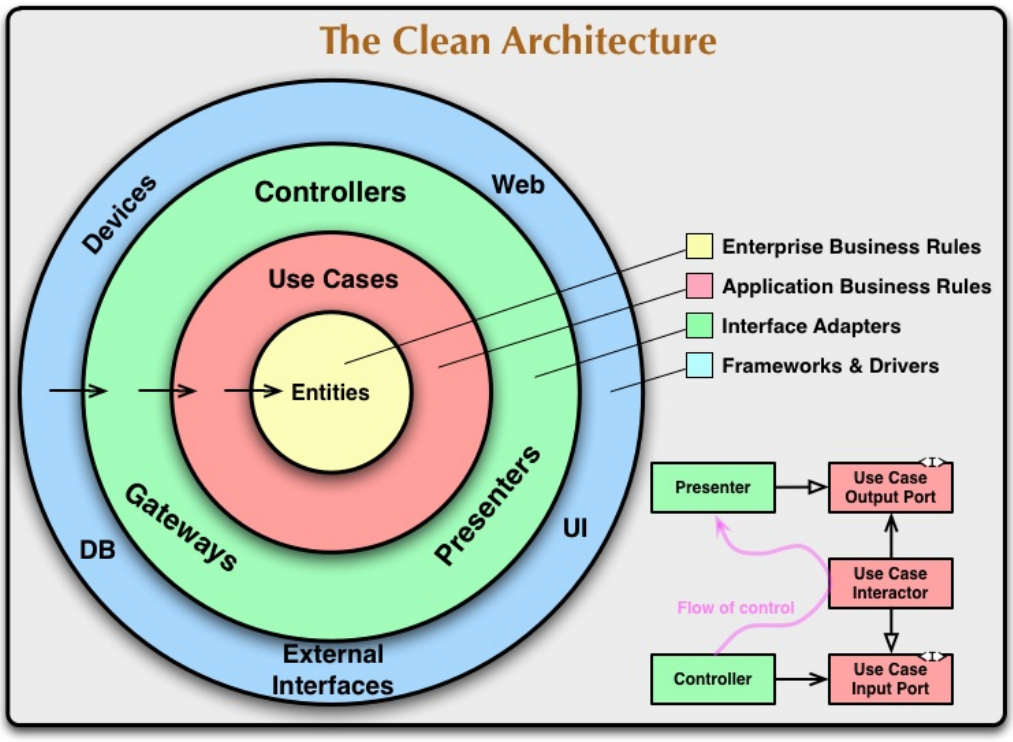
클린아키텍처

🏛️ 앱 아키텍처 **by @kiboom**

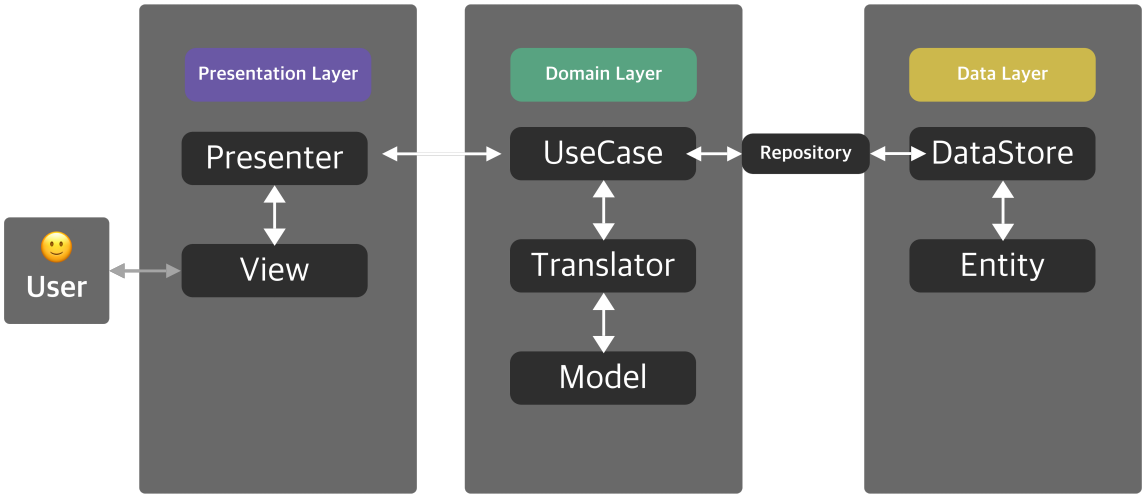


By Robert C. Martin (Clean Code, Clean Architecture, Clean Agile)

화살표의 방향은 의존성

의존성은 밖 ⇒ 안으로 향하고, 바깥 원은 안쪽 원에 영향을 미치지 않는다.

모바일 앱 클린아키텍처



Domain Layer 에서 Data Layer를 알아야지만 데이터를 가져올 수 있는거 아닌가?

의존성 역전(Dependency Inversion)

> 상위 계층(정책 결정)이 하위 계층(세부 사항)에 의존하는 전통적인 의존 관계를 반전(역전)시킴으로써 상위 계층이 하위 계층의 구현으로부터 독립되게 할 수 있다.

👉 **추상화 클래스 활용**

get_it 활용 예시

as-is

```
registerSingleton<AmericanoRecipe>(AmericanoRecipe());

------------------------------------------------------------------------------

print(getIt<AmericanoRecipe>().info);

/// 카페라떼로 변경하려면?
/// 두 곳 모두 바꿔줘야하고 참조도 달라짐
```

to-be

```
/// <Generic 타입>에는 추상화 클래스 넣고 실제 사용할 클래스 인스턴스 등록
registerSingleton<CoffeeRecipe>(AmericanoRecipe());
// registerSingleton<CoffeeRecipe>(CafeLatteRecipe());

------------------------------------------------------------------------------

/// 사용 할 때는 추상화 클래스 타입으로 가져오기
print(getIt<CoffeeRecipe>().info);

/// CoffeeRecipe 추상화 클래스만 참조하면 됨.
```

🥘 화면 (Presentation Layer)

> ## 데이터를 어떻게 보여줄지(입력 받을 지)

- 화면을 어떻게 그릴지를 정의함.

👩🏻‍🍳 Presenter

- Provider / BloC / Cubit / GetxController / ViewModel
- UseCase 로 부터 받은 데이터를 받아서 특정 화면에 최적화된 데이터로 가공
- 로직을 담당
    - 화면에서 발생하는 이벤트로 인한 로직 처리
      사용자 입력이 왔을 때 어떻게 처리 할 지에 대한 판단

🪀 View

- Widget
- Presenter 를 통해 데이터를 받아서 UI를 그려주는 곳
- 사용자에게 이벤트를 받는 곳

🏭 데이터 가공 (Domain Layer)

> **데이터를 어떻게 가공해서 보여줄지(저장할지)**

- **비즈니스 로직**을 정의함.
- 앱의 종류(`Domain`)에 따라 데이터를 가공하는 방법(`Business Logic`)이 달라짐.

🏭 UseCase

- 비즈니스 로직이 들어가는 영역
- 여러 Repository를 취합해서,
  여러 화면들에서 사용될 법한 데이터를 가공함.
- 비즈니스 로직이 단순한 앱에서는 UseCase 대신 Repository만 씀.

🥓 Model

- 앱의 실질적인 데이터 형태

🏪 Repository

- `abstract class`
- input, output 을 정한 메서드 선언만 되어있음

```
abstract class TodoRepository {
  Future<TodoEntity> create(TodoModel todo);

  Future<List<TodoEntity>> read();
}
```

🐖 데이터 (Data Layer)

어떤 데이터를 다룰 것인지

- **데이터의 형태**와 **데이터 저장소**를 정의함.
- 가공되지 않는 데이터 그대로를 다룸.

🐖 Entity

- **DTO** (Data Transfer Object)
  - 계층 간 데이터 교환을 하기 위해 사용하는 객체
- Flutter 에서는 `json_serializable`, `freezed` 활용하여 **Model** 로도 활용

👷 DataSource

- 실제 데이터의 입출력이 실행되는 곳
  - Remote : 서버 API
  - Local : 로컬 DB
- Entity 를 받아오고 저장하는 역할

🏪 Repository

- 도메인 레이어가 필요로 하는 데이터의 CRUD 실질적 구현체
- DataSource 참조하여 Local DB 또는 네트워크 통신 활용
- 데이터의 출처를 따지지 않고 전달하는 것에만 집중함.
  즉, 데이터 출처를 '**주입**' 받음.

```
class TodoRepositoryImpl implements TodoRepository {
  final TodoRemoteDatasource remoteDatasource;

  TodoRepositoryImpl(this.remoteDatasource);

  @override
  Future<TodoModel> create(TodoModel todoModel) {
    return remoteDatasource.create(todoModel);
  }

  @override
  Future<List<TodoModel>> read() {
    return remoteDatasource.getAll();
  }

  /// ...
}
```

활용 사례

테스트 용이

- Layer 단위로 테스트 코드 작성 가능
- Data Layer 만 Mock data 로 교체 후 테스트 용이

```
class MockTodoRepository implements TodoRepository {
  @override
  Future<List<TodoModel>> read() {
    return [TodoModel('Hello', TodoModel('World')];
  }
}
```

“앱 UI 변경 요청드려요. 보여지는 데이터 유형은 같아요.”

👉 **프레젠테이션 레이어만 수정**

“우리 서비스를 웹 또는 데스크탑 앱으로 확장합시다.”

👉 **비즈니스 로직은 거의 동일. 따라서, Domain / Data 레이어 재사용**
프레젠테이션 레이어만 새롭게 추가

꼭 저 많은 클래스들을 다 만들어야할까? 많은 규칙을 모두 지켜야 할까?

- **[Domain]** Model 대신에 **[Data]** Entity 활용
- UseCase 생략 ⇒ **[Domain]** Repository 활용
- DataSource 생략 ⇒ **[Data]** Repository 에 포함
- …

Consider your own context.

실습.

- 완전 초보자 대상 강의가 아니에요.
- 어려워요.
- 한번에 이해 못하는게 당연해요.

학습 순서.

1. 최대한 이해해보려고 노력하면서 강의 내용 다 보기.
    - 클린아키텍처 적용된 샘플 프로젝트 분석
    - 기존 프로젝트에 적용
2. 원칙들을 학습하고 이해하면서 따라하기.
3. 각자 실습 프로젝트에 적용해보기.
4. 여러 고민해보고 여러분만의 아키텍처 만들기.

아키텍처 적용.

폴더링

- presentation
  - IDE replace 활용
  - 상대 경로 단점 체크
- data
- domain

참고 자료

Clean Coder Blog

Over the last several years we've seen a whole range of ideas regarding the architecture of systems. These include: Though these architectures all vary somewhat in their details, they are

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html