Design Patterns 요약

1.1 디자인패턴 이란?

- 패턴이란 특정 컨텍스트 내에서 주어진 문제에 대한 해결책이다.
- 어떤 컨텍스트 내에서 일련의 제약조건에 의해 영향을 받을 수 있는 문제에 봉착했다면, 그 제약조건 내에서 목적을 달성하기 위한 해결책을 찾아낼 수 있는 디자인을 적용하면 된다.

주의점 및 추가 사항

- 디자인 패턴의 과다한 사용은 불필요하게 복잡한 코드를 초래할 수 있다.
  항상 가장 간단한 해결책으로 목적을 달성할 수 있도록 하고, 반드시 필요할 때만 디자인 패턴을 적용하자.
- 코딩할 때 어떤 패턴을 사용하고 있는지 주석으로 적어주자.
  클래스와 메서드 이름을 만들 때도 사용 중인 패턴이 분명하게 드러날 수 있도록 해보자.
  다른 개발자들이 그 코드를 볼 때 무엇을 어떻게 구현했는지 훨씬 빠르게 이해할 수 있다.

1.2 디자인패턴 설계 원칙

- 코드에서 달라지는 부분을 찾아내고, 달라지지 않는 부분으로부터 분리.
- 기능구현이 아닌 인터페이스에 맞춰서 프로그래밍한다.
- 상속보다는 implements를 활용한다.
- 상호작용 하는 객체 사이에서는 가능한 느슨한 결합으로 설계한다
- 클래스는 확장에 대해서는 열려 있어야 하지만
  코드 변경에 대해서는 닫혀 있어야 한다. (OCP : Open-Closed Principle)
- 추상화된 것에 의존하도록 만들어라.
- 최소 지식 원칙 - 아래의 경우에 해당되는 것만 호출하도록 디자인
  1. 객체 자체에 있는 메서드
  2. 메서드에 매개변수로 전달된 객체
  3. 메서드에서 생성하거나 인스턴스를 만든 객체
  4. 그 객체에 속하는 멤버
- 클래스를 바꾸는 이유는 한 가지 문제여야 한다.
- 차후 변경될 여지에 대해 고민한다

2.1 디자인패턴 종류

- **Creational Patterns (생성)**
  객체 인스턴스 생성을 위한 패턴으로, 인스턴스를 생성하는 객체와 생성된 인스턴스사이의 연결(의존성)을 해제해 준다

  Singleton | Factory Method | Abstract Factory | Prototype | Builder

- **Structural Patterns (구조)**
  클래스 및 객체들을 구성을 통해서 더 큰 구조로 만들 수 있게 해준다

  Adapter | Bridge | Composite | Decorator | Facade | Flyweight | Proxy

- **Behavioral Patterns (행동)**
  클래스와 객체들이 상호작용하는 방법 및 역할을 분담하는 방법을 정의

  Command | Iterator | Mediator | Memento | Observer | Interpreter | State | Strategy | Visitor | Template Method | Chain of Responsibility

2.2 디자인패턴 구분

- Class Patterns

클래스 사이의 관계가 상속을 통해서 어떤 식으로 정의되는지를 다룬다. 클래스 패턴은 컴파일시에 관계가 결정된다

Factory Method | Adapter | Template Method | Interpreter

- Object(Instance) Patterns

객체사이의 관계를 다루며, 객체사이의 관계가 구성을 통해 결정된다. 객체 패턴은 실행중에 관계가 만들어지기 때문에 유연하게 적용할 수 있다

Singleton | Builder | Abstract Factory | Prototype | Facade

Flyweight | Proxy | Bridge | Composite | Decorator

Command | Iterator | Mediator | Memento | Observer

Chain of Responsibility | State | Strategy | Visitor

3. Creational Patterns : 생성

1. Abstract Factory(추상 팩토리)
- 서로 관련성 있거나 독립적인 여러 객체의 집합을 생성할 수 있는 인터페이스 제공하는 패턴

2. Builder(빌더)
- 복잡한 객체를 생성하는 방법과 표현하는 방법을 정의하는 클래스를 분리하여, 서로 다른 표현 결과를 만들 수 있게 하는 패턴

3. Factory Method(팩토리 메소드)
- 객체를 생성하는 인터페이스는 미리 정의하고, 인스턴스를 만들 클래스의 결정은 서브클래스 쪽에서 내리는 패턴

4. Protytype(원형)
- 생성할 객체의 종류를 명세하는 데에 원형이 되는 예시물을 이용하고, 그 원형을 복사함으로써 새로운 객체를 생성하는 패턴

5. Singleton(단일체)
- 클래스의 인스턴스가 하나만 생성이 되게 보장하며, 전역으로 설정이 되어 어디서든 접근이 가능하도록 제공하는 패턴

4. Structural Pattern : 구조

1. Adapter(적응자)
- 서로 다른 클래스끼리 함께 동작할 수 있도록 인터페이스를 변환하는 패턴

2. Bridge(가교)
- 구현에서 추상을 분리하여, 이들을 독립적으로 다양성을 가질 수 있도록 하는 패턴

3. Composite(복합체)
- 각각의 단일 객체와 복합체를 한 종류의 클래스로 설계하여 사용자는 단일 객체든 복합체든 상관 없이 똑같이 다룰 수 있는 편리함을 부여하는 패턴

4. Decorator(작성자)
- 클래스에게 동적인 기능이나 임무를 추가하는 패턴

5. Facade(퍼사드)
- 복잡한 서브 시스템에 대해서 간단한 인터페이스를 제공하는 패턴

6. Flyweight(플라이웨이트)
- 크기가 작은 다수의 객체가 있을 때, 모두가 갖는 본질적인 요소를 클래스화하여 공유하므로써 메모리 절감의 효과를 보는 패턴

7. Proxy(프록시)
- 실제 객체로 접근을 통제 및 정보은닉을 하기 위해 대리 객체를 제공하는 패턴

5.1. Behavioral Pattern 1 : 행동

1. Chain of Responsibility (책임 연쇄)
- 각종 요청에 대해서 그것을 처리할 수 있는 객체가 존재할 때가지 연속적으로 객체를 탐사하며 요청을 처리할 수 있는 객체를 찾아주는 패턴
2. Command(명령)
- 요청을 객체의 형태로 캡슐화하여 서로 요청이 다른 사용자의 명령을 지원하게 만드는 패턴
3. Interpreter(해석자)
- 언어의 다양한 해석, 구체적으로 구문을 나누고, 그 분리된 구문의 해석을 맡는 클래스를 각각 작성하여, 여러 형태의 언어 구문을 해석할 수 있게 만드는 패턴
4. Iterator(반복자)
- 컨테이너의 반복에 있어서 일관된 인터페이스를 두어 순차적으로 접근할 수 있는 방법을 제공하는 패턴
5. Mediator(중재자)
- 한 집합에 속한 객체들의 상호작용을 캡슐화하는 객체를 정의하는 패턴
6. Memento(메멘토)
- 객체의 상태를 관리하여 해당 객체가 이전 상태로 돌아갈 수 있도록 하는 패턴

5.2. Behavioral Pattern 2 : 행동

7. Observer(감시자)
- 데이터에 대한 모든 감시자들의 갱신을 자동으로 할 수 있게 하는 패턴

8. State(상태)
- 상태를 일반적인 데이터 변수로 두지않고, 객체로 만들어 그 상태에 따른 행동을 변경할 수 있도록 하는 패턴

9. Strategy(전략)
- 비슷한 객체들을 캡슐화하고, 교환이 가능하도록 만든 패턴

10. Template Method(템플릿 메소드)
- 템플릿을 만들고, 서브클래스에서 구체적으로 처리할 수 있도록 하는패턴

11. Visitor(방문자)
- 데이터(또는 객체)의 구조와 처리(기능)을 분리할 수있는 패턴
  구조 안을 돌아다니면서 일을 한다

6. 요약

생성
1. Abstract factory : 인스턴스화 할 패토리에 대한 결정을 중앙 집중화 함
2. Factory method : 여러 개의 구현 중에 특정 타입의 객체를 선택 생성하는 작업을 중앙 집중화함
3. Builder : 생성 절차는 동일하나 서로 다른 결과를 가져오는 경우, 복잡한 객체의 구축을 표현(represention)과 분리함
4. Lazy initialization : 객체 생성, 값 계산, 다른 값 비싼 프로세스를 실제로 필요한 시점까지 지연하는 전술
5. Object pool : 더이상 사용하지 않는 객체의 재활용을 통해 값 비싼 획득 및 해지를 피함
6. Prototype pattern : 어떤 애플리케이션에서 표준적인 객체 생성이 값비쌀 때 사용함
7. Singleton pattern : 클래스의 객체 생성을 하나로 제한함

구조
1. Adapter : 어떤 클래스를 위해, 클라이언트가 기대하는 하나의 인터페이스를 채택함
2. Bridge : 추상적인 부분을 구현과 분리함으로써 두 요소가 독립적으로 확장 가능하게 함
3. Composite : 각 객체가 동일한 인터페이스를 가지는 객체들의 트리구조
4. Decorator : 상속을 사용할 경우 클래스 갯수가 지수적으로 늘어날 경우, 클래스에 기능 추가
5. Facade : 기존의 복잡한 인터페이스들을 사용하기 쉽도록 단순화한 인터페이스 생성
6. Flyweight : 공간을 절약하기 위해 공통의 속성을 공유하는 많은 수의 객체
7. Proxy : 다른 것들을 위한 인터페이스 역할을 하는 클래스

행위
1. Chain of responsibility : 명령 객체가 로직을 포함하고 처리하는 객체에 의해 다른 객체로 전달되거나 처리됨
2. Command : 행위와 매개변수를 포함하고 있는 명령 객체
3. Interpreter : 특정 문제 집합을 신속하게 풀기 위해 특수화된 컴퓨터 언어를 구현함
4. Iterator : 객체 내부 표현방식을 노출하지 않고 집합 객체의 요소에 순차적으로 접근할 때 사용함
5. Mediator : 하위 시스템 안에서 인터페이스의 집합을 위하여 통합된 인터페이스를 제공
6. Memento : 객체를 이전읫 ㅏㅇ태로 되돌릴 수 있는 역량을 제공함 (Rollback)
7. Null Object : 객체의 기본값(default value)으로 작동하도록 설계함
8. Observer : 별칭 : Puvlish/Subscribe 또는 이벤트 리스너; 다른 객체에 의해서 발생할 이벤트를 관찰하기 위해 객체를 등록
9. State : 실행 시점에 알고리즘을 선택할 수 있도록 함
10.Specification : 부울리언 방식으로 재결합할 수 있는 비즈니스 로직
11.Template method : 프로그램의 뼈대를 기술함으로써 처리 절차를 공유함
12.Visitor : 객체로부터 알고리즘을 분리하는 방법

