# Architect DESIGN

Effective java 4주차 미니 세미나

# Architect Design

- 주어진 상황에서의 소프트웨어 아키텍쳐에서 일반적으로 발생하는 문제점들에 대한 일반화되고 재사용 가능한 솔루션
  - 아키텍쳐 패턴은 소프트웨어 디자인 패턴과 유사하지만 더 큰 범주에 속한다

1.계층화 패턴 (Layered pattern)
2.클라이언트-서버 패턴 (Client-server pattern)
3.마스터-슬레이브 패턴 (Master-slave pattern)
4.파이프-필터 패턴 (Pipe-filter pattern)
5.브로커 패턴 (Broker pattern)
6.피어 투 피어 패턴 (Peer-to-peer pattern)
7.이벤트-버스 패턴 (Event-bus pattern)
8.MVC 패턴, MVP 패턴, MVVM 패턴
9.블랙보드 패턴 (Blackboard- pattern)
10.인터프리터 패턴 (Interpreter pattern)

# Architect Design

- 3계층 애플리케이션 아키텍처
  - Presentation Layer
  - Business Layer
  - Data Access Layer

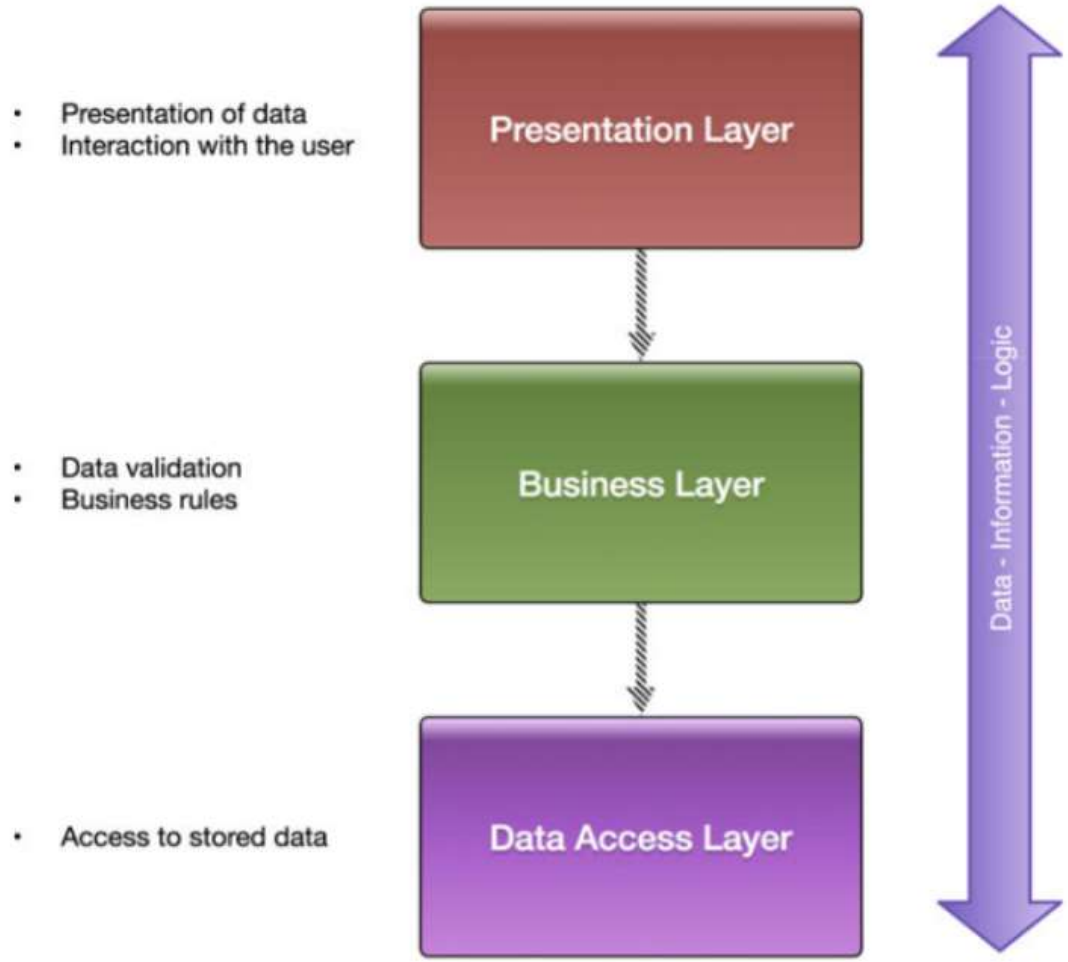
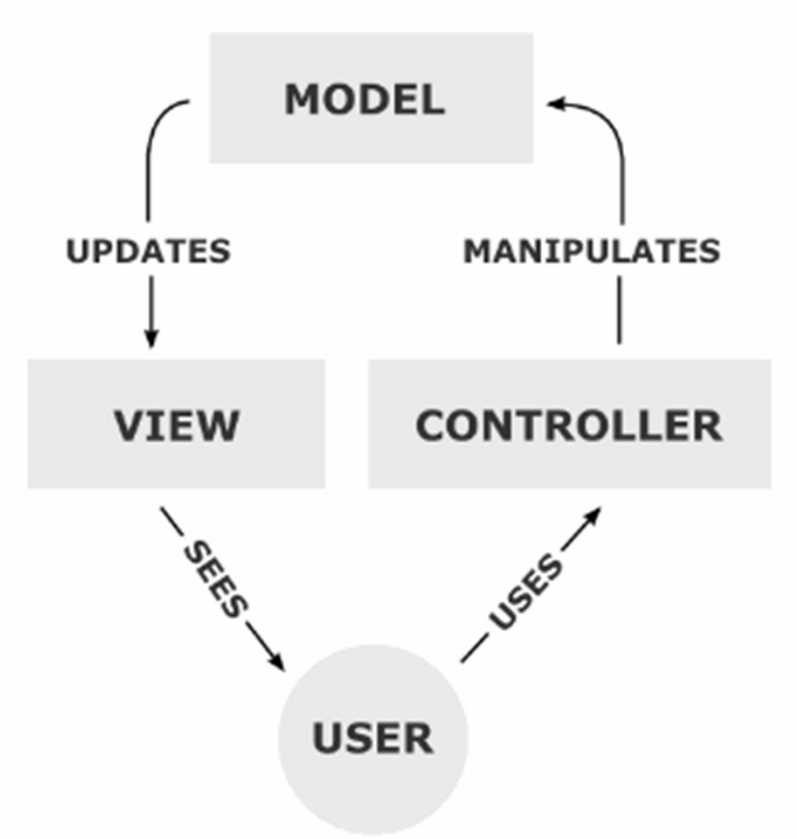



***Figure 1-1.*** *Three-tier application structure*

# MVC

- Model + View + Controller




1.사용자가 웹사이트에 접속한다. (Uses)
2.Controller는 사용자가 요청한 웹페이지를 서비스 하기 해서 모델을 호출한다. (Manipulates)
3.모델은 데이터베이스나 파일과 같은 데이터 소스를 제어 후에 그 결과를 리턴한다.
4.Controller는 Model이 리턴한 결과를 View에 반영한다 (Updates)
5.데이터가 반영된 View는 사용자에게 보여진다. (Sees)

# MVC

- 3계층 구조에서의 단점

기능의 중복

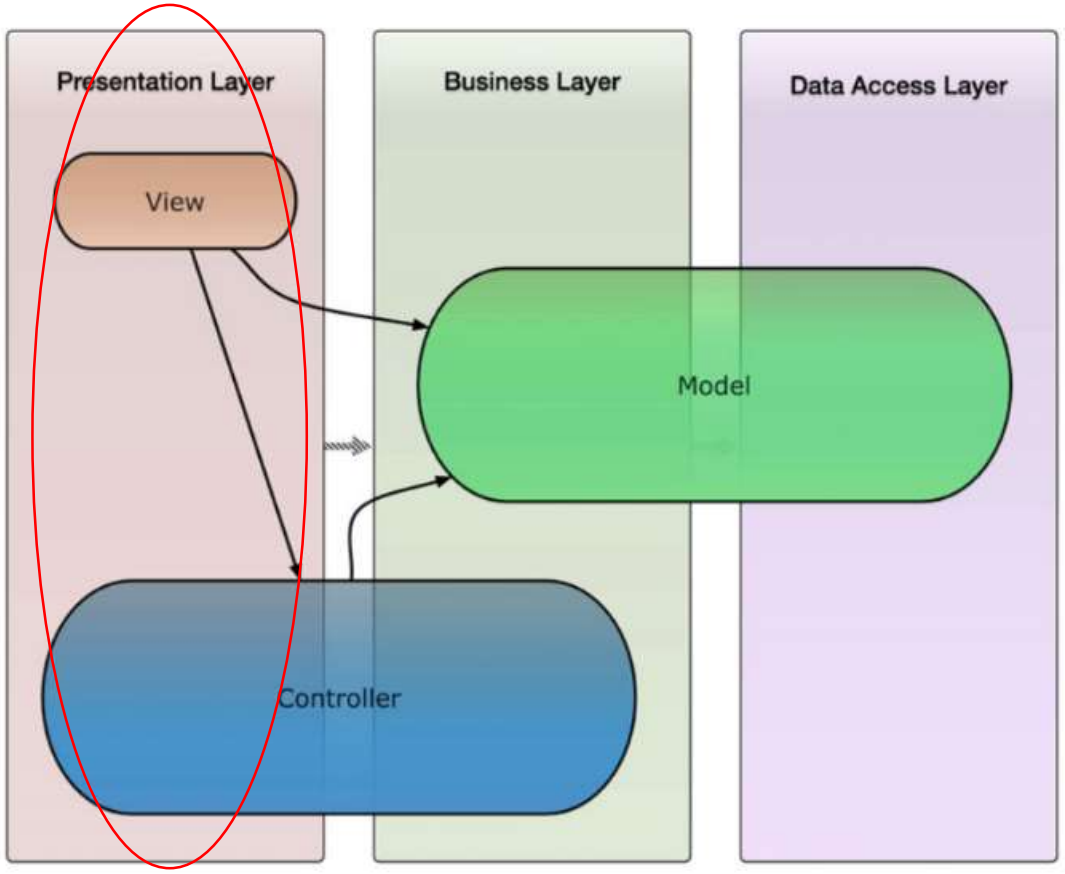
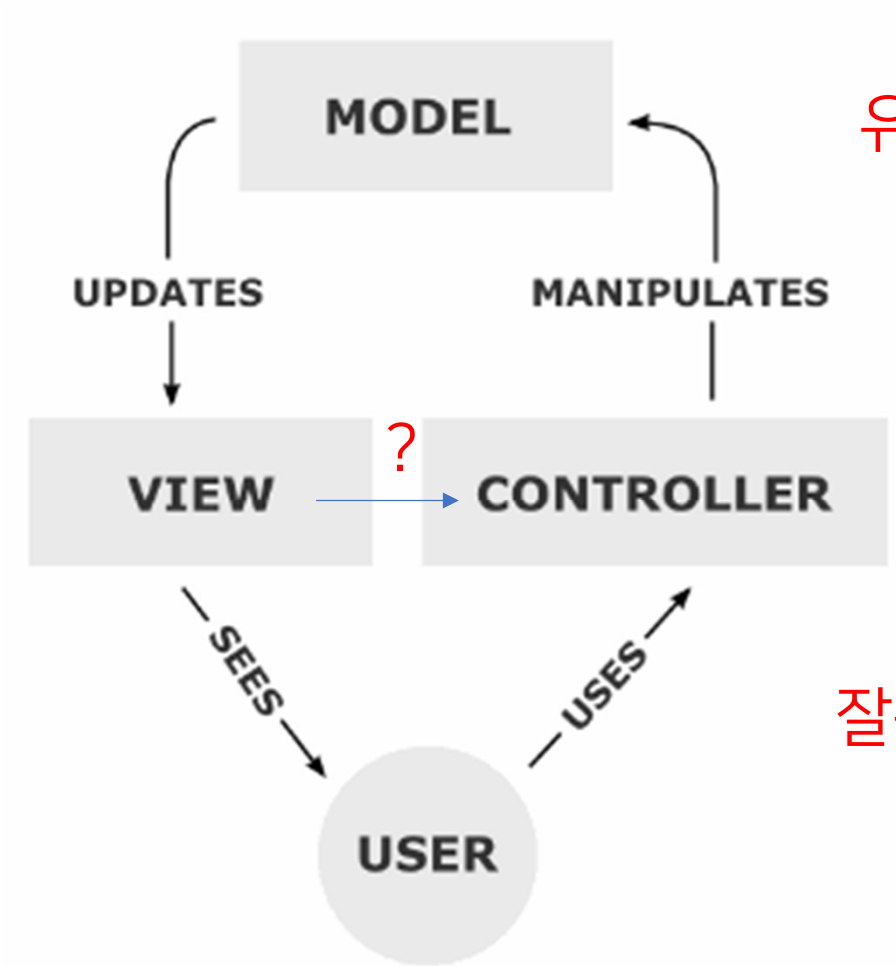



***Figure 1-3.*** *Relationship between the three-tier design and MVC*

# MVC




유효성 검사 확인

다시 컨트롤러 수정

값 변경을 위한
프로그래밍 로직 구현

잘못된 값 입력

# MVP

- MVC의 단점을 해결하기 위해 제시
- Controller -> Presenter
- View와 Model이 분리

Figure 1-5 shows the three-tier structure for the MVP pattern.

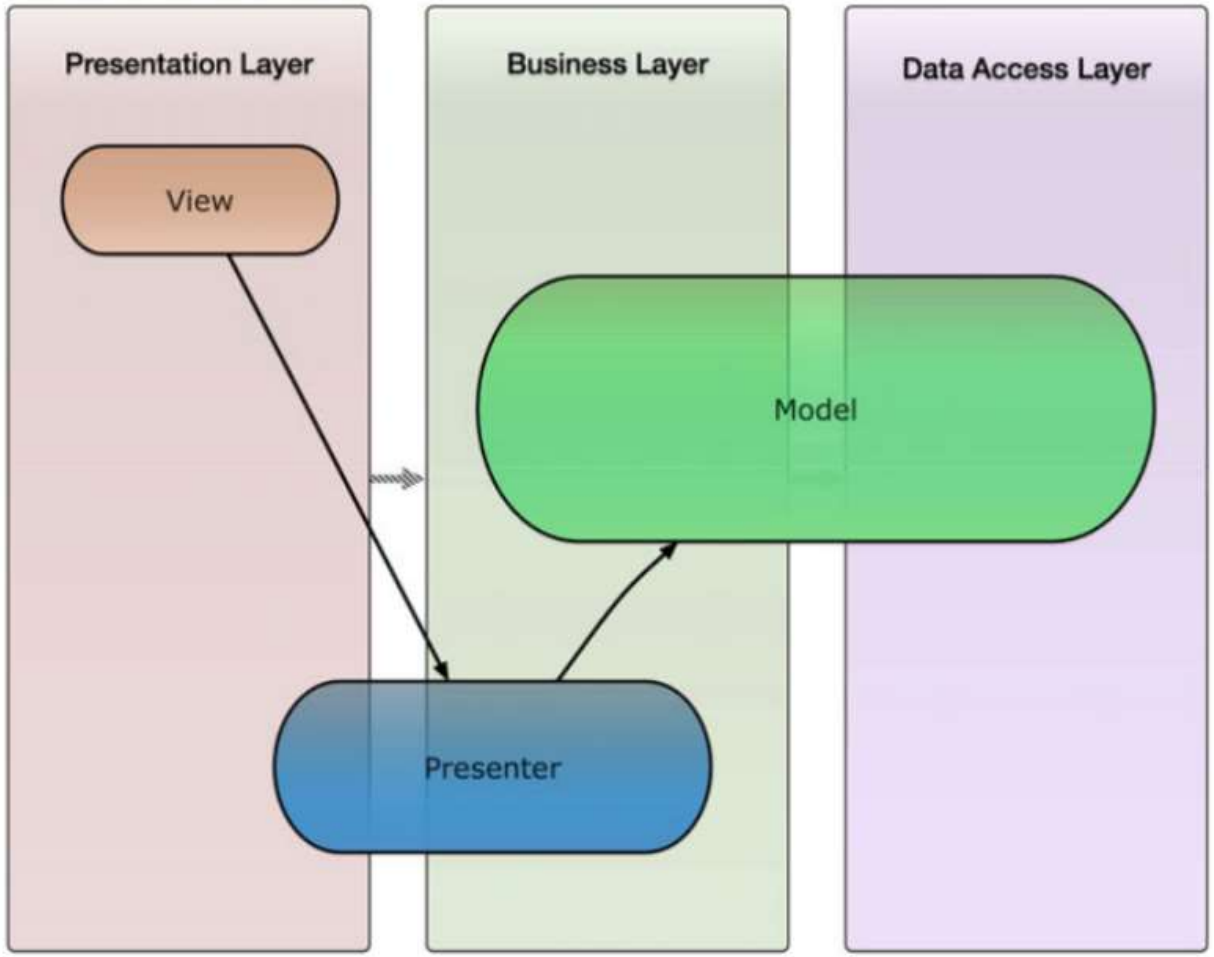
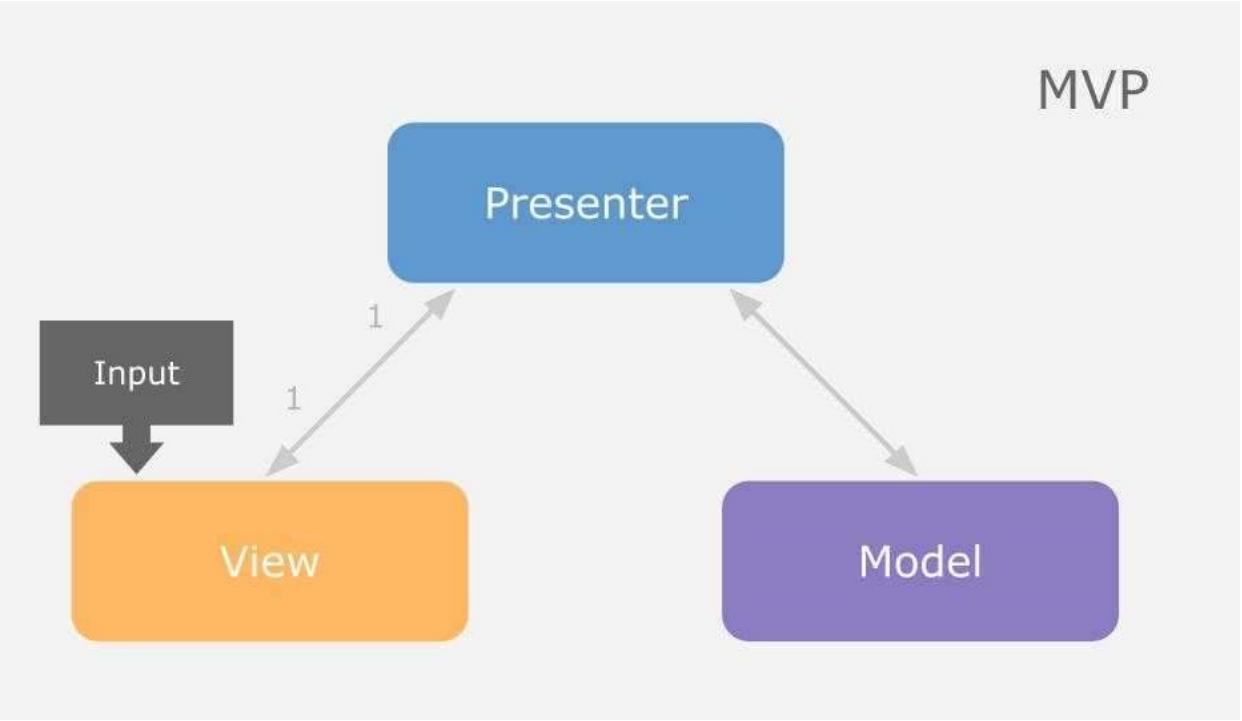



***Figure 1-5.*** *Relationship between the three-tier design and MVP*

# MVP




1.사용자의 input이 View를 통해 들어온다.
2.View는 필요한 데이터를 Presenter에게 요청
3.Presenter는 Model에게 데이터를 요청한다.
4.Model은 변경된 데이터를 Presenter에게 전달
5.Presenter는 View에게 데이터를 전달한다.
6.View는 받은 데이터를 UI에 표현한다.

Presenter에 너무 큰 의존도

# MVVM

- MVP의 단점을 해결하기 위해 제시
- 기존의 View는 너무 의존적
- ViewModel
- Command 패턴, 데이터 바인딩

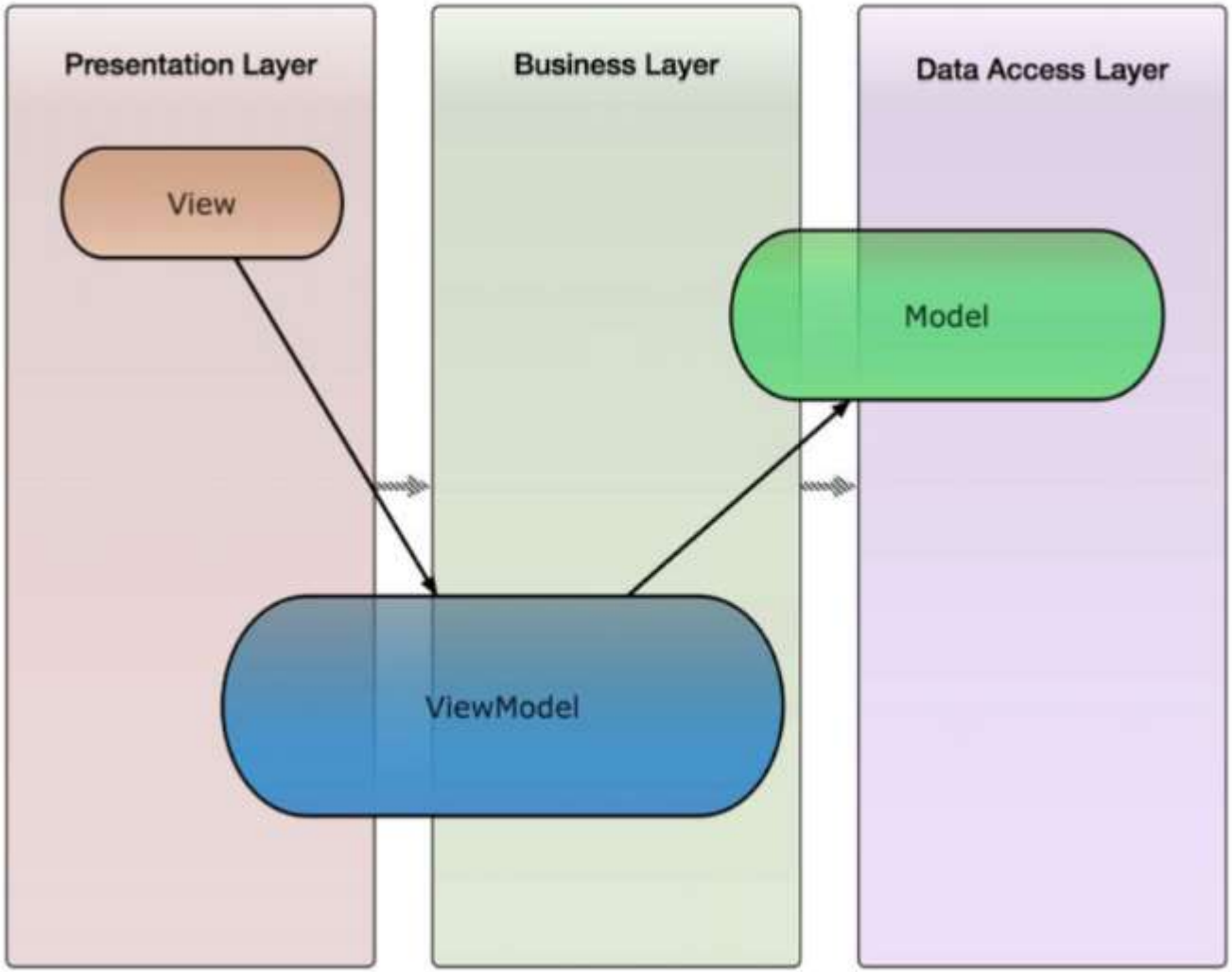
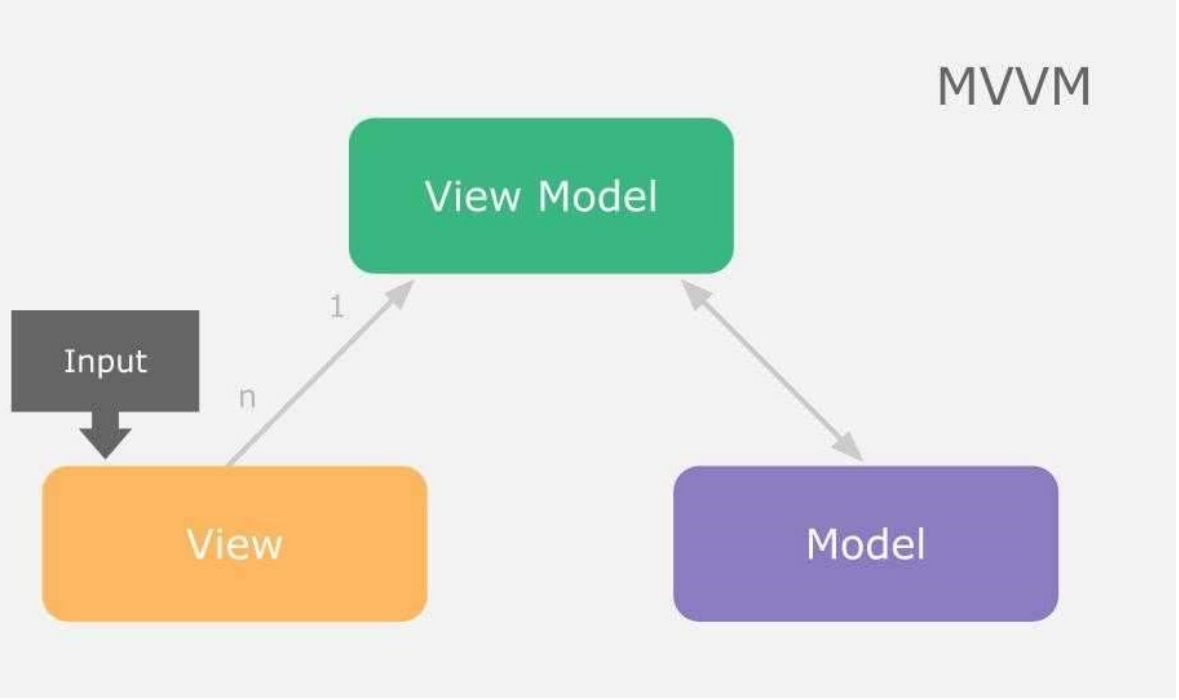



***Figure 1-9.*** *Relationship between the three-tier design and MVVM*

# MVVM




1.사용자의 input이 View를 통해 들어온다.
2.View는 필요한 데이터를 **Command 패턴을** 통해 View Model에게 요청한다.

- Command 패턴
  - 요청을 객체의 형태로 캡슐화하여 사용자가 낸 요청을 **나중에 이용할 수 있도록** 매서드 름, 매개변수 등 요청에 필요한 정보를 저장 는 로깅, 취소할 수 있게 하는 패턴이다.

# MVVM

ommand 클래스

```
public interface Command { public abstract void execute(); }
```

utton 클래스

```
public class Button {
private Command theCommand;
// 생성자에서 버튼을 눌렀을 때 필요한 기능을 인지로 받는다.
public Button(Command theCommand) { setCommand(theCommand); }
public void setCommand(Command newCommand) { this.theCommand = newCommand
// 버튼이 눌리면 주어진 Command의 execute 메서드를 호출한다.
public void pressed() { theCommand.execute(); }
```

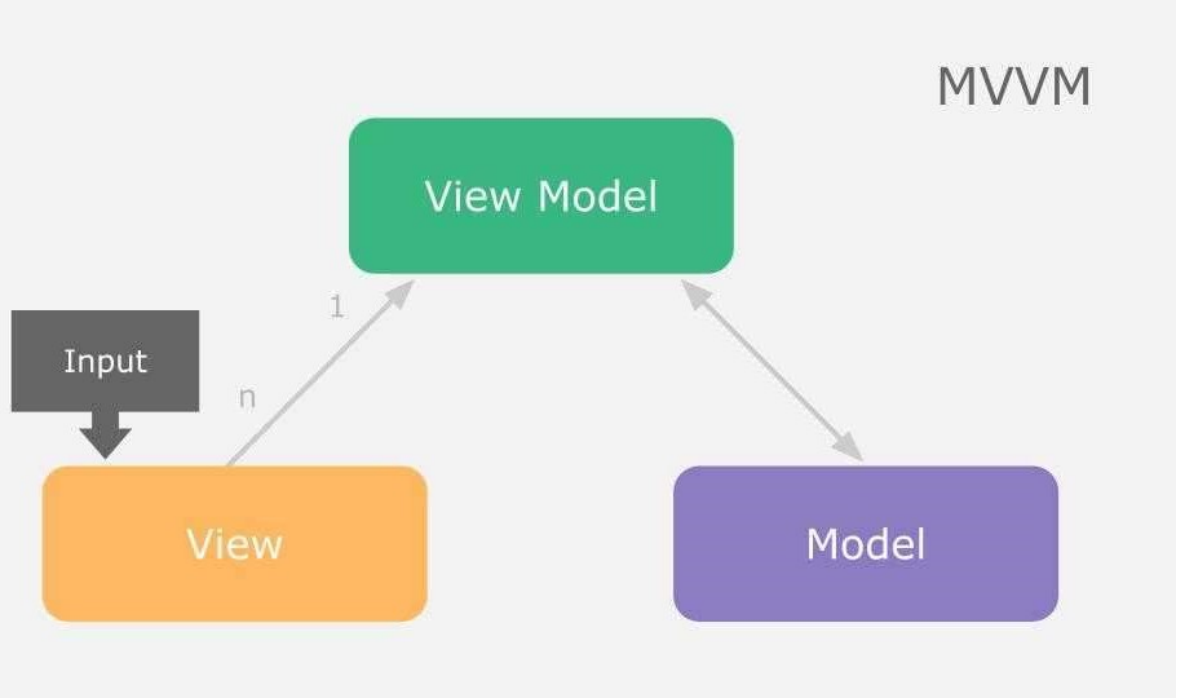
Lamp, LampOnCommand 클래스

```
public class Lamp {
public void turnOn(){ System.out.println("Lamp On"); }
}
/* 램프를 켜는 LampOnCommand 클래스 */
public class LampOnCommand implements Command {
private Lamp theLamp;
public LampOnCommand(Lamp theLamp) { this.theLamp = theLamp; }
// Command 인터페이스의 execute 메서드
public void execute() { theLamp.turnOn(); }
}
```

C:₩Users₩user₩Desktop₩20210315152209.png 저장

- Command 인터페이스를 구현하는 LampOnCommand와 AlarmStartCommand 객체를 Button 객체에 설정한다.
- Button 클래스의 pressed 메서드에서 Command 인터페이스의 execute 메서드를 호출한다.
- 즉, 버튼을 눌렀을 때 필요한 임의의 기능은 Command 인터페이스를 구현한 클래스의 객체를 Button 객체에 설정해서 실행할 수 있다.
- 이렇게 Command 패턴을 이용하면 Button 클래스의 코드를 변경하지 않으면서 다양한 동작을 구현할 수 있게 된다.

# MVVM




3. View Model은 Model에게 필요한 데이터를 요
   다.
4. Model은 View Model에게 필요한 데이터를 응
   다.
5. Data Binding을 통해 View Model의 값이 변하
   View의 정보가 자동으로 변경된다.

- 데이터 바인딩
  - 두 데이터 혹은 정보의 소스를 모두 일
    키는 기법

ViewModel의 설계가 까다롭다

# MVC, MVP, MVVM

- 각 패턴마다 고유의 장단점이 존재

- 프로젝트 크기와 각 상황에 맞게 검토하고 사용

# Reference


- https://m.blog.naver.com/PostView.nhn?blogId=ksieofficial&logNo=221520697316&proxyReferer=https:%2F%2Fwww.google.com%2F
- https://swimjiy.github.io/2019-05-28-web-mvc-mvp-mvvm
- 커맨드 패턴 - https://gmlwjd9405.github.io/2018/07/07/command-pattern.html
- 데이터 바인딩 –
- https://poiemaweb.com/angular-component-data-binding