>



⑧ 책임 연쇄 패턴(Chain-of Responsibility Pattern)

책임 연쇄 패턴은 핸들러들의 연속적인 호출에 따라 요청을 전달할 수 있게 해주는 패턴이다. 이러한 패턴은 일련의 객체들에게 순서대로 처리를 요청하여 만약 요청을 처리할 수 없다면, 다음 객체로 전달되는 방식의 디자인 패턴이다. 예를 들어 은행마다 대출할 수 있는 최대 금액이 정해져 있다고 가정해보면, 대출자가 원하는 대출 금액을 요청했을 때 은행마다 차례대로 대출 금액과 최대 대출 가능 금액을 대조하여 첫번째 은행이 안될 경우 두번째 은행으로 넘기고 두번째 은행도 대출이 불가하면 세번째 은행으로 넘겨 최종적으로 가능한 은행이 오면 멈추는 방식이다.

이러한 패턴의 장점은 요청을 처리하는 과정에서 그것을 담당할 노드를 동적으로 변경이 가능하다 또한 이러한 과정들에 노드를 추가하여 요청되는 처리를 확장할 수도 있다. 추가적으로 발신자와 수신자가 분리됨으로 인해 사용자는 노드의 집합 내부 구조를 알 필요가 없어진다

사용자의 요청을 노드들이 연결된 체인에서 순차적으로 처리하기 때문에 만약 이러한 체인에 노드들이 많이 존재한다면 요청 처리 과정이 매우 느려질 수 있다. 또한 연쇄적으로 처리되다 보니 디버깅에 있어 번거로움이 생긴다.

책임 연쇄 패턴은 일반적인 프로그래밍에서 자주 사용하기에는 규모가 너무 방대한 면이 있는 것 같다. 범위 하나를 판단할 때 마다 하나의 객체를 생성하고 이를 다음 객체로 넘기고 지우는 작업을 반복 하기 때문에 프로그램의 성능 저하가 우려되고 많은 클래스들을 관리해야 하기 때문에 필자는 이러한 패턴이 소규모에서는 비효율적이지 않을까 하는 생각을 해보았다.

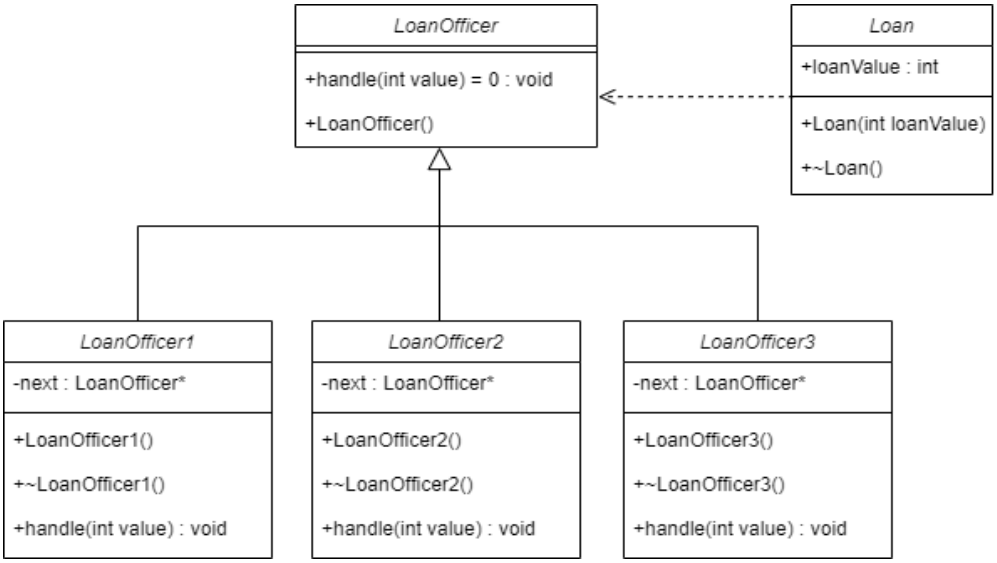
<예시 20-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 20-1>

위 코드는 앞서 설명한 예시처럼 대출자가 대출하려는 금액을 노드들이 처리할 수 있는지를 검사해 해당되면 “대출가능” 해당되지 않으면 다음 노드로 넘기는 방식으로 작동하는 코드이다. 만약 끝까지 처리할 수 없는 값이면, “해당 대출은 불가능합니다.” 를 출력한다.

<예시 20-2> <예시 20-1> 코드의 클래스 다이어그램이다.

<예시 20-2>



⑨ 중재자 패턴(Mediator Pattern)

중재자 패턴은 객체간의 상호작용을 캡슐화 하여 객체간의 직접적인 통신을 제한하고 중재가 객체를 통해서만 통신하도록 하는 디자인 패턴이다. 예를 들면 사장이 모든 직원에게 메시지를 보낸다고 가정했을 때 모든 직원에게 하나하나 전부 발송하는 것이 아닌 일괄적으로 처리하기 위해 중재자를 통해 메시지를 전달하면 중재자는 모든 직원에게 해당 메시지를 보낼 수 있을것이다.

이렇게 객체간의 통신을 막고 중재자를 통한 통신의 이점은 객체들간의 직접적인 통신을 하지 않기 때문에 객체간의 결합을 느슨하게 할 수 있어 유지 보수에 이점이 있다. 또한 다른 패턴과 마찬가지로 캡슐화를 통해 서로간의 의존 관계가 없기 때문에 새로운 객체를 생성함에 있어 유연해질 수 있다.

중재자 패턴의 단점으로는 각각의 객체끼리는 캡슐화 되었지만, 모든 의존도가 중재자로 집중되므로 중재자 클래스의 권한이 매우 커지며 이는 중재자 클래스의 잘못된 설계나 혹은 수정이 필요할 경우 매우 힘들어질 수 있다는 것을 의미한다.

중재자 패턴은 객체끼리의 직접적인 통신을 막고 중재자를 통한 통신만을 허용하는 패턴이다. 필자가 생각한 이러한 특징을 갖는 대표적인 사례가 있는데, 바로 인터넷 통신이다. 지금 당장 집에 존재하는 모뎀 공유기 또한 중재자의 역할을 하고 있기 때문이다. 사용자들 간의 직접적인 통신을 허가하면 위험해질 수 있기 때문에 공유기를 한번 거침으로써 허용되지 않은 포트로의 접근을 차단해주는 등 중재자는 필터의 역할도 하고 있었다. 이와 반대로 중계 서버가 없는 p2p 전송 방식은 중재자 패턴이 적용되지 않은 사례이지 않을까 하는 생각을 해보았다.

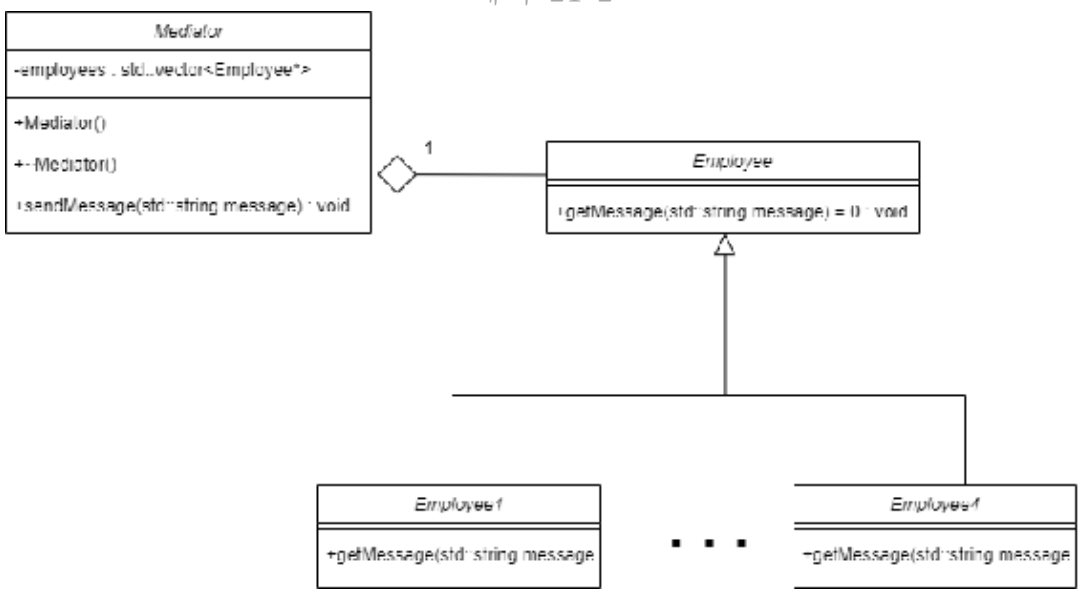
<예시 21-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 21-1>

위 코드는 직원이 sendMessage()함수를 통해 메시지를 중재자 객체로 넘기면 중재가 객체는 Employee1 ~ Employee4 에 해당하는 직원들에게 일괄적으로 메세지를 보내주는 기능의 프로그램이다. 이 역시 중재자 패턴으로 중재자 클래스인 Mediator 가 존재하며 Mediator 안에는 메시지를 보낼 수 있는 기능인 sendMessage 함수가 구현되어 있다.

<예시 21-2> <예시 21-1> 코드의 클래스 다이어그램이다.

<예시 21-2>



⑩ 방문자 패턴(Visitor Pattern)

방문자 패턴은 개발자가 만든 공간을 사용자가 방문의 목적으로 들어오길 희망한다. 방문자는 방문의 목적이기 때문에 마음대로 원래의 것을 수정하거나 제거할 수 없다. 이와 같은 맥락으로 방문자 패턴은 캡슐화의 원칙과 다형성의 원칙을 방문자가 위반하는 것을 원치 않을 때 즉, 객체의 구현을 바꾸지 못하게 하고 싶을 때 사용하는 패턴이다.

방문자 패턴의 장점은 방문하는 장소와 그곳에서 하는 일을 분리하여 데이터의 독립성을 높여준다. 또한 객체의 구조를 수정하지 않고 새로운 기능을 쉽게 추가하여 확장할 수 있다.

방문자 패턴의 단점으로는 방문 장소가 새로 추가될 때마다 이에 맞는 로직을 방문자도 추가해주어야 하기 때문에 불편함이 발생한다. 이렇게 방문자와 방문 장소의 결합이 생겨 문제가 발생할 여지가 있다.

필자가 생각하는 방문자 패턴의 단점은 이렇게 캡슐화를 진행하였을 때는 개발자의 코드가 매우 잘 숨겨지기 때문에 코드의 노출을 최소화 할 수 있다는 장점은 있지만 방문자 패턴을 사용하게 되면 코드의 재사용성 면에서나 어떠한 특정 상황에 대한 유연한 대처가 힘들어지지 않을까 하는 생각이 든다.

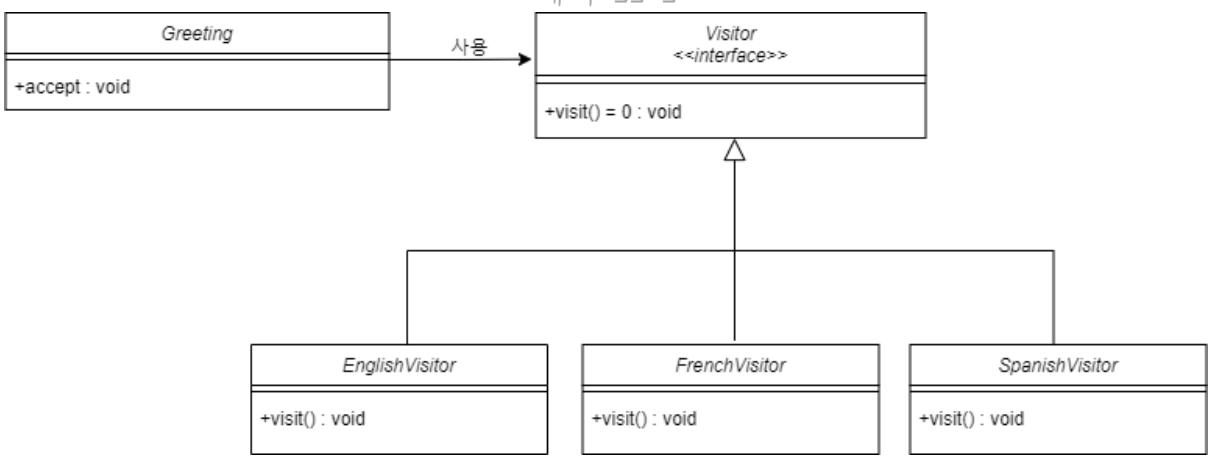
<예시 22-1> 은 C++로 작성된 간단한 예시 코드이다. (부록 참고)

<예시 22-1>

위 코드는 방문자(visitor)가 accept()함수의 인자로써 들어가게 되면서 해당 언어에 맞는 인사를 출력하는 프로그램이다. 방문자 패턴의 특징으로 공간과 방문자가 분리 되어있기 때문에 해당 코드에서도 greeting과 visitor가 분리되어 있는 것을 볼 수 있다. 코드를 파일 하나에 작성하였기 때문에 모든 코드가 노출되어있는 것 처럼 보이지만, 분할 컴파일을 하게되면 greeting 헤더파일이 숨겨지게 되면서 사용자는 해당 함수에 (accept()) 방문자 객체를 넘김으로써 기능을 사용할 수 있게 된다

<예시 22-2> <예시 22-1> 코드의 클래스 다이어그램이다.

<예시 22-2>



⑪ 인터프리터 패턴(Interpreter Pattern)

인터프리터 패턴은 어떠한 문법적인 규칙을 클래스로 만든 것으로 하나의 언어를 문법적인 규칙에 따라 해석하는 것을 목적으로 디자인하는 패턴을 의미한다. 이러한 인터프리터 패턴은 반드시 언어의 해석 과정이 필요한 경우 사용되는데 예를 들면 프로그램이 시스템에서 정상적으로 작동되려면 고급 언어로 쓰여진 코드가 기계어로 해석되어지는 과정이 반드시 필요하다. 이러한 해석에 있어 문법적인 규칙들을 디자인하는 것이 인터프리터 패턴이다.

인터프리터 패턴은 주로 언어를 문법적 규칙에 따라 해석하는 것에 목적을 두고 있기 때문에 문법이 클래스 형태로 표현되어 있어 언어를 쉽게 변경하거나 확장할 수 있다. 또한 개발자가 새로운 문법을 정의하여 도입하는 것이 가능하다.

인터프리터 패턴의 단점으로는 복잡한 문법 구조를 갖는 언어 에서는 많은 객체를 생성해야 할 필요성이 있기 때문에 프로그램의 성능 저하로 이어질 수 있다.

인터프리터 패턴에 대한 필자의 생각은 특정한 상황에서만 필요한 디자인 방식이라는 점이다. 일반적인 개발자는 컴파일러와 같이 매우 low 한 프로그램을 개발해야 하는 일이 잘 없기 때문에 이러한 패턴이 실질적으로 많이 쓰일지에 대한 의문이 들었다. 만약 게임에서의 특정한 아이템 객체에 코드가 존재하고 이를 코드로써 해석하는 단순하면서도 당연하게 여겨지는 것들 또한 인터프리터 패턴의 하나의 예시라고 한다면, 이미 자주 접하고 있는 패턴일지도 모르겠다.

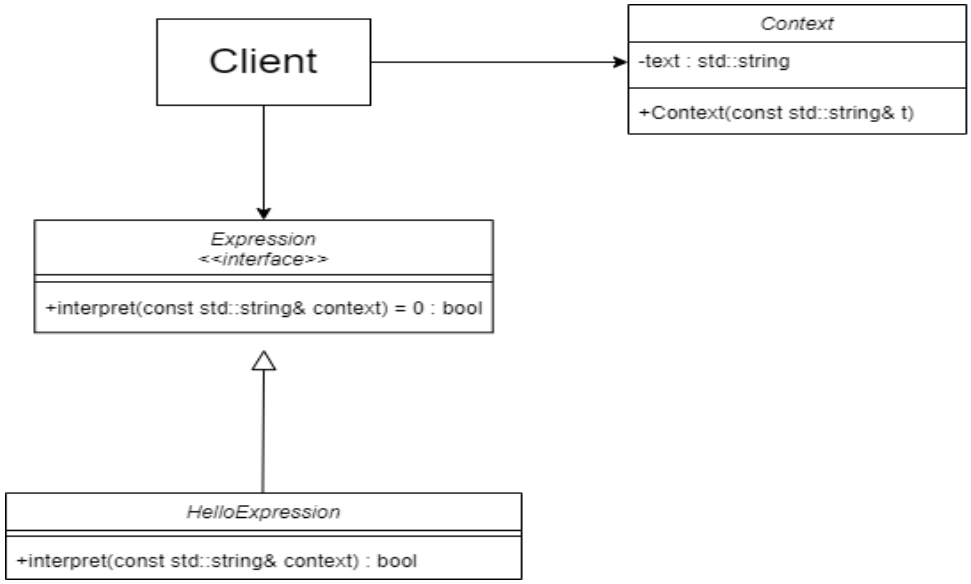
<예시 23-1> 은 C++로 작성된 예시 코드이다. (부록 참고)

<예시 23-1>

위 코드는 사용자가 문자열을 입력하면 문자열을 검사하여 만약 “hello”라는 문자열이 포함되어 있다면 "hello is here!!" 이라는 메시지를 출력하고 존재하지 않는다면 “No hello”메시지를 출력한다.

<예시 23-2> <예시 23-1> 코드의 클래스 다이어그램이다.

<예시 23-2>


Client
Context
-text : std::string
+Context(const std::string& t)
Expression
<<interface>>
+interpret(const std::string& context) = 0 : bool
HelloExpression
+interpret(const std::string& context) : bool

Ⅲ. 결론

1. 고찰

패턴들에 대해 조사하면서 스스로 많이 부족함을 느꼈다. 디자인 패턴에 관하여 책의 첫 장을 폈을 때는 코드가 무엇을 의미하는지 무슨 이야기를 하는 것인지 목적조차 알지 못했다. 클래스와 객체들은 서로 얽혀 있어 예시 코드를 봐도 해당 패턴에 관한 의미를 추측하기란 힘든 일이었다. 이러한 상황 속에서 예시코드를 찾고 그것을 이해하여 클래스 다이어그램을 그리는 것 또한 쉽지 않은 작업이었다. 최대한 아는 것을 동원하여 코드에 알맞게 uml 을 만들었지만 아직도 이것이 맞는 것인지 자신이 없다. 계속해서 모든 패턴들의 조사를 마치고 나서야 비로소 책에 수록되어 있는 코드들이 매우 간단하게만 소개된 코드였다는 사실을 인지할 수 있었다. 이번 패턴들에 대해 조사를 진행하면서 클래스다이어그램 부터 앞으로 마주쳐야 할 알고리즘의 문제들까지 간접적으로 경험할 수 있는 계기가 되었다.

2. 한계점

직접 패턴들마다 이해하면서 코드를 직접 작성해보려 하였지만 한계에 부딪혔다. 더군다나 내가 작성한 코드의 클래스 다이어그램이 맞는 것일까? 하는 의문은 지워지질 않았다. 그나마 가장 정확하게 작성할 수 있는 방법은 책에 있는 코드를 이해하여 내것으로 만든 후 이미 수록된 클래스 다이어그램을 좀 더 정확하게 그리는 정도에 그쳤다. 책에서 모든 패턴들을 다루지는 않았다. GoF 의 디자인 패턴은 23 가지이지만 책에는 패턴들마다 하나가 빠져 있다. 이것을 조사하는 과정에서 정말 정확한 것인지에 대한 의문은 아직도 남아있다. 해당 디자인 패턴의 설계 목적 까지는 이해했으나 예시 코드를 100% 정확하게 이해하지 못한 상태에서 조사를 이어 나갔다. 이러한 불확실성에 직면하면서 자료를 모으다 보니 레포트의 전반적인 정확도가 다소 떨어질 것으로 예상되어진다.

Ⅳ. 참고문헌

▶Refactoring Guru. "Design Pattern." Refactoring Guru, https://refactoring.guru/ko/design-patterns/abstract-factory

▶BlackWasp. "Factory Method Pattern in C#." BlackWasp, https://www.blackwasp.co.uk/Builder.aspx

▶ChachaYelmo. "Design Pattern Builder." https://chachayelmo.github.io/designpattern/designpattern-builder

▶Behrouz A. Forouzan, Richard F. Gilberg.『포르잔 C++ 바이블』. 윤인성(역). 한빛아카데미, 2020.

▶ha0kim. “Structural Patterns.” Ha0kim, https://velog.io/@ha0kim/Design-Pattern-%EA%B5%AC%EC%A1%B0-%ED%8C%A8%ED%84%B4Structural-Patterns.

▶개발자의 메모장: 티스토리. https://leetaehoon.tistory.com/46.

▶invrtd.h. https://invrtd-h.tistory.com/46.

▶user-pw9fm4gc7e. YouTube. https://www.youtube.com/user/user-pw9fm4gc7e

V. 부록

<예시 1-1> - 추상팩토리패턴 예시코드

```
#include <iostream>
#include <memory>

//shape 인터페이스
class shape
{
public:
    virtual void draw() = 0;
    virtual ~shape() { std::cout << "shape destructor has been called" <<
std::endl; }
};

class ellipse : public shape
{
    void draw() override;
};

class circle : public ellipse
{
    void draw() override;
};

class rectangle : public shape
{
    void draw() override;
};

//객체 생성을 위한 추상 팩토리 클래스
class factory
{
public:
    virtual shape* get_object() = 0;
    virtual ~factory() { std::cout << "factory destructor has been called" <<
std::endl; }
};

class ellipse_factory : public factory
{
public:
    shape* get_object() override
    {
        return new ellipse;
    }
```

```
};

class circle_factory : public factory
{
public:
    shape* get_object() override
    {
        return new circle;
    }
};

class rectangle_factory : public factory
{
    shape* get_object() override
    {
        return new rectangle;
    }
};

int main(void)
{
    //ellipse 객체 생성
    std::unique_ptr<factory> my_ellipse_factory(new ellipse_factory());
    std::unique_ptr<shape> my_ellipse_object(my_ellipse_factory->get_object());
    my_ellipse_object->draw();
    //circle 객체 생성
    std::unique_ptr<factory> my_circle_factory(new circle_factory());
    std::unique_ptr<shape> my_circle_object(my_circle_factory->get_object());
    my_circle_object->draw();

    //rectangle 객체 생성
    std::unique_ptr<factory> my_rectangle_factory(new rectangle_factory());
    std::unique_ptr<shape> my_rectangle_object(my_rectangle_factory-
>get_object());
    my_rectangle_object->draw();
    return 0;
}

void ellipse::draw()
{
    std::cout << "The ellipse has been successfully drawn" << std::endl;
}

void circle::draw()
{
    std::cout << "The circle has been successfully drawn" << std::endl;
}
```

```
void rectangle::draw()
{
    std::cout << "The rectangle has been successfully drawn" << std::endl <<
std::endl;
}
```

```
The ellipse has been successfully drawn
The circle has been successfully drawn
The rectangle has been successfully drawn

shape destructor has been called
factory destructor has been called
shape destructor has been called
factory destructor has been called
shape destructor has been called
factory destructor has been called


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 2-1> - 싱글톤패턴 예시코드

```cpp
#include <iostream>
#include <memory>

class mobile_option
{
private:
    mobile_option() { std::cout << "consructor called!!" << std::endl; }
    static mobile_option* mobile_option_object;
    std::string password;
    std::string alarm;
public:
    static mobile_option* get_object();
    void set_password(std::string password);
    void set_alarm(std::string alarm);

    std::string get_password();
    std::string get_alarm();
};
mobile_option* mobile_option::mobile_option_object = nullptr;

int main(void)
{
    mobile_option* my_mobile_option = mobile_option::get_object();
    my_mobile_option->set_password("singleton");
    my_mobile_option->set_alarm("19:30");
    std::cout << "-------------------------------------" << std::endl;
    std::cout << "password: " << my_mobile_option->get_password() << std::endl;
    std::cout << "alarm: " << my_mobile_option->get_alarm() << std::endl;

    std::cin.get();
    return 0;
}


mobile_option* mobile_option::get_object()
{
    if(mobile_option_object != nullptr)
    {
        return mobile_option_object;
    }
    else if(mobile_option_object == nullptr)
    {
        mobile_option_object = new mobile_option;
        return mobile_option_object;
    }
}
```

```
void mobile_option::set_password(std::string password)
{
    this->password = password;
}
void mobile_option::set_alarm(std::string alarm)
{
    this->alarm = alarm;
}

std::string mobile_option::get_password()
{
    return password;
}

std::string mobile_option::get_alarm()
{
    return alarm;
}
```

```
consructor called!!
-------------------------------------
password: singleton
alarm: 19:30
```

<예시 3-1> - 팩토리메서드패턴 예시코드

```
//shape 인터페이스
```

```
class shape
{
public:
    virtual void draw() = 0;
    virtual ~shape() { std::cout << "shape destructor has been called" <<
std::endl; }
};

class circle : public shape
{
    void draw() override;
};

class rectangle : public shape
{
    void draw() override;
};

//객체 생성을 위한 추상 팩토리 클래스
class factory
{
public:
    virtual shape* get_object() = 0;
    virtual ~factory() { std::cout << "factory destructor has been called" <<
std::endl; }
};

class circle_factory : public factory
{
public:
    shape* get_object() override
    {
        return new circle;
    }
};

class rectangle_factory : public factory
{
    shape* get_object() override
    {
        return new rectangle;
    }
};

int main(void)
{
    //circle 객체 생성
    std::unique_ptr<factory> my_circle_factory(new circle_factory());
```

```
    std::unique_ptr<shape> my_circle_object(my_circle_factory->get_object());
    my_circle_object->draw();

    //rectangle 객체 생성
    std::unique_ptr<factory> my_rectangle_factory(new rectangle_factory());
    std::unique_ptr<shape> my_rectangle_object(my_rectangle_factory-
>get_object());
    my_rectangle_object->draw();
    return 0;
}
void circle::draw()
{
    std::cout << "The circle has been successfully drawn" << std::endl;
}

void rectangle::draw()
{
    std::cout << "The rectangle has been successfully drawn" << std::endl <<
std::endl;
}
```

```
The circle has been successfully drawn
The rectangle has been successfully drawn

shape destructor has been called
factory destructor has been called
shape destructor has been called
factory destructor has been called


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 4-1> - 빌더패턴 예시코드

```
#include <iostream>
```

```
class Computer
{
private:
    std::string mainboard_;
    std::string cpu_;
    std::string power_;
    std::string graphics_;
    std::string case_;
    std::string ssd_;
    int price_;

public:
    void set_mainboard(const std::string& mainboard);
    void set_cpu(const std::string& cpu);
    void set_power(const std::string& power);
    void set_graphics(const std::string& graphics);
    void set_case(const std::string& Case);
    void set_ssd(const std::string& ssd);
    void set_price(int price);

    void print_computer();
};

class ComputerBuilder
{
private:
    Computer computer_;

public:
    ComputerBuilder& set_mainboard(const std::string& mainboard)
    {
        computer_.set_mainboard(mainboard);
        return *this;
    }

    ComputerBuilder& set_cpu(const std::string& cpu)
    {
        computer_.set_cpu(cpu);
        return *this;
    }

    ComputerBuilder& set_power(const std::string& power)
    {
        computer_.set_power(power);
        return *this;
    }

    ComputerBuilder& set_graphi                ring& graphics)
```

```
    {
        computer_.set_graphics(graphics);
        return *this;
    }

    ComputerBuilder& set_case(const std::string& Case)
    {
        computer_.set_case(Case);
        return *this;
    }

    ComputerBuilder& set_ssd(const std::string& ssd)
    {
        computer_.set_ssd(ssd);
        return *this;
    }

    ComputerBuilder& set_price(int price)
    {
        computer_.set_price(price);
        return *this;
    }

    Computer build()
    {
        return computer_;
    }

};

int main(void)
{
    ComputerBuilder computer_builder;

    //첫번째 컴퓨터
    Computer my_computer1 = computer_builder.set_mainboard("MSI MAG
B760M").set_cpu("intel core i5-13 13400f")
    .set_graphics("RTX 4060Ti").set_case("NCORE G30").set_ssd("samsung
970EVO").set_power("MICRONICS Classic 2 850W").set_price(16).build();
    my_computer1.print_computer();

    //두번째 컴퓨터
    Computer my_computer2 = computer_builder.set_mainboard("MSI MAG
B760M").set_cpu("intel core i5-13 13500")
    .set_graphics("RTX 3060 STORM X").set_case("DAVEN D6 MESH").set_ssd("SK
Hynix Gold P31").set_power("SeaSonic new focus gx-1000").set_price(16).build();
    my_computer2.print_computer
```

```
    std::cin.get();
    return 0;
}

void Computer::set_mainboard(const std::string& mainboard)
{
    this->mainboard_ = mainboard;
}
void Computer::set_cpu(const std::string& cpu)
{
    this->cpu_ = cpu;
}
void Computer::set_power(const std::string& power)
{
    this->power_ = power;
}
void Computer::set_graphics(const std::string& graphics)
{
    this->graphics_ = graphics;
}
void Computer::set_case(const std::string& Case)
{
    this->case_ = Case;
}
void Computer::set_ssd(const std::string& ssd)
{
    this->ssd_ = ssd;
}
void Computer::set_price(int price)
{
    this->price_ = price;
}

void Computer::print_computer()
    {
        std::cout << "mainboard: " << mainboard_ << std::endl
        << "cpu: " << cpu_ << std::endl
        << "power: " << power_ << std::endl
        << "graphics: " << graphics_ << std::endl
        << "case: " << case_ << std::endl
        << "ssd: " << ssd_ << std::endl
        << "price: " << price_ << std::endl
        << std::endl << std::endl;
    }
```

```
mainboard: MSI MAG B760M
cpu: intel core i5-13 13400I
power: MICRONICS Classic 2 850W
graphics: RTX 4060Ti
case: NCORE C30
ssd: samsung 970EVO
price: 16


mainboard: MST MAG B760M
cpu: intel core i5 13 13500
power: SeaSonic new focus gx 1000
graphics: RTX 3060 STORM X
case: DAVEN D6 MESH
ssd: SK Hynix Gold P31
price: 16
```

<예시 5-1> - 프로토타입 패턴 예시코드

```cpp
#include <iostream>
#include <memory>

class prototype
{
public:
    virtual std::unique_ptr<prototype> clone() = 0;
    virtual ~prototype() { std::cout << "prototype destructor called" <<
std::endl; }
};

class Car : public prototype
{
private:
    std::string car_name_;
    std::string car_color_;

public:
    Car(std::string car_name, std::string car_color);
    std::unique_ptr<prototype> clone() override;
};

class bus : public prototype
{
private:
    std::string bus_size_;
    std::string bus_color_;

public:
    bus(std::string bus_size, std::string bus_color);
    std::unique_ptr<prototype> clone() override;
};

int main()
{
    std::unique_ptr<prototype> pro1(new Car("세단", "흰색"));
    std::unique_ptr<prototype> pro2(pro1->clone());
    std::unique_ptr<prototype> pro3(new bus("큰", "검정색"));
    std::unique_ptr<prototype> pro4(pro3->clone());

    std::cin.get();
    return 0;
}

Car::Car(std::string car_name, std::string car_color) : car_name_(car_name),
car_color_(car_color)
{
```

```
    std::cout << car_color_ << " " << car_name_ << " car produced." <<
std::endl;
}

std::unique_ptr<prototype> Car::clone()
{
    return std::make_unique<Car>(*this);
}

bus::bus(std::string bus_size, std::string bus_color) : bus_size_(bus_size),
bus_color_(bus_color)
{
    std::cout << bus_color_ << " " << bus_size_ << " bus produced." <<
std::endl;
}

std::unique_ptr<prototype> bus::clone()
{
    return std::make_unique<bus>(*this);
}
```

```
흰색 세단 car produced.
검정색 큰 bus produced.
prototype destructor called
prototype destructor called
prototype destructor called
prototype destructor called
```

<예시 6-1> - 브리지패턴 예시코드

```
#include <iostream>
#include <memory>


class Shape {
protected:
    std::unique_ptr<Color> color;

public:

    Shape(Color* color) : color(color) {}
    virtual void draw() = 0;
    virtual ~Shape() { std::cout << "shape destrctor called" << std::endl; }
};

class Circle : public Shape {
public:
    Circle(Color* color) : Shape(color) {}

    void draw() override {
        std::cout << "circle :";
        color->use_Color();
    }
};

class Rectangle : public Shape {
public:
    Rectangle(Color* color) : Shape(color) {}

    void draw() override {
        std::cout << "rectangle :";
        color->use_Color();
    }
};

class Color {
public:
    virtual void use_Color() = 0;
    virtual ~Color() { std::cout << "color destructor called" << std::endl; }
};

class RedColor : public Color {
public:
    void use_Color() override {
        std::cout << "red" << std::endl;
    }
```

```
};

class BlueColor : public Color {
public:
    void use_Color() override {
        std::cout << "blue" << std::endl;
    }
};
//-----------------------------------------------
int main(void) {
    std::unique_ptr<Shape> redCircle(new Circle(new RedColor()));
    redCircle->draw();

    std::unique_ptr<Shape> blueRectangle(new Rectangle(new BlueColor()));
    blueRectangle->draw();
    std::cin.get();
    return 0;
}
```

```
circle :red
rectangle :blue

shape destrctor called
color destructor called
shape destrctor called
color destructor called


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 7-1> - 어댑터패턴 예시코드

```
#include <iostream>
#include <memory>

class Target
{
public:
    virtual void Target_call() const
    {
        std::cout << "Target called" << std::endl;
    }
    virtual ~Target() { std::cout << "Target destructor called" << std::endl; }
};
class Adaptee
{
public:
    virtual void Adaptee_call() const
    {
        std::cout << "Adaptee called" << std::endl;
    }
    virtual ~Adaptee() { std::cout << "Adaptee destructor called" <<
std::endl; }
};

class Adapter : public Target
{
private:
    Adaptee* adaptee;
public:
    Adapter(Adaptee* Ap) : adaptee(Ap) {  }
    void Target_call() const override
    {
        adaptee->Adaptee_call();
    }
    ~Adapter()
    {
        std::cout << "Adapter destructor called" << std::endl;
    }
};
int main(void)
{
    Target* target = new Target;
    target->Target_call();

    Adaptee* adaptee = new Adaptee;
    adaptee->Adaptee_call();
```

```
//  어댑터를 사용한 호출
    Adapter* adapter = new Adapter(adaptee);
    adapter->Target_call();

    delete adaptee;
    delete adapter;

    return 0;
}
```

```
Target called
Adaptee called
Adaptee called
Target destructor called
Adaptee destructor called
Adapter destructor called
Target destructor called


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 8-1> - 퍼사드패턴 예시코드

```cpp
#include <iostream>

class LivingRoom
{
double size_;
public:
    LivingRoom(double size) : size_(size) { }
    void draw()
    {
        std::cout << "거실의 크기는: " << size_;
        std::cout << "평방 피트로 그립니다." << std::endl;
    }
};

class BedRoom
{
double size_;
public:
    BedRoom(double size) : size_(size) { }
    void draw()
    {
        std::cout << "침실의 크기는: " << size_;
        std::cout << "평방 피트로 그립니다." << std::endl;
    }
};

class Kitchen
{
double size_;
public:
    Kitchen(double size) : size_(size) { }
    void draw()
    {
        std::cout << "부엌의 크기는: " << size_;
        std::cout << "평방 피트로 그립니다." << std::endl;
    }
};

class BathRoom
{
double size_;
public:
    BathRoom(double size) : size_(size) { }
    void draw()
    {
        std::cout << "욕실의 크기는: " << size_;
        std::cout << "평방 피트로 그립니다." << std::endl;
    }
```

```
};

class House
{
double size_;
public:
    House(double size) : size_(size) { }
    void draw()
    {
        LivingRoom living_room(size_ * 0.35);
        living_room.draw();

        BedRoom bed_room(size_ * 0.35);
        bed_room.draw();

        Kitchen kitchen(size_ * 0.15);
        kitchen.draw();

        BathRoom bath_room(size_ * 0.15);
        bath_room.draw();
    }

};

int main(void)
{
    House house(1000);
    house.draw();
    return 0;
}
```

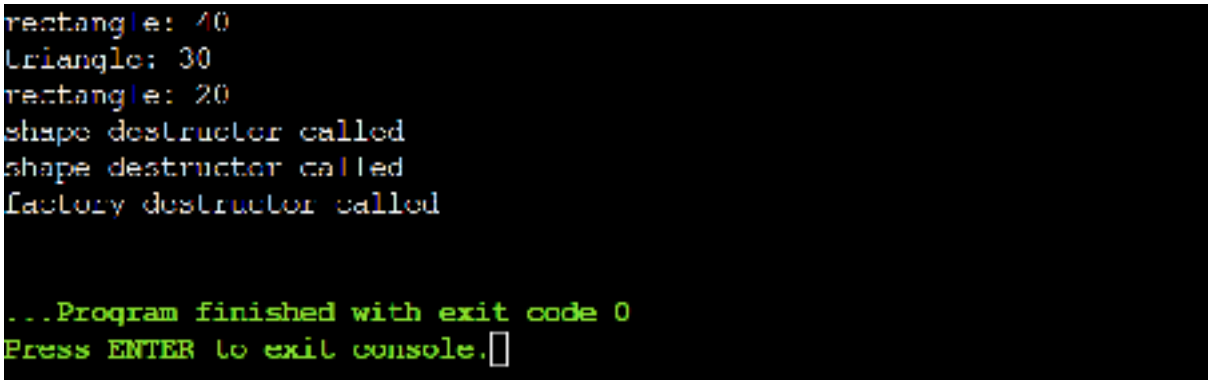
```
거실의 크기는: 350평입니다.
침실의 크기는: 350평입니다.
부엌의 크기는: 150평입니다.
욕실의 크기는: 150평입니다.


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 9-1> - 플라이웨이트패턴 예시코드

```
#include <iostream>
```

```cpp
#include <map>

class shape
{
public:
    virtual void shape_print(int side) const = 0;
    virtual ~shape() { std::cout << "shape destructor called" << std::endl; }
};

class Rectangle : public shape
{
public:
    void shape_print(int side) const override
    {
        std::cout << "rectangle: " << side * 4 << std::endl;
    }
};

class Triangle : public shape
{
public:
    void shape_print(int side) const override
    {
        std::cout << "triangle: " << side * 3 << std::endl;
    }
};

class shape_factory
{
private:
    std::map<std::string, shape*> shapes;
    shape* create_shape(const std::string& shape_name)
    {
        if (shape_name == "rectangle")
        {
            return new Rectangle;
        }
        else if (shape_name == "triangle")
        {
            return new Triangle;
        }
        return nullptr;
    }
public:
    shape* get_object(const std::string& shape_name)
    {
        auto it = shapes.find(shape_name);
```

```
        if (it == shapes.end())
        {
            shape* new_shape = create_shape(shape_name);
            shapes[shape_name] = new_shape;
            return new_shape;
        }
        else
        {
            return it->second;
        }
    }

    const std::map<std::string, shape*>& get_shapes() const
    {
        return shapes;
    }
    ~shape_factory() { std::cout << "factory destructor called" << std::endl; }
};

int main()
{
    shape_factory factory;

    shape* my_rectangle1 = factory.get_object("rectangle");
    my_rectangle1->shape_print(10);

    shape* my_triangle1 = factory.get_object("triangle");
    my_triangle1->shape_print(10);


    shape* my_rectangle2 = factory.get_object("rectangle");
    my_rectangle2->shape_print(5);


    for (const std::pair<std::string, shape*>& pair : factory.get_shapes())
    {
        delete pair.second;
    }
    return 0;
}
```



<예시 10-1> - 프록시패턴 예시코드

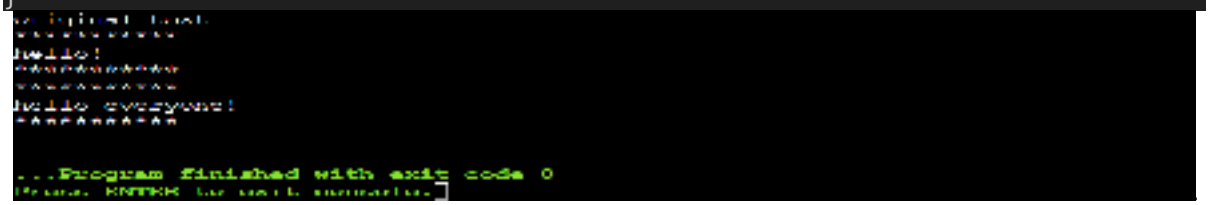
```cpp
#include <iostream>

class Subject
{
public:
    virtual void draw(int x, int y) = 0;
    virtual void erase() = 0;
    virtual ~Subject() {}
};

class Real : public Subject
{
public:
    void draw(int x, int y)
    {
        std::cout << "( " << x << " , " << y << " ) draw" << std::endl;
    }
    void erase()
    {
        std::cout << "object delete" << std::endl;
    }
};

class Proxy : public Subject
{
private:
    Subject* real;
public:
    Proxy()
    {
        real = 0;
    }
    ~Proxy()
    {
        delete real;
    }
    void draw(int x, int y)
    {
        if(real == 0)
        {
            real = new Real;
        }
        real ->draw(x,y);
    }
    void erase()
    {
        real->erase();
    }
```

```
};

int main(void)
{
    Proxy p;
    for(int i = 0; i < 5; i++)
    {
        p.draw(i, i);
        p.erase();
    }

    return 0;
}
( 0 , 0 ) draw
objec delete
( 1 , 1 ) draw
objec delete
( 2 , 2 ) draw
objec delete
( 3 , 3 ) draw
objec delete
( 4 , 4 ) draw
objec delete


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 11-1> - 데코레이터패턴 예시코드

```
#include <iostream>
```

```
class Component
{
private:
    std::string text_;
public:
    Component(std::string text) : text_(text) { }
    void draw()
    {
        std::cout << text_ << std::endl;
    }
};

class Decorator
{
private:
    Component component_;
public:
    Decorator(Component component) : component_(component) { }

    void draw()
    {
        std::cout << "**********" << std::endl;
        component_.draw();
        std::cout << "**********" << std::endl;
    }
};

int main(void)
{
    Component my_text("original text");
    my_text.draw();

    Decorator my_deco1(Component("hello!"));
    my_deco1.draw();

    Decorator my_deco2(Component("hello everyone!"));
    my_deco2.draw();
    return 0;
}
```



<예시 12-1> - 컴포짓패턴 예시코드

```
#include <iostream>
```

```cpp
#include <utility>
#include <vector>

class Figure
{
public:
    virtual void show() = 0;
};

class Point : public Figure
{
private:
    std::pair<double, double> point;
public:
    Point(double x, double y)
    {
        point.first = x;
        point.second = y;
    }
    void show()
    {
        std::cout << "(" << point.first << ", " << point.second << ")" <<
std::endl;
    }
};

class Multipoint : public Figure
{
private:
    int size_;
    std::vector<Figure*> points;
public:
    Multipoint()
    {
        size_ = 0;
    }
    void addPoint(Figure* point)
    {
        points.push_back(point);
        size_++;
    }
    void show()
    {
        for(int i = 0; i < size_ ; i++)
        {
            points[i]->show();
        }
    }
```

```
};

int main(void)
{
    std::cout << "Point 객체 만들고 출력하기" << std::endl;
    Point point(7.77, 2.24);
    point.show();
    std::cout << std::endl;
    std::cout << "Multipoint 객체 만들고 출력하기" << std::endl;
    Multipoint Multipoint;
    Multipoint.addPoint(new Point(3.22, 4.51));
    Multipoint.addPoint(new Point(4.12, 8.32));
    Multipoint.addPoint(new Point(1.12, 7.44));
    Multipoint.show();
    std::cout << std::endl;

    return 0;
}
```

```
Point 객체 만들고 출력하기
(7.77, 2.24)

Multipoint 객체 만들고 출력하기
(3.22, 4.51)
(4.12, 8.32)
(1.12, 7.44)


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 13-1> - 반복자패턴 예시코드

```
#include <iostream>
#include <vector>
```

```cpp
// 반복자 인터페이스
class Iterator
{
public:
    virtual int getNext() = 0;
    virtual bool hasNext() = 0;
};

// 숫자 목록을 저장하는 컬렉션
class NumberCollection
{
private:
    std::vector<int> numbers;

public:
    void addNumber(int number)
    {
        numbers.push_back(number);
    }

    Iterator* createIterator();

    const std::vector<int>& getNumbers() const
    {
        return numbers;
    }
};


class ConcreteIterator : public Iterator
{
private:
    const NumberCollection* collection;
    size_t index;

public:
    ConcreteIterator(const NumberCollection* collection) :
collection(collection), index(0) {}

    int getNext() override
    {
        if (hasNext())
        {
            return collection->getNumbers()[index++];
        }
        return -1;
    }
```

```
    bool hasNext() override
    {
        return index < collection->getNumbers().size();
    }
};

Iterator* NumberCollection::createIterator()
{
    return new ConcreteIterator(this);
}

int main()
{
    NumberCollection numbers;
    numbers.addNumber(11);
    numbers.addNumber(22);
    numbers.addNumber(33);
    numbers.addNumber(44);


    Iterator* iterator = numbers.createIterator();

    while (iterator->hasNext())
    {
        std::cout << iterator->getNext() << " ";
    }
    delete iterator;

    return 0;
}
```

```
11 22 33 44

...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 14-1> - 템플릿메소드패턴 예시코드

```
#include <iostream>
```

```
class Meeting
{
public:
    void meeting()
    {
        greeting();
        farewell();
    }
    virtual void greeting() = 0;
    virtual void farewell() = 0;
};

class EnglishMeeting : public Meeting
{
public:
    void greeting()
    {
        std::cout << "Hello my friends!" << std::endl;
    }
    void farewell()
    {
        std::cout << "Bye my friends!" << std::endl;
    }

};

class FrenchMeeting : public Meeting
{
public:
    void greeting()
    {
        std::cout << "Bonjour mes amis" << std::endl;
    }
    void farewell()
    {
        std::cout << "Au revoir mes amis" << std::endl;
    }
};

int main(void)
{
    EnglishMeeting engMt;
    FrenchMeeting freMt;

    std::cout << "English Greetings: " << std::endl;
    engMt.meeting();
    std::cout << "French Greetings: " << std::endl;
```

```
    freMt.meeting();
    return 0;
}
```

```
English Greetings:
Hello my friends!
Bye my friends!
French Greetings:
Bonjour mes amis
Au revoir mes amis


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 15-1> - 스트래티지패턴 예시코드

```
#include <iostream>
```

```cpp
#include <algorithm>
#include <vector>
#include <functional>
#include <iomanip>

void print(std::string message, std::vector<int> v)
{
    std::cout << message;
    for(int i = 0; i < v.size(); i++)
    {
        std::cout << std::setw(4) << std::left << v[i];
    }
    std::cout << std::endl;
}

int main(void)
{
    std::vector<int> vec;

    vec.push_back(17);
    vec.push_back(10);
    vec.push_back(13);
    vec.push_back(18);
    vec.push_back(15);
    vec.push_back(11);
    print("원본 벡터: ", vec);
    sort(vec.begin(), vec.end(), std::less<int>());
    print("오름차순: ", vec);
    sort(vec.begin(), vec.end(), std::greater<int>());
    print("내림차순: ", vec);
    return 0;
}
```

```
원본 벡터: 17  10  13  18  15  11
오름차순: 10  11  13  15  17  18
내림차순: 18  17  15  13  11  10


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 16-1> - 커맨드패턴 예시코드

```cpp
#include <iostream>
```

```
class Command
{
public:
    virtual void execute() = 0;
};
class ClickCommand : public Command
{
public:
    void execute() override
    {
        std::cout << "버튼이 클릭됨!" << std::endl;
    }
};
class Invoker
{
private:
    Command* command;
public:
    void setCommand(Command* cmd)
    {
        command = cmd;
    }

    void click()
    {
        command->execute();
    }
};
int main()
{
    ClickCommand clickAction;
    Invoker myButton;
    myButton.setCommand(&clickAction);
    myButton.click();

    return 0;
}
```

```
버튼이 클릭됨!


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 17-1> - 옵저버패턴 예시코드

```
#include <iostream>
```

```
#include <set>
#include <string>
#include <cstdlib>
#include <ctime>
class Observer
{
public:
    virtual void update(int price) = 0;
    virtual ~Observer() {}
};
class Subject
{
private:
    std::set<Observer*> observers;
public:
    void subscribe(Observer* observer)
    {
        observers.insert(observer);
    }
    void unsubscribe(Observer* observer)
    {
        observers.erase(observer);
    }
    void notify(int price)
    {
        std::set<Observer*>::iterator iter;
        for(iter = observers.begin(); iter != observers.end(); iter++)
        {
            (*iter)->update(price);
        }
    }
};

class Observer1 : public Observer
{
private:
    Subject* subject_;
public:
    Observer1(Subject* subject) : subject_(subject) { }
    void update(int price)
    {
        std::cout << "Observer1: 현재 가격이 " << price << "이므로 구매합니다."
<< std::endl;
    }
};
class Observer2 : public Observer
{
private:
```

```
    Subject* subject_;
public:
    Observer2(Subject* subject) : subject_(subject) { }
    void update(int price)
    {
        std::cout << "Observer2: 현재 가격이 " << price << "이므로 구매합니다."
<< std::endl;
    }
};

int main(void)
{
    Subject subject;

    Observer1 observer1(&subject);
    Observer2 observer2(&subject);

    subject.subscribe(&observer1);
    subject.subscribe(&observer2);

    bool flag = true;
    while(flag)
    {
        srand(time(0));
        int temp = rand();
        int price = temp % (100 - 10 + 1) + 10;
        if(price < 40)
        {
            subject.notify(price);
            flag = false;
        }
    }

    subject.unsubscribe(&observer1);
    subject.unsubscribe(&observer2);

    return 0;
}
```

```
Observer1: 현재 가격이 21이므로 구매합니다.
Observer2: 현재 가격이 21이므로 구매합니다.


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 18-1> - 상태패턴 예시코드

```
#include <iostream>
#include <windows.h>
```

```
class State;

class Light
{
private:
    State* state;
public:
    void setState(State* state);
    void deleteState();
    void turnSwitch();
};

class State
{
public:
    virtual void handle(Light* light) = 0;
};

class OffState : public State
{
public:
    void handle(Light* light);
};

class OnState : public State
{
public:
    void handle(Light* light);
};

void Light::setState(State* st)
{
    state = st;
}

void Light::deleteState()
{
    delete state;
}

void Light::turnSwitch()
{
    state->handle(this);
}

void OffState::handle(Light* light)
{
```

```
    std::cout << "불을 켭니다." << std::endl;
    light->deleteState();
    light->setState(new OnState);
}

void OnState::handle(Light* light)
{
    std::cout << "불을 끕니다." << std::endl;
    light->deleteState();
    light->setState(new OffState);
}

int main(void)
{
    Light* light = new Light;
    State* state = new OffState;
    light->setState(state);
    light->turnSwitch();
    light->turnSwitch();
    light->turnSwitch();

    delete state;
    delete light;
    return 0;
}
```

```
불을 켭니다.
불을 끕니다.
불을 켭니다.

...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 19-1> - 메멘토패턴 예시코드

```
#include <iostream>
```

```
class Memento;

class Originator
{
private:
    std::string state;
    Memento* memento;
public:
    Originator();
    ~Originator();
    std::string getState() const;
    void setState(std::string state);
    void restoreState();
};

class Memento
{
private:
    std::string state;
public:
    std::string getState() const;
    void setState(std::string state);
};

Originator::Originator()
{
    memento = new Memento;
}

Originator::~Originator()
{
    delete memento;
}

std::string Originator::getState() const
{
    return state;
}

void Originator::setState(std::string st)
{
    memento->setState(state);
    state = st;
}

void Originator::restoreState()
{
```

```
    state = memento->getState();
}

std::string Memento::getState() const
{
    return state;
}

void Memento::setState(std::string st)
{
    state = st;
}

int main(void)
{
    Originator originator;
    originator.setState("안녕하세요");
    std::cout << originator.getState() << std::endl;
    originator.setState("안녕");
    std::cout << originator.getState() << std::endl;
    originator.restoreState();
    std::cout << originator.getState() << std::endl;
    return 0;
}
```

```
안녕하세요
안녕
안녕하세요


...Program finished with exit code 0
Press ENTER to exit console.
```

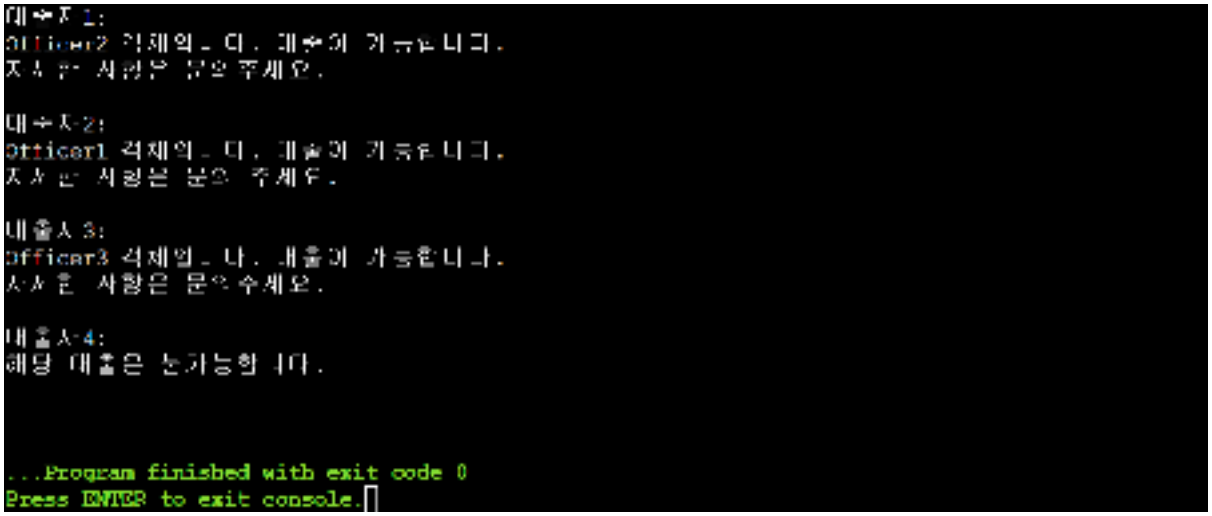
<예시 20-1> - 책임연쇄패턴 예시코드

```
#include <iostream>

class LoanOfficer
```

```
{
public:
    virtual void handle(int value) = 0;
    virtual ~LoanOfficer() { }
};

class LoanOfficer1 : public LoanOfficer
{
private:
    LoanOfficer* next;
public:
    LoanOfficer1();
    ~LoanOfficer1();
    void handle(int value);
};

class LoanOfficer2 : public LoanOfficer
{
private:
    LoanOfficer* next;
public:
    LoanOfficer2();
    ~LoanOfficer2();
    void handle(int value);
};

class LoanOfficer3 : public LoanOfficer
{
private:
    LoanOfficer* next;
public:
    LoanOfficer3();
    ~LoanOfficer3();
    void handle(int value);
};

class Loan
{
private:
    int loanValue;
    LoanOfficer* next;
public:
    Loan(int loanValue);
    ~Loan();
};

LoanOfficer1::LoanOfficer1()
{
```

```
    next = new LoanOfficer2();
}

LoanOfficer1::~LoanOfficer1()
{
    delete next;
}

void LoanOfficer1::handle(int loanValue)
{
    if(loanValue <= 1000)
    {
        std::cout << "Officer1 객체입니다. 대출이 가능합니다." << std::endl;
        std::cout << "자세한 사항은 문의 주세요." << std::endl;
    }
    else
    {
        next->handle(loanValue);
    }
}

LoanOfficer2::LoanOfficer2()
{
    next = new LoanOfficer3();
}

LoanOfficer2::~LoanOfficer2()
{
    delete next;
}

void LoanOfficer2::handle(int loanValue)
{
    if(loanValue <= 100000)
    {
        std::cout << "Officer2 객체입니다. 대출이 가능합니다." << std::endl;
        std::cout << "자세한 사항은 문의주세요." << std::endl << std::endl;
    }
    else
    {
        next->handle(loanValue);
    }
}
LoanOfficer3::LoanOfficer3()
{
    next = 0;
}
LoanOfficer3::~LoanOfficer3()
```

```
{
    delete next;
}

void LoanOfficer3::handle(int loanValue)
{
    if(loanValue > 100000 && loanValue <= 1000000)
    {
        std::cout << "Officer3 객체입니다. 대출이 가능합니다." << std::endl;
        std::cout << "자세한 사항은 문의주세요." << std::endl;
    }
    else
    {
        std::cout << "해당 대출은 불가능합니다." << std::endl;
    }
}

Loan::Loan(int value) : loanValue(value)
{
    next = new LoanOfficer1();
    next->handle(loanValue);
}

Loan::~Loan()
{
    delete next;
}

int main(void)
{
    std::cout << "대출자 1: " << std::endl;
    Loan loan1(82000);

    std::cout << "대출자 2: " << std::endl;
    Loan loan2(700);
    std::cout << std::endl;

    std::cout << "대출자 3: " << std::endl;
    Loan loan3(146000);
    std::cout << std::endl;

    std::cout << "대출자 4: " << std::endl;
    Loan loan4(3200000);
    std::cout << std::endl;

    return 0;
}
```

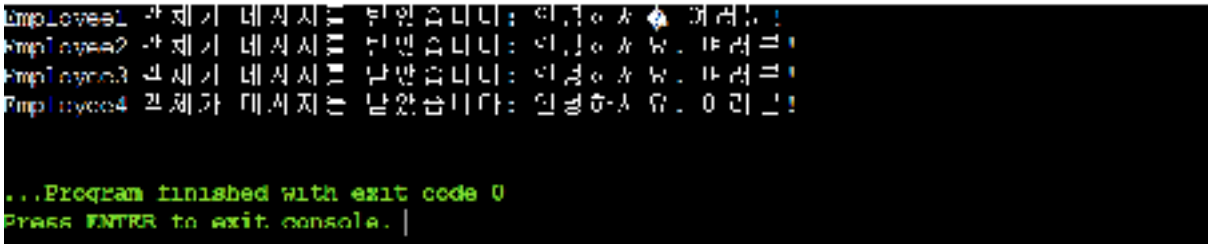


<예시 21-1> - 중재자패턴 예시코드

```
#include <iostream>
#include <vector>

class Employee
{
```

```
public:
    virtual void getMessage(std::string message) = 0;
};

class Employee1 : public Employee
{
public:
    void getMessage(std::string message);
};

class Employee2 : public Employee
{
public:
    void getMessage(std::string message);
};

class Employee3 : public Employee
{
public:
    void getMessage(std::string message);
};

class Employee4 : public Employee
{
public:
    void getMessage(std::string message);
};

class Mediator
{
private:
    std::vector<Employee*> employees;
public:
    Mediator();
    ~Mediator();
    void sendMessage(std::string message);
};

void Employee1::getMessage(std::string message)
{
    std::cout << "Employee1 객체가 메시지를 받았습니다: " << message <<
std::endl;
}

void Employee2::getMessage(std:
{
    std::cout << "Employee2 객체
std::endl;
```

```
}

void Employee3::getMessage(std::string message)
{
    std::cout << "Employee3 객체가 메시지를 받았습니다: " << message <<
std::endl;
}

void Employee4::getMessage(std::string message)
{
    std::cout << "Employee4 객체가 메시지를 받았습니다: " << message <<
std::endl;
}

Mediator::Mediator()
{
    employees.push_back(new Employee1());
    employees.push_back(new Employee2());
    employees.push_back(new Employee3());
    employees.push_back(new Employee4());
}

Mediator::~Mediator()
{
    for(int i = 0; i < employees.size(); i++)
    {
        delete employees[i];
    }
}

void Mediator::sendMessage(std::string message)
{
    for(int i = 0; i < employees.size(); i++)
    {
        employees[i]->getMessage(message);
    }
}

int main(void)
{
    Mediator mediator;
    mediator.sendMessage("안녕하세요 여러분!");
    return 0;
}
```



<예시 22-1> - 방문자패턴 예시코드

```
#include <iostream>

//-----------Visitor-----------
```

```
class Visitor
{
public:
    virtual void visit() = 0;
};
//------------------------------
//---------greeting-------------
class Greeting
{
public:
    void accept(Visitor* v)
    {
        v->visit();
    }
};
//--------------------------------

class EnglishVisitor : public Visitor
{
public:
    void visit()
    {
        std::cout << "Good Morning." << std::endl;
    }
};

class FrenchVisitor : public Visitor
{
public:
    void visit()
    {
        std::cout << "Bon Jour" << std::endl;
    }
};

class SpanishVisitor : public Visitor
{
public:
    void visit()
    {
        std::cout << "Buenos Dias." << std::endl;
    }
};

int main(void)
{
    Greeting greeting;
```

```
    Visitor* visitor1 = new EnglishVisitor;
    greeting.accept(visitor1);

    Visitor* visitor2 = new FrenchVisitor;
    greeting.accept(visitor2);

    Visitor* visitor3 = new SpanishVisitor;
    greeting.accept(visitor3);


    return 0;
}
```

```
Good Morning.
Bon Jour
Buenos Dias.


...Program finished with exit code 0
Press ENTER to exit console.
```

<예시 23-1> - 인터프리터패턴 예시코드

```
#include <iostream>
#include <string>
#include <algorithm>
class Expression
```

```
{
public:
    virtual bool interpret(const std::string& context) const = 0;
};
class HelloExpression : public Expression
{
public:
    bool interpret(const std::string& context) const override {
        return context.find("hello") != std::string::npos;
    }
};
class Context
{
private:
    std::string text;
public:
    Context(const std::string& t) : text(t) {}

    std::string getText() const
    {
        return text;
    }
};

int main()
{
    Context context("hello_world");
    Expression* expression = new HelloExpression();
    bool result = expression->interpret(context.getText());
    std::cout << "대상 문자열: " << context.getText() << std::endl;
    std::cout << "결과: " << (result ? "hello is here!!" : "No hello.") <<
std::endl;
    delete expression;
    return 0;
}
```

```
대상 문자열: hello_world
결과: hello is here!!


...Program finished with exit code 0
Press ENTER to exit console.
```