SWIFT
Design Pattern
Introduction

Bill Kim(김정훈) | ibillkim@gmail.com

목차

Design Pattern?

디자인 패턴이란 용어와 방법론은 사실 건축학 영역에서 고안이 되어 그 시초를 바탕으로 하여 여러 다양한 분야에서도 많이 도입되고 있습니다.

건축학에서의 디자인 패턴은 건축가 크리스토퍼 알렉산더가 제안한 건축의 기법을 틀로 고안한 아이디어였습니다.

이 디자인 패턴을 이용하는 방식은 건물 또는 도시의 설계를, 이전에 벌어졌던 몇가지의 단편적인 설계 결과들을 디자인 패턴으로 두고, 이를 묶어 조합하는 형태를 띄게 된다.

이들 패턴의 수집을, 각각의 패턴을 '패턴 언어'를 통해 기술하여 정리한 것이 크리스토퍼 알렉산더의 업적이라 할 수 있습니다.

Design Pattern?

소프트웨어에서의 디자인 패턴 또한 건축학에서 사용하는 디자인 패턴과 그 맥락과 목적은 같습니다.

소프트웨어 개발 방법에서 사용하는 디자인 패턴은 기존에 잘 설계된 코드나 큰 틀들을 잘 접목하여 결국 효율적이며 좋은 코드들을 재사용하기 용이하도록 도와주기 위한 방법론입니다.

Purpose

그렇다면 사실 디자인 패턴은 왜 사용해야 할까요?

그 이유는 앞서 소프트웨어에서 사용하는 디자인 패턴의 정의에서 말했던 효율적이며 좋은 코드들의 재사용에서 찾을 수 있습니다.

효율적이며 좋은 코드란?

1. 코드가 명확하고 단순하여야 한다.
2. 각 모듈 및 객체는 최소한의 기능만을 하도록 세분화한다.
3. 재사용성이 높아야 한다.
4. 유지보수가 적거나 쉬어야 한다.
5. 리소스의 낭비가 없거나 최소화하여야 한다.

Purpose

객체지향 방법론에서는 아래의 같은 SOLID 원칙을 두고 효율적이고 좋은 코드를 정의하고 있습니다.

SRP (Single Responsiblity Principle, 단일 책임 원칙)
: 객체는 둘 이상의 책임을 갖지 않게 한다.

OCP (Open-Closed Principle, 개방-폐쇄 원칙)
: 클래스 확장에 대해서는 열려 있으나 변경에 대해서는 닫혀 있다.

LSP (Liskov Substitution Principle, 리스코프 치환 원칙)
: 자식 타입은 부모 타입이 사용되는 곳에 대체 가능하며 자식 클래스는 부모의 책임을 넘지 말고 자식 기능 제약이 필요하다.

ISP (Interface Segregation Principle, 인터페이스 분리 원칙)
: 인터페이스를 구체적으로 분리한다.

DIP (Dependency Inversion Principle, 의존 역전 원칙)
: 구현 클래스에 의존하지 말고 추상화된 클래스에 의존한다.

Classification

소프트웨어에서의 디자인 패턴은 다양한 종류와 분류 방식이 있습니다.

그 중에서 가장 많이 활용하고 있는 디자인 패턴은 바로 GoF(Gang Of Fout) 디자인 패턴입니다.

Gof라고 불리는 사람들(에리히 감마(Erich Gamma), 리차드 헬름(Richard Helm), 랄프 존슨(Ralph Johnson), 존 블리시디스(John Vissides)에 의해서 소프트웨어에서의 디자인 패턴을 구체화하고 체계화가 되었다고 볼 수 있습니다.

GoF에서는 23가지의 디자인 패턴을 정리하고 각각의 디자인 패턴을 생성(Creational), 구조(Structural), 행위(Behavioral) 3가지로 분류했습니다.

Classification

아래의 표는 GoF에서 분류한 디자인 패턴의 종류입니다.

본 강의에서는 각 디자인 패턴들의 핵심적인 정의에 대해서만 살펴보도록 하겠습니다. 자세한 디자인 패턴의 내용은 개별 강좌를 통하여 살펴볼 예정입니다.

생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
• 추상 팩토리(Abstract Factory) • 빌더(Builder) • 팩토리 메서드(Factory Methos) • 프로토타입(Prototype) • 싱글턴(Singleton)	• 어댑터(Adapter) • 브리지(Bridge) • 컴퍼지트(Composite) • 데커레이터(Decorator) • 퍼사드(Facade) • 플라이웨이트(Flyweight) • 프록시(Proxy)	• 책임 연쇄(Chain of Responsibility) • 커맨드(Command) • 인터프리터(Interpreter) • 이터레이터(Iterator) • 미디에이터(Mediator) • 메멘토(Memento) • 옵서버(Observer) • 테이트(State) • 스트래티지(Strategy) • 템플릿 메서드(Template Method) • 비지터(Visitor)

출처 : https://gmlwjd9405.github.io/2018/07/06/design-pattern.html

Creation Pattern

- 생성(Creational) 패턴

객체의 생성과 관련된 패턴입니다.

객체의 생성과 조합을 캡슐화해 특정 객체가 생성되거나 변경되어도 프로그램 구조에 영향을 크게 받지 않도록 유연성을 제공합니다.

Creation Pattern

추상 팩토리(Abstract Factory) :
구제적인 클래스에 지정하지 않고 관련성이 있는 독립적인 객체들을 생성하기 위한 인터페이스를 제공하는 패턴, 구상 클래스를 지정하지 않으면서도 일군의 객체를 생성

빌더(Builder) :
복합 객체의 생성과정과 표현과정을 분리시켜 동일한 생성과정에서 다양한 표현을 생성할 수 있는 패턴, 제품을 여러개로 분류하고 생산 단계를 캡슐화

팩토리 메서드(Factory Method) :
객체를 생성하는 인터페이스를 정의하지만, 인스턴스를 만드는 클래스는 서브클래스에서 결정하도록 하는 패턴, 즉 생성할 구상 클래스를 서브클래스에서 결정한다.

프로토타입(Prototype) :
생성할 객체의 종류를 명시하는 데 원형이 되는 예시물을 이용하고 새로운 객체를 이 원형들을 복사함으로써 생성하는 패턴

싱글톤(Singleton) :
클래스의 인스턴스가 하나임을 보장하고 접근할 수 있는 전역적인 접근점을 제공하는 패턴

Structral Pattern

- 구조(Structural) 패턴

클래스나 객체를 조합해 더 큰 구조를 만드는 패턴

서로 다른 인터페이스를 지닌 2개의 객체를 묶어 단일 인터페이스를 제공하거나 객체들을 서로 묶어 새로운 기능을 제공하는 패턴입니다.

Structral Pattern

어댑터(Adapter) :
클래스의 인터페이스를 사용자가 기대하는 다른 인터페이스로 변환하는 패턴으로, 호환성이 없는 인터페이스 때문에 함께 동작할 수 없는 클래스들이 함께 작동하도록 해주는 패턴, 객체를 감싸서 다른 인터페이스를 제공

브릿지(Bridge) :
구현부에 추상층을 분리하여 각자 독립적으로 변형할 수 있도록 하는 패턴

컴포지트(Composite) :
객체들의 관계를 트리 구조로 구성하여 부분-전체 계층을 표현하는 패턴으로, 사용자가 단일 / 복합객체 모두 동일하게 다루도록 하는 패턴

데코레이터(Decorator) :
주어진 상황 및 용도에 따라 어떤 객체에 책임을 덧붙이는 패턴으로 서브클래스 사용을 하지 않고 기능 확장을 할때 용이, 객체를 감싸서 새로운 행동을 제공

Structral Pattern

퍼사드(Facade) :
서브시스템에 있는 인터페이스 집합에 통합된 하나의 인터페이스를 제공하는 패턴, 일련의 클래스에 대해서 간단한 인터페이스를 제공

플라이웨이트(Flyweight) :
객체 공유 시 이미 생성한 객체라면 공유하여 전달해주는 패턴

프록시(Proxy) :
어떤 다른 객체로 접근하는 것을 통제하기 위해 그 객체의 매니저 또는 대리자(Proxy)를 제공하는 패턴

Behavioral Pattern

- 행위(Behavioral) 패턴

객체나 클래스 사이의 알고리즘이나 책임 분배에 관련된 패턴

한 객체가 혼자 수행할 수 없는 작업을 여러 개의 객체로 어떻게 분배하는지, 또 그렇게 하면서도 객체 사이의 결합도를 최소화하는 것에 중점을 둡니다.

Behavioral Pattern

책임 연쇄(Chain of Responsibility) :
요청을 처리하는 기회를 하나 이상의 객체에 부여하여 요청을 보내는 쪽과 받는 쪽의 결합을 피하는 패턴요청을 받는 객체를 연쇄적으로 묶고 객체를 처리할 수 있을 때까지 요청을 전달합니다. 한 요청을 두 개 이상의 객체에서 처리하고 싶을 때 사용

커맨드(Command) :
실행될 기능을 캡슐화함으로써 주어진 여러 기능을 실행할 수 있는 재사용성이 높은 클래스를 설계하는 패턴, 즉 요청을 객체로 감싸서 사용

이터레이터(Iterator) :
내부 표현부를 노출하지 않고 어떤 객체 집합의 원소들을 순차적으로 접근할 수 있는 방법을 제공하는 패턴, 컬렉션이 어떤 식으로 구현되었는지 드러내진 않으면서도 컬렉션 내에 있는 모든 객체에 대해 반복 작업을 처리

미디에이터(Mediator) :
한 집합에 속해있는 객체들의 상호 작용을 캡슐화하는 객체를 정의하는 패턴, 서로 관련된 객체 사이의 복잡한 통신과 제어를 한 곳으로 집중시키고자 할 때 사용

Behavioral Pattern

메멘토(Mememto) :
객체의 상태를 저장하고 이전 상태로 복원할 수 있도록 도와주는 패턴

옵저버(Observer) :
객체들 사이에 1 : n 의 의존관계를 정의하여 어떤 객체의 상태가 변할 때, 의존관계에 있는 모든 객체들이 통지받고 자동으로 갱신될 수 있게 만드는 패턴, 상태가 변경되면 모든 구독자들에게 알림

상태(State) :
객체 내부의 상태에 따라서 객체가 다른 행동을 할 수 있도록 해주는 패턴, 알고리즘의 개별 단계를 구현하는 방법을 서브클래스에서 결정

전략(Strategy) :
동일 계열의 알고리즘들을 정의하고, 각각 캡슐화하며 이들을 상호교환 가능하도록 만들고 알고리즘을 사용하는 사용자로부터 독립적으로 알고리즘이 변경될 수 있도록 하는 패턴, 즉 교환 가능한 행동을 캡슐화하고 위임을 통해서 어떤 행동을 사용할지 결정

Behavioral Pattern

템플릿 메서드(Template Method) :
객체의 연산에서 알고리즘의 뼈대만 정의하고, 나머지는 서브클래스에서 이루어지게 하는 패턴, 즉 알고리즘의 개별 단계를 구현하는 방법을 서브클래스에서 결정

방문자(Visitor) :
객체의 구조와 기능을 분리하여 구조는 변하지 않으면서 기능을 추가하거나 확장할 수 있는 패턴, 다양한 객체에 새로운 기능을 추가해야 하는데 캡슐화가 별로 중요하지 않은 경우

인터프리터(Interpreter) :
주어진 언어에 대해서 문법을 위한 표현수단을 정의하고, 해당 언어로 된 문장을 해석하는 해석기를 사용하는 패턴

References

[1] 디자인 패턴의 정의와 종류 : https://jwprogramming.tistory.com/68

[2] [Design Pattern] 디자인 패턴 종류 : https://gmlwjd9405.github.io/2018/07/06/design-pattern.html

[3] 디자인 패턴 : https://ko.wikipedia.org/wiki/디자인_패턴

[4] [디자인 패턴 1편]. 디자인 패턴 개요 : https://dailyheumsi.tistory.com/148

[5] 디자인 패턴 한 방에 정리하기(feat. 혹시 디자인 패턴 아시는 것 있으세요?) : https://jeong-pro.tistory.com/98

References

[6] 디자인 패턴의 종류에 대해 알아보자 : https://hyeonstorage.tistory.com/99

[7] 디자인 패턴(Design Pattern) 이란 : https://gone-sw.tistory.com/4

[8] 디자인패턴 : https://www.slideshare.net/jinhwason/ss-60227208

[9] [Design pattern] 디자인 패턴이란 : https://alleysark.tistory.com/197

[10] Design Pattern : https://dreamlog.tistory.com/577

Thank you!