



디자인 패턴

소프트웨어 패턴

[디자인 패턴이란?](#)

[아키텍처 패턴이란?](#)

[아키텍처 패턴과 디자인 패턴의 차이](#)

[소프트웨어 패턴이 개발에 필요한 이유](#)

아키텍처 패턴

1 [MVC \(Model-View-Controller\) 패턴](#)

[MVC 패턴 흐름](#)

[MVC 패턴 장단점](#)

2 [MVP \(Model-View-Presenter\) 패턴](#)

[MVP 패턴 흐름](#)

[MVP 패턴 장단점](#)

3 [MVVM \(Model-View-ViewModel\) 패턴](#)

[MVVM 패턴 흐름](#)

[MVVM 패턴 장단점](#)

소프트웨어 패턴

“바퀴를 다시 발명하지 마라(*Don't reinvent the wheel*)”

이미 만들어져서 잘 되는 것을 처음부터 다시 만들 필요가 없다는 의미이다.

디자인 패턴이란?

- 객체 지향 프로그래밍을 설계할 때 자주 발생하는 문제에 대해서 피하기 위해 사용되는 패턴을 의미한다.

- 소수의 뛰어난 엔지니어가 해결한 문제를 다수의 엔지니어들이 처리할 수 있도록 한 규칙이면서 구현자들 간의 커뮤니케이션의 효율성을 높이는 기법이다.

예) 싱글톤, 커맨드, 빌더, 옵저버 패턴 등 ...

아키텍처 패턴이란?

- 소프트웨어의 구조를 패턴화 한 것을 의미한다.
- 소프트웨어 아키텍처의 공통적인 발생 문제에 대한 재사용 가능한 해결책을 의미한다.

예) MVC, MVP, MVVM 패턴 등 ...

아키텍처 패턴과 디자인 패턴의 차이

아키텍처 패턴은 소프트웨어 구조 자체를 패턴화 한 것이고 디자인 패턴은 소프트웨어 구조 내에서 특정 문제를 피하기 위해 사용되는 패턴이므로 디자인 패턴 상위에 아키텍처 패턴이 존재한다.

소프트웨어 패턴이 개발에 필요한 이유

- 시행착오를 줄여 개발 시간을 단축 시키고, 고품질의 소프트웨어를 생산할 수 있다.
- 검증된 구조로 개발하기 때문에 안정적인 개발이 가능하다.
- 이해관계자들이 공통된 아키텍처를 공유할 수 있어 의사소통이 간편해진다.

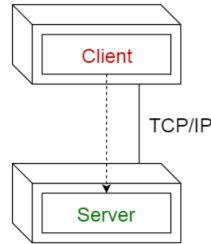
아키텍처 패턴

2. 클라이언트-서버 패턴 (Client-server pattern)

이 패턴은 하나의 서버와 다수의 클라이언트, 두 부분으로 구성된다. 서버 컴포넌트는 다수의 클라이언트 컴포넌트로 서비스를 제공한다. 클라이언트가 서버에 서비스를 요청하면 서버는 클라이언트에 게 적절한 서비스를 제공한다. 또한 서버는 계속 클라이언트로부터의 요청을 대기한다.

활용

- 이메일, 문서 공유 및 은행 등의 온라인 애플리케이션

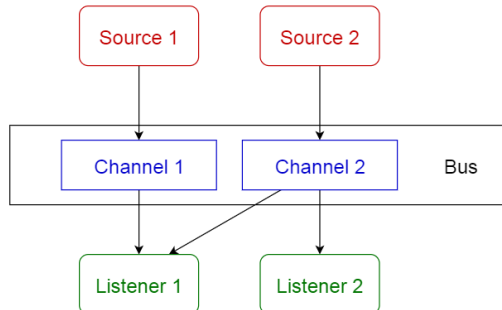


7. 이벤트-버스 패턴 (Event-bus pattern)

이 패턴은 주로 이벤트를 처리하며 이벤트 소스 (event source), 이벤트 리스너 (event listener), 채널 (channel) 그리고 **이벤트 버스 (event bus)**의 4가지 주요 컴포넌트들을 갖는다. 소스는 이벤트 버스를 통해 특정 채널로 메시지를 발행하며 (publish), 리스너는 특정 채널에서 메시지를 구독한다 (subscribe). 리스너는 이전에 구독한 채널에 발행된 메시지에 대해 알림을 받는다.

활용

- 안드로이드 개발
- 알림 서비스



4. 파이프-필터 패턴 (Pipe-filter pattern)

이 패턴은 데이터 스트림을 생성하고 처리하는 시스템에서 사용할 수 있다. 각 처리 과정은 필터 (filter) 컴포넌트에서 이루어지며, 처리되는 데이터는 **파이프 (pipes)**를 통해 흐른다. 이 파이프는 버퍼링 또는 동기화 목적으로 사용될 수 있다.

활용

- 컴파일러. 연속한 필터들은 어휘 분석, 파싱, 의미 분석 그리고 코드 생성을 수행한다.
- 생물정보학에서의 워크플로우



8. 모델-뷰-컨트롤러 패턴 (Model-view-controller pattern)

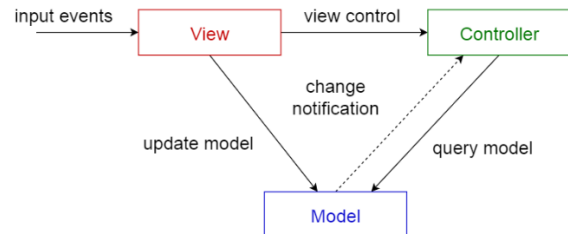
MVC 패턴이라고도 하는 이 패턴은 대화형 애플리케이션 (interactive application)을 다음의 3 부분으로 나눈다.

- 모델 (model) — 핵심 기능과 데이터를 포함한다
- 뷰 (view) — 사용자에게 정보를 표시한다 (하나 이상의 뷰가 정의될 수 있음)
- 컨트롤러 (controller) — 사용자로부터의 입력을 처리한다

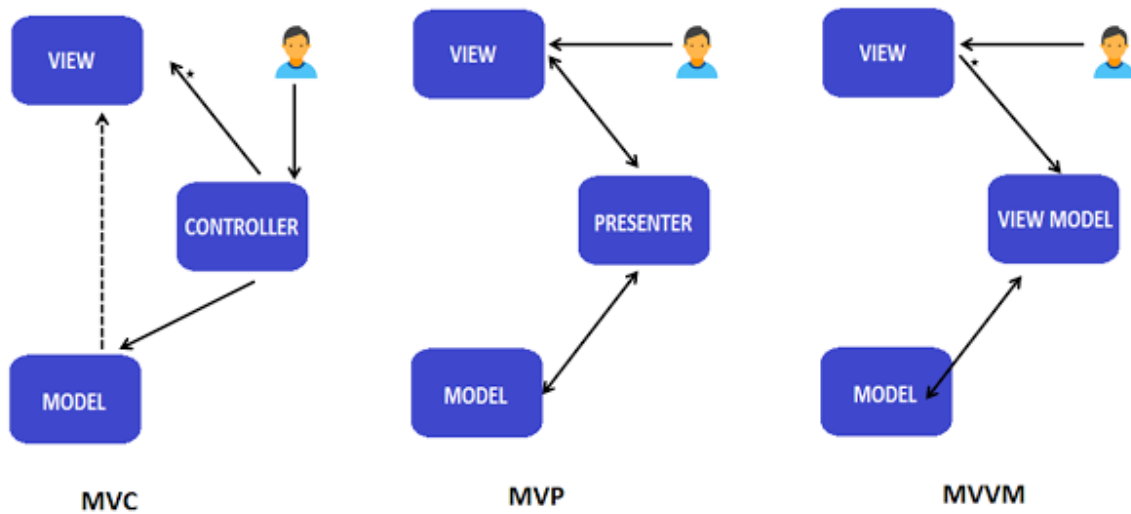
이는 정보가 사용자에게 제공되는 방식과 사용자로부터 받아 들여지는 방식에서 정보의 내부적인 표현을 분리하기 위해 나뉘어진다. 이는 컴포넌트를 분리하며 코드의 효율적인 재사용을 가능케한다.

활용

- 일반적인 웹 애플리케이션 설계 아키텍처
- **Django**나 **Rails**와 같은 웹 프레임워크



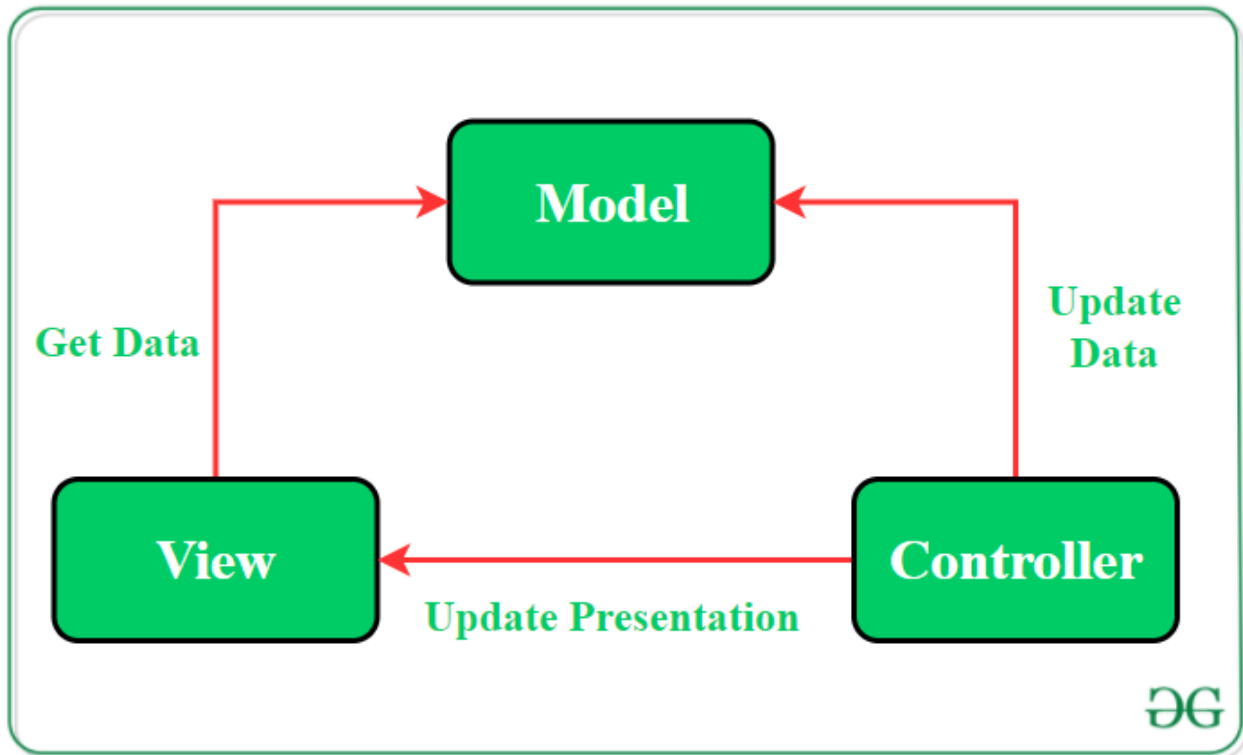
- 아키텍처 패턴 일부



- 자주 쓰이는 아키텍처 패턴 3가지

1 MVC (Model-View-Controller) 패턴

- **Model-View-Controller**로 애플리케이션을 세 가지의 계층으로 구분한 방법론을 의미한다.
- 사용자가 처음 페이지를 출력하는 경우 **Controller**로 요청이 발생하고 **Model**에서 데이터를 가져와서 그 정보를 바탕으로 시각적 표현인 **View**를 그려주는 아키텍처 패턴이다.
- 웹 애플리케이션에서 가장 많이 쓰이는 아키텍처 패턴 중 하나이다.



계층	설명
Model	애플리케이션에서 데이터를 저장하고 처리하는 계층
View	애플리케이션에서 사용자가 직접 보는 화면(UI)을 담당하는 계층
Controller	View와 Model간의 관계를 설정하는 계층이며, 해당 부분에서 애플리케이션의 로직을 담당하는 계층 (View에서 UI를 갱신하고 Model에서는 데이터를 업데이트 하는 구조)

MVC 패턴 흐름



1. 사용자의 Action을 Controller 에서 받는다. (**사용자 Action** → **Controller**)
2. Controller 에서는 이를 확인하고 Model 을 업데이트한다. (**Controller** → **Model**)

3. 수정된 값을 `Controller` 로 반환한다. (*Model* → *Controller*)
4. `Controller`에서는 `View`를 수정한다. (*Controller* → *View*)
5. 사용자에게 변경된 화면을 반환한다. (*View* → 사용자)

MVC 패턴 장단점

장점

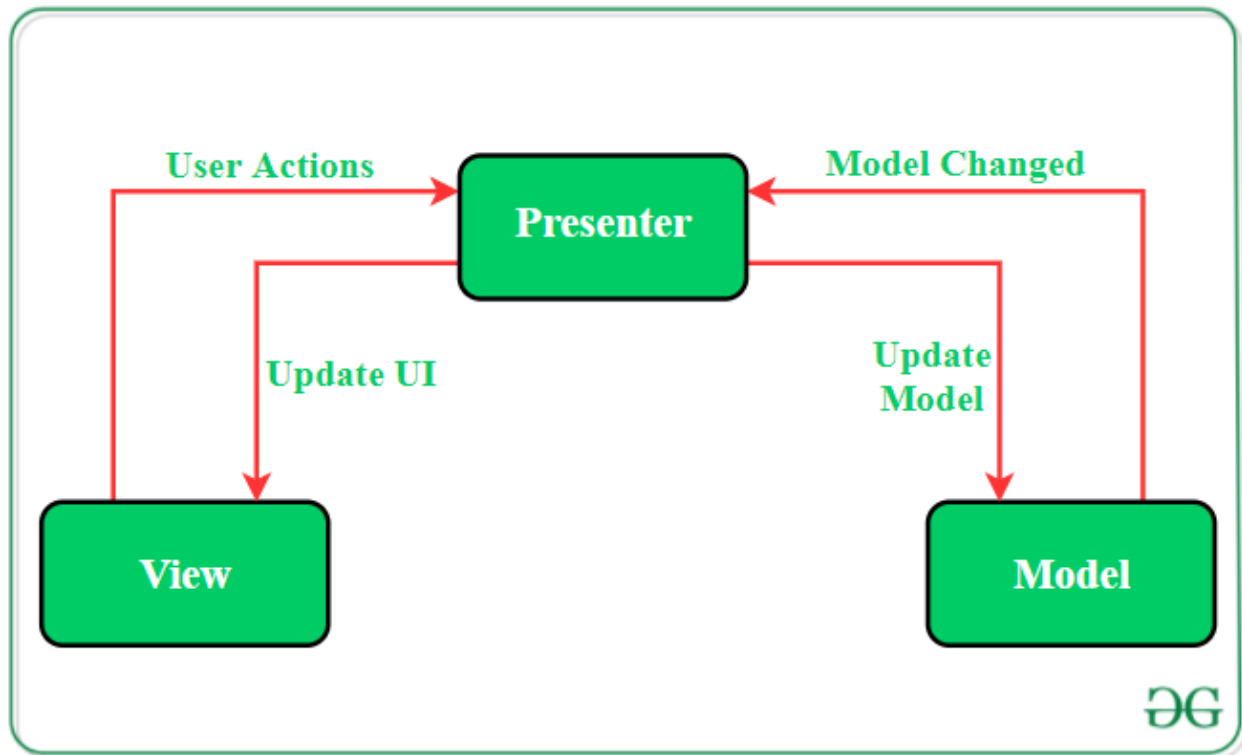
- 가장 단순한 패턴으로 여러 개발 분야에서 보편적으로 사용되는 디자인 패턴이다.

단점

- `View`와 `Model` 사이의 의존성이 높다.
→ 애플리케이션이 커질수록 유지보수가 어려워진다.

2 MVP (Model-View-Presenter) 패턴

- **Model-View-Presenter**로 애플리케이션을 세 가지의 계층으로 구분한 방법론을 의미한다.
- MVC의 구조를 보완하기 위해 나온 아키텍처 패턴으로, MVC의 `Controller` 역할을 MVP의 `Presenter`가 한다고 생각하면 된다.
- `View`와 `Model`을 분리하고 서로 간에 상호작용을 `Presenter`에서 그 역할을 해줌으로써 서로 간의 의존성을 최소화하는 특징이 있는 아키텍처 패턴이다.



Presenter	뷰와 모델을 완전히 분리하고 서로간의 의존성을 없앴다. 🙌 (View ↔ Presenter / Model ↔ Presenter 구조) Presenter 는 Model 을 참조하고 있다가 Update가 발생시 View 를 업데이트 한다. 또한, Presenter 는 이벤트 발생시 View 를 참조해서 Model 의 데이터를 업데이트 한다.
------------------	---

MVP 패턴 흐름

🌱 사용자의 Action → View → Presenter → Model → Presenter → View → 사용자

1. 사용자의 Action을 View 에서 받는다. (**사용자 Action** → **View**)
2. View 에서는 Presenter 로 요청을 한다. (**View** → **Presenter**)
3. Presenter 에서는 Model 로 데이터를 요청한다. (**Presenter** → **Model**)
4. Model 은 Presenter 로 데이터를 전달한다. (**Model** → **Presenter**)

5. **Presenter** 는 **View** 에게 데이터를 전달한다. (**Presenter** → **View**)
6. **View** 에서 사용자로 화면을 보여준다. (**View** → **사용자**)

MVP 패턴 장단점

장점

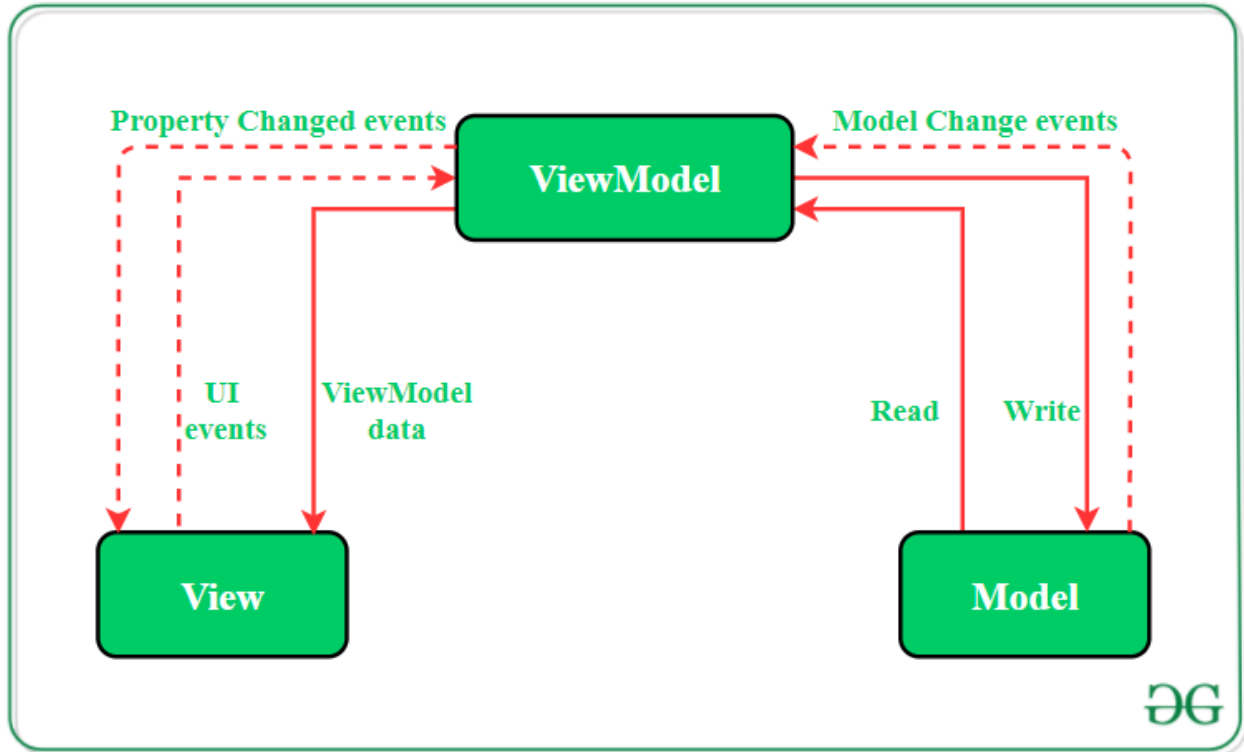
- **View** 와 **Model** 사이의 의존성이 없다.

단점

- **View** 와 **Presenter** 사이의 의존성이 높다.
→ 애플리케이션이 커질수록 이 의존성은 더 강해진다.


3 MVVM (Model-ViewModel-Model) 패턴

- **Model-ViewModel-Model**로 애플리케이션을 세 가지의 계층으로 구분한 방법론을 의미한다.
- **비즈니스 로직**과 **프레젠테이션 로직**을 분리하여 유지 및 보수가 용이해진 디자인 패턴이다.



ViewModel	애플리케이션에서 View 와 Model 사이에 존재하며, 서로 간의 중재를 하는 역할을 수행한다. View ↔ ViewModel : 사용자와의 뷰의 상호작용(클릭, 키보드 동작 등)을 수신하여 이에 대한 처리를 View 와 ViewModel 을 연결하고 있는 데이터 바인딩을 통해 서로간을 연결한다. Model → ViewModel: 사용자의 데이터의 변경이 발생하는 경우 데이터를 가져오거나 갱신 한 뒤 View 에게 전달하여 사용자에게 전달하는 역할을 수행한다.
+추가 개념 Data Binding	Android 에서는 View 에 해당하는 xml파일과 데이터를 연결해서 하나로 묶는 작업을 의미한다. 이를 통해서 데이터를 UI 요소에 연결하여 필요한 코드를 최소화 할 수 있는 장점이 있다.

MVVM 패턴 흐름

 사용자 Action → View → ViewModel → Model → ViewModel → View → 사용자

1. 사용자가 입력한 값이 **View**를 통해 들어온다. (사용자 → View)

2. **View**에 입력값이 들어오면 **ViewModel**로 입력 값을 전달한다. (**View** → **ViewModel**)
3. 전달 받은 **ViewModel**은 **Model**에게 데이터 요청을 보낸다. (**ViewModel** → **Model**)
4. **Model**은 **ViewModel**에게 요청 받은 데이터를 Response 한다. (**ViewModel** ← **Model**)
5. **ViewModel**은 그 값을 처리하여 내부에 저장한다.
6. **View**는 **ViewModel**과의 **데이터 바인딩**을 통해 화면상에 표출한다. (**View** ↔ **ViewModel**)

MVVM 패턴 장단점

장점

- **View**와 **Model** 사이의 의존성이 없다.
- 데이터 바인딩을 사용하여 **View**와 **ViewModel** 사이의 의존성이 없다.
- 개발 기간 동안 개발자와 디자이너가 동시에 독립적(병렬적) 작업이 가능하다.
→ UI 디자인이 나오지 않더라도 뷰와 뷰 모델을 먼저 개발 가능

단점

- 거대하고 복잡한 앱을 위해 고안된 패턴이기에 소형 앱에서 사용하게 되면 **오버헤드**가 커진다.

▼ 오버헤드 ?

어떤 처리를 하기 위해 들어가는 간접적인 처리 시간 · 메모리 등

- 구조가 복잡하여 설계가 쉽지 않다.

참고

<https://adjh54.tistory.com/60>

<https://ko.wikipedia.org/wiki/오버헤드>

https://ko.wikipedia.org/wiki/아키텍처_패턴

https://ko.wikipedia.org/wiki/디자인_패턴

<https://velog.io/@blucky8649/MVC-MVP-MVVM-패턴의-특징>

