**1. 서론**

**1.1 디자인 패턴이란?**

- 객체지향 소프트웨어 설계에서 얻은 세세한 경험들을 문서화하고 카탈로그화 한 것을 말한다.

**1.2 패턴의 요소**

**1) 패턴 이름 ( Name )**

- 한두 단어로 설계 문제와 해법을 서술한 것.

- 좋은 이름을 통해 설계의 의도를 명확하게 나타내어 의사소통의 효율성을 높인다.

**2) 문제 ( Problem )**

- 언제 패턴을 사용해야 하는지에 대한 서술을 한다.

- 해결할 문제와 그 배경을 설명해야 한다.

**3) 해법 ( Solution )**

- 설계를 구성하는 요소들과 그 요소들 간의 관계, 책임 그리고 협력 관계를 서술한다.

- 문제에 대한 추상적인 설명을 제공하고 문제를 해결하기 위해서 클래스나 객체들의 나열 방법을 제공한다.

**4) 결과 ( Consequences )**

- 디자인 패턴을 적용해서 얻은 결과와 장단점을 서술한다.

- 패턴의 결과로는 처리 시간, 저장 공간, 사용 언어의 영향 등을 작성한다.

- 또한 패턴은 시스템의 유연성, 확장성, 이식성 등에도 영향을 준다.

- 이러한 결과들을 통하여, 차후 패턴들을 재 이해 및 평가하는데 도움을 받을 수 있다.

**1.3 디자인 패턴 카탈로그**

**1) 추상 펙토리 ( Abstract Factory )**

- 구체적인 클래스를 지정하지 않고, 관련성을 갖는 객체들의 집합을

생성하거나 서로 독립적인 객체들의 집합을 생성할 수 있는 인터페이스를 제공한다.

**2) 어댑타 ( Adapter )**

- 클래스의 인터페이스를 클라이언트가 기대하는 다른 인터페이스로 변환한다.

- Adapter 패턴은 호환성이 없는 인터페이스 때문에 함께 사용할 수 없는 클래스를

개조하여 함께 작동하도록 해준다.

**3) 브릿지 ( Bridge )**

- 추상화와 구현을 분리하여 각각을 독립적으로 변형할 수 있게 해준다.

**4) 빌더 ( Builder )**

- 복합 객체의 생성 과정과 표현 방법을 분리함으로써,

동일한 생성 공정이 서로 다른 표현을 만들 수 있게 한다.

**5) 책임 결합 ( Chain of Responsibility )**

- 요청을 처리할 수 있는 기회를 하나 이상의 객체에게 부여함으로써 요청하는 객체와 처리하는 객체 사이의 결합도를 없애려는 것이다.

…..

**1.4 카탈로그 조직화하기**

- 23개가 넘는 디자인 패턴의 수를 2가지 기준으로 분류하였다.

**1) 목적**

**(1) 생성 패턴**

- 객체의 생성 과정에 관여한다.

**(2) 구조 패턴**

- 클래스나 객체의 합성에 관여한다.

**(3) 행위 패턴**

- 클래스나 객체들이 상호작용하는 방법과 책임을 분산하는 방법을 정의한다.

**2) 범위**

- 패턴을 주로 클래스에 적용하는지 아니면 객체에 적용하는지를 구분하는 것이다.

**(1) 클래스 패턴**

- 클래스들과 서브클래스들 간의 관련성을 다루는 패턴이다.

- 관련성은 주로 상속이며 컴파일 시점에 정적으로 결정된다.

**(2) 객체 패턴**

- 객체 관련성을 다루는 것으로 런타임 시에 동적으로 변경할 수 있다.

- 대부분의 패턴들은 객체 영역에 속한다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 목적 | | |
| 생성 | 구조 | 행위 |
| 범위 | 클래스 | Factory Method | Adapter | Interpreter |
| Template method |
| 객체 | Abstrac Factory | Adapter | Chain of Responsibility |
| Command |
| Builder | Bridge | Iterator |
| Mediator |
| Prototype | Composite | Memento |
| Flyweight |
| Singleton | Decorator | Observer |
| State |
| Façade | Strategy |
| Proxy | Visitor |

**3) 목적 + 범위**

**(1) 생성 클래스 패턴**

- 객체를 생성하는 책임의 일부를 서브클래스가 담당하도록 한다.

**(2) 생성 객체 패턴**

- 객체를 생성하는 책임의 일부를 다른 객체에게 위임한다.

**(3) 구조 클래스 패턴**

- 상속을 이용해서 클래스를 합성한다.

**(4) 구조 객체 패턴**

- 객체를 합성하는 방법을 정의하고 있다.

**(5) 행위 클래스 패턴**

- 상속을 이용해서 알고리즘과 제어 흐름을 기술한다.

**(6) 행위 객체 패턴**

- 하나의 작업을 수행하기 위해 객체 집합이 어떻게 협력하는지를 기술한다.

**# 패턴을 조직하는 다른 방법**

- 일부 패턴은 함께 사용해야 하는 경우가 있다.

Ex) Composite 패턴 + Iterator 패턴 + Visitor 패턴

- 어떤 패턴은 다른 패턴의 대안이 될 수도 있다.

Ex) Prototype 패턴 ⬄ Abstract Factory 패턴

- 의도는 서로 다르지만, 결과적으로 유사한 설계 구조를 만드는 패턴도 있다.

Ex) Composite 패턴 ⬄ Decorator 패턴

**1.5 디자인 패턴을 이용하여 문제를 푸는 법**

**1) 적당한 객체 찾기**

- 객체 지향 프로그램은 ‘객체’로 이루어진다.

- ‘**객체**’는 데이터와 이 데이터를 처리하는 프로시저를 함께 묶은 단위이다.

- ‘**프로시저**’를 일반적으로 ‘**메소드’ or ‘오프레이션**’이라 부른다.

- ‘**객체**’는 ‘**요청**’ 또는 ‘**메시지**’를 클라이언트로부터 받으면 ‘**메소드**’을 수행한다.

- ‘**요청**’은 객체로 하여금 ‘**메소드**’를 실행하게 하는 유일한 방법이다.

- ‘**메소드**’는 객체의 내부 데이터의 상태를 변경하는 유일한 방법이다.

- 이러한 접근의 제약 사항으로 객체의 내부 상태는 ‘**캡슐화**’되어 있다 표현한다.

- 객체 지향 설계의 가장 어려운 부분은 시스템을 구성할 객체의 분할을 결정하는 것이다.

- ‘**캡슐화**’, ‘**크기** **정하기**’, ‘**종속성**’, ‘**유연성**’, ‘**성능**’, ‘**진화**’, ‘**재사용성**’ 등을 고려해야 한다.

- 각종 디자인 패턴의 특성을 파악하여, 적절한 적용을 수행해야 한다.

**2) 객체의 크기에 대한 결정**

- 객체는 크기와 수에서 매우 다양함을 보일 때도 있다.

Ex) Façade 패턴 : 서브시스템을 어떻게 객체로 표현할 수 있는지 설명.

Flyweight 패턴 : 규모는 작지만 개수는 많은 객체를 다루는 방법을 설명

Abstract Factory 패턴 & Builder 패턴 : 다른 객체를 생성하는 책임만 갖는 객체를 만들어 낸다.

Visitor 패턴 & Command 패턴

: 요청을 자신이 처리하는 것이 아니라,

다른 객체들이나 다른 객체 집합이 요청을 처리하여 구현하도록 책임지는 객체를 만들어 낸다.

**3) 객체 인터페이스 명세**

- 객체가 선언하는 모든 메소드는 메소드 이름, 파라미터로 받는 객체들, 메소드의 반환 값을 갖는다.

- 이러한 것들을 ‘**오퍼레이션 시그니처(Signature)**’라 한다.

- ‘**인터페이스**’란 객체가 정의하는 메소드의 모든 시그니처들을 일컫는다.

- ‘**객체의 인터페이스**’는 객체가 받아서 처리할 수 있는 메소드들의 집합이다.

- ‘**객체 인터페이스**’에 정의된 시그니처와 일치하는 어떤 요청이 객체에 전달되면,

객체는 메소드를 수행함으로써 그 요청을 처리한다.

- ‘**타입**’은 특정 인터페이스를 상징할 때 사용하는 용어입니다.

Ex) 객체가 ‘**Window 타입**’을 갖는다는 것은, 객체가 ‘**Window 인터페이스**’에 정의된

메소드들을 모두 처리할 수 있다는 것을 의미합니다.

- ‘**객체**’는 여러 타입을 가질 수 있고, 서로 다른 객체가 하나의 타입을 공유할 수도 있다.

- ‘**인터페이스**’가 다른 ‘**인터페이스**’를 부분집합으로 포함하는 경우가 있다.

- 이때, 포함하는 ‘**인터페이스**’를 ‘**서브타입**’이라 하고 포함당하는 ‘**인터페이스**’를 ‘**슈퍼타입**’이라 한다.

- ‘**서브타입**’은 ‘**슈퍼타입**’에 정의된 메소드를 포함하게 됩니다.

- ‘**객체**’는 인터페이스를 통해 자신을 드러냅니다.

- 외부에서는 ‘**객체**’에 대해 알 수 있는 것은 ‘**인터페이스**’ 밖에 없다.

- 따라서, ‘**요청**’은 ‘**객체의** **인터페이스**’를 통해서만 이루어질 수 있다.

- 객체의 ‘**인터페이스**’는 구현에 대해서는 전혀 알려주지 않는다.

-> 서로 다른 객체는 ‘**인터페이스**’에 정의한 요청의 구현 방법을 자유롭게 선택할 수 있다.

-> 동일한 ‘**인터페이스**’를 갖는 두 객체가 완전히 다른 구현을 갖을 수 있게 한다.

-> 동일한 요청이라도 처리하는 객체들이 서로 다른 객체라면 다른 결과가 나올 수 있게 한다.

-> 이와 같이, 요청과 요청을 처리할 객체를 런타임 시에 결정하는 기법을 ‘**동적 바인딩**’이라 한다.

- ‘**동적** **바인딩**’은 요청이 어떻게 구현되어 어떤 결과를 만들어 낼지를,

런타임에 결정할 수 있음을 의미한다.

- 결과적으로 프로그램을 작성할 때는 객체가 어떤 특정 ‘**인터페이스**’를 갖도록 작성한다.

- 즉, 이 객체는 요청을 처리할 정확한 ‘**인터페이스**’를 갖게 된다.

- ‘**동적** **바인딩**’은 프로그램이 기대하는 객체를 동일한 인터페이스를 갖는

다른 객체로 대체하게 한다.

- 이를 ‘**다형성**’이라 하며, 런타임 시에 객체와 새로운 관련성을 수립되는 것을 의미한다.

- ‘**디자인** **패턴**’은 ‘**인터페이스**’에 정의해야 하는 중요 요소가 무엇이고

어떤 종류의 데이터를 주고 받아야 하는지 식별하여 ‘**인터페이스**’를 정의하도록 도와준다.

- **가끔** ‘**디자인** **패턴**’은 ‘**인터페이스**’에 넣지 말아햐 할 것을 알려 주기도 한다.

- ‘**디자인** **패턴**’은 ‘**인터페이스**’들 간의 관령성도 정의한다.

**4) 객체 구현 명세하기**

- ‘**객체의** **구현**’은 ‘**클래스**’에서 정의한다.

- ‘**클래스**’는 ‘**객체의** **내부** **데이터**’와 ‘**표현** **방법**’과 ‘**메소드**’를 정의하여 객체를 구현한다.

- ‘**클래스**’로부터 ‘**인스턴스**’ 생성을 통해 ‘**객체**’가 만들어진다.

- 즉, ‘**객체**’는 ‘**클래스의** **인스턴스**’이다.

- ‘**클래스의** **인스턴스화**’는 ‘**객체**’의 내부 데이터의 공간 할당, 데이터를 메소드와 관련짓는 작업이다.

- ‘**추상** **클래스**’는 모든 서브 클래스들 사이의 공통되는 ‘**인터페이스**’를 정의한다.

- ‘**추상** **클래스**’는 정의한 모든 메소드나 일부 메소드 구현을 서브클래스로 떠넘긴다.

- 정의한 메소드들이 구현된 것이 아니기에, ‘**인스턴스화**’가 불가능하다.

- ‘**추상** **클래스**’가 아닌 클래스를 ‘**구체적** **클래스**(**Concrete** **class**)’라 한다.

- ‘오버라이딩’을 통해 서브클래스는 부모 클래스에 정의된 처리 방식을 변경할 수 있다.

-

- ‘클래스 인스턴스화’ : 점선 + 검은 화살표

- ‘클래스 상속’ : 실선 + 하얀 화살표

- ‘추상 클래스’ : 이탤릭체로 작성 ( 추상 메소드도 이탤릭체로 작성 )

- ‘mixin 클래스’ : 다중 상속으로 사용.

- ‘주석 다이어그램’ : ‘점선’ + 모서리 접힌 노트 기호

**5) 클래스 상속 대 인터페이스 상속**

- ‘**클래스**’와 ‘**타입**’ 간의 차이를 이해하는 것이 중요하다.

- ‘**클래스의** **상속**’은 객체의 구현을 정의할 때 이미 정의된 객체의 구현을 바탕으로 한다.

- 즉, 코드 공유의 방법이다.

- ‘**인터페이스** **상속**’은 객체가 다른 곳에서 사용될 수 있음을 의미한다.

- 즉, 구현 부분이 없이, 메소드 이름, 파라미터, 리턴 값만 정의된 것을 상속한다.

- C++에서의 상속은 ‘**인터페이스**’와 ‘**구현**’ 상속 모두를 의미한다.

- 가상 함수를 갖는 클래스를 public으로 상속하는 것을 뜻한다.

- C++ ‘**인터페이스의** **상속**’은 순수 가상 함수를 정의한 추상 클래스를 public으로 상속받고,

구현부분을 ‘private’로 접근제어하는 것이다.

- ‘**구체적인** **클래스**’를 정의하고 요청을 보내는 것 보다,

‘**추상** **클래스**’의 객체에게 ‘**메시지**’를 보내도록 프로그래밍하는 것이 매우 좋다.

- 이렇게 하면, 런타임 시에 ‘**구체적** **클래스**’의 인스턴스로 바꿀 수 있다.

- 즉, 추상 클래스를 상속한다는 것은 단순한 코드의 재사용을 위한 상속이 아니라

추상 클래스가 정의하고 있는인터페이스를 상속하겠다는 의미이다.

**6) ‘구현’이 아닌, ‘인터페이스’에 따라서 프로그래밍 하자**

- ‘**클래스** **상속**’은 ‘**구현의** **재사용**’이라는 특징만 있는 것이 아니다.

- ‘**동일한** **인터페이스**’를 갖는 객체군을 정의하는 것 또한 매우 중요한 특징이다.

- 객체군을 정의함으로써, ‘**다형성**’을 얻어 낼 수 있기 때문이다.

- ‘**상속**’을 적절하게 이용하면, 모든 클래스는 ‘**추상** **클래스**’를 상속하여 ‘**인터페이스**’를 공유하게된다.

- 이것은, 모든 ‘**서브클래스**’가 ‘**부모** **클래스**’의 메소드를 추가 및 재정의하여 구현할 수 있다.

- 즉, 모든 ‘**서브** **클래스**’가 ‘**부모** **클래스**’의 ‘**서브** **타입**’이 되는 것이다. 이에 대한 이점은 다음과 같다.

a. 클라이언트가 원하는 인터페이스를 객체가 만족하고 있는 한

클라이언트는 그들이 사용하는 특정 객체타입에 대해 알아야 할 필요는 없다.

b. 클라이언트는 이들 객체를 구현하는 클래스에 대해 알 필요가 없고,

단지 인터페이스를 정의하는 추상 클래스가 무엇인지만 알고 있을 뿐이다.

- 이는, 서브시스템 간의 구현 종속성을 없애고, 재사용 가능한 객체지향의 원칙을 만족하게 한다.

- 그러므로, 변수를 구체적 클래스의 인스턴스로 선언하지 말고,

추상 클래스의 인터페이스를 만족하도록 인스턴스 변수를 정의하도록 하자.

**7) 재사용을 실현 가능한 것으로**

**(1) 상속 대 합성**

- ‘**객체지향** **시스템**’에서 ‘**기능의** **재사용**’을 위한 대표적인 기법은 ‘**상속**’과 ‘**객체** **합성**’이다.

- ‘**클래스** **상속**’은 다른 클래스를 이용해서 한 클래스의 구현을 정의하는 것이다.

- 이와 같은, ‘**서브클래싱**’에 의한 재사용을 ‘**화이트박스** **재사용**’이라고 한다.

-> 상속을 받으면, 부모 클래스의 내부가 서브클래스에 공개되어 이런 의미가 붙었다.

- ‘**장점**’으로는, 컴파일 시점에 정적으로 정의되고 프로그래밍 언어가 직업 지원하므로

그대로 구현하면 된다.

- ‘**부모** **클래스**’의 구현을 쉽게 재정의 할 수도 있고, 일부분만 재정의 해도 된다.

- ‘**단점**’으로는, 런타임 시에 상속받은 부모 클래스의 구현을 변경할 수 없다는 것이다.

-> 상속은 컴파일 시점에 결정되는 사항이기 때문이다.

- ‘**클래스** **상속**’은 코드를 공유하는 것과 비슷하므로, ‘**캡슐화를** **파괴**’하기도 한다.

-> 그래서 서브클래스는 부모 클래스의 ‘**구현에** **종속**’되고, 부모 클래스 구현에 변경이 생기면

서브 클래스도 변경해야 한다.

-> 또한, ‘**구현의** **종속성**’ 때문에 ‘**서브클래스를** **재사용**’하려고 할 때 문제가 발생한다.

-> ‘**상속한** **구현**’이 새로운 문제에 맞지 않을 때, 부모 클래스를 재작성해야 하거나

다른 것으로 대체하는 일이 생기게 된다. ( 유연성, 재사용성 감소 )

- 이를 해결하기 위해서는, ‘**인터페이스** **상속**’을 하는 것이다.

-> ‘**인터페이스** **상속**’은 ‘**구현**’을 서브클래스에서 정의하기에, ‘**구현의** **종속성**’은 해결된다.

-> 이후, ‘**서브클래스의** **재사용**’은 문제 없이 진행될 수 있다.

- ‘**객체** **합성은**’ ‘**클래스** **상속**’에 대한 대안이다.

- 새로운 기능성을 위해서 객체들을 합성하는 것이다.

- ‘**객체**’를 ‘**합성**’하려면, 합성할 객체들의 ‘**인터페이스**’를 명확하게 정의해야만 한다.

- 이를, ‘**블랙박스** **재상용**’이라 한다.

-> 객체의 내부는 공개되지 않고 ‘**인터페이스**’만을 통해 재사용되기 때문이다.

(2) 위임

(3) 상속 대 파라미터화된 타입

**1.3 디자인 패턴 기술하기**

- 해당 책이 디자인 패턴의 설계를 재사용하기 위해서, 설계를 하기까지의

다양한 결정, 대안, 장단점 등을 고려하여 기술한다.

1) 패턴 이름과 분류 ( Pattern Name and Classification )

- 패턴의 이름은 핵심을 간결하게 전해준다.

2) 의도 ( Intent )

- ‘이 디자인 패턴은 무엇을 하는 가?’, ‘의도와 논리적인 근거는 무엇인가?’,

‘어떤 특정한 문제나 이슈를 해결하기 위한 것이가?’를 서술한다.

3) 다른 이름 ( Also Known As )

- 패턴을 다른 이름으로 정의할 수 있다면, 그것은 무엇일까?

4) 동기 ( Motivation )

- 설계 문제를 제시, 패턴 안에서 클래스나 객체 구조가 어떻게 문제를 해결하는지 설명해준다.

- 패턴에 대한 좀 더 추상화된 설명을 이해할 수 있게 도와준다.

5) 활용성

-