MVVM 패턴을 활용한 클린 아키텍처

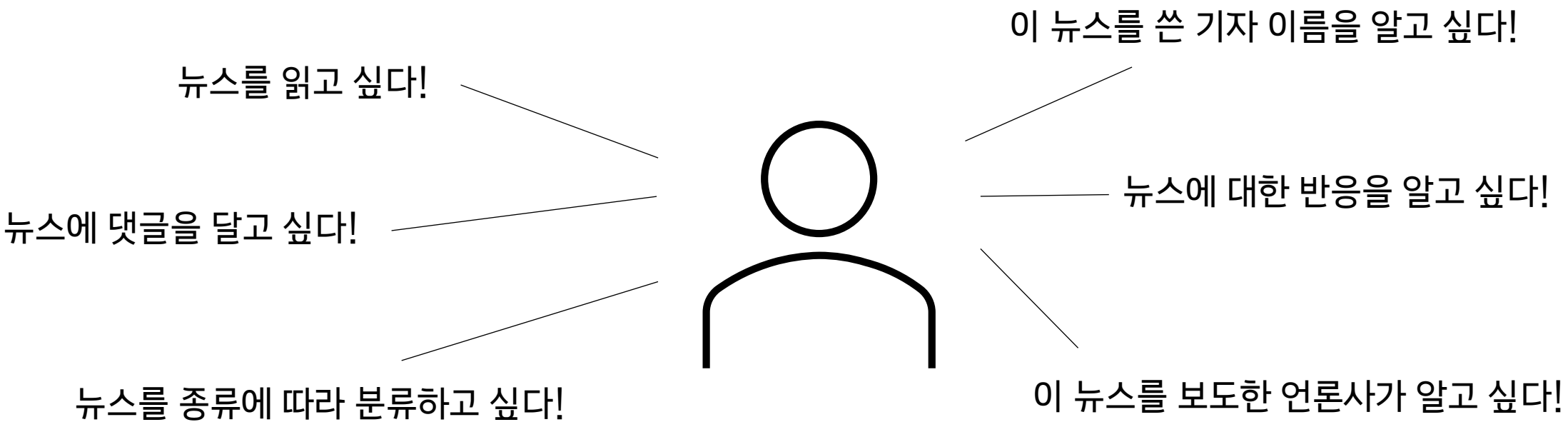
2016010873 박정욱

목차

- 관심사 분리(Separation of Concern, SoC) 원칙
- MVC, MVP, MVVM 패턴의 이해
- Clean Architecture
- Google이 제시하는 Android 앱 권장 아키텍처
- 실제 적용 사례

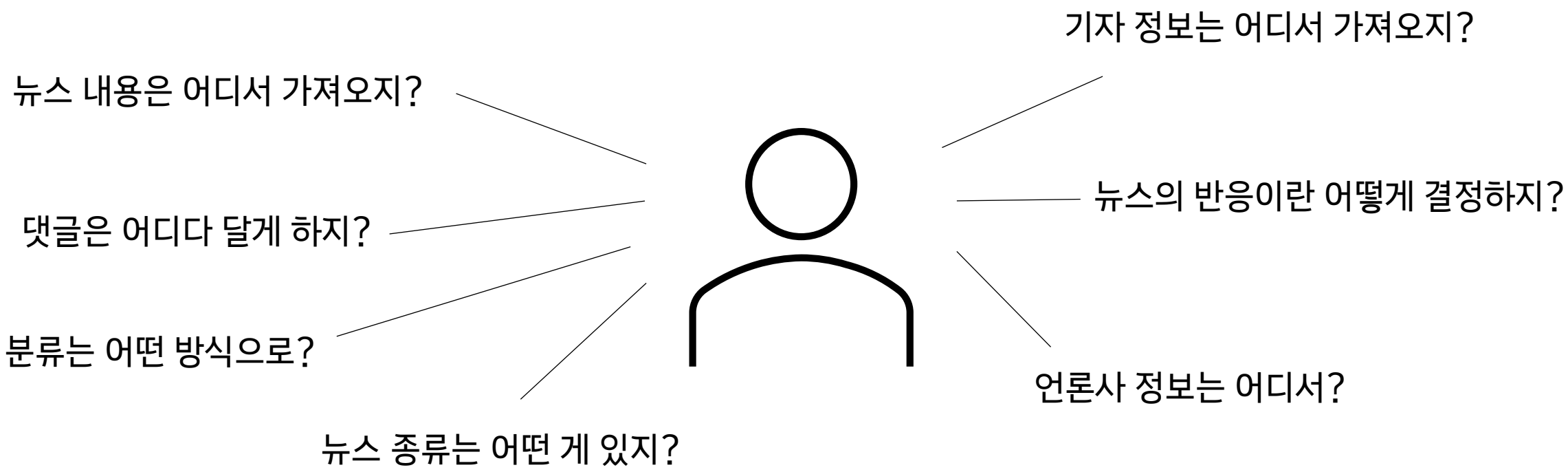
관심사 분리 원칙

- 작업에 영향을 미치는 특정한 정보의 집합, 관심사(Concern)



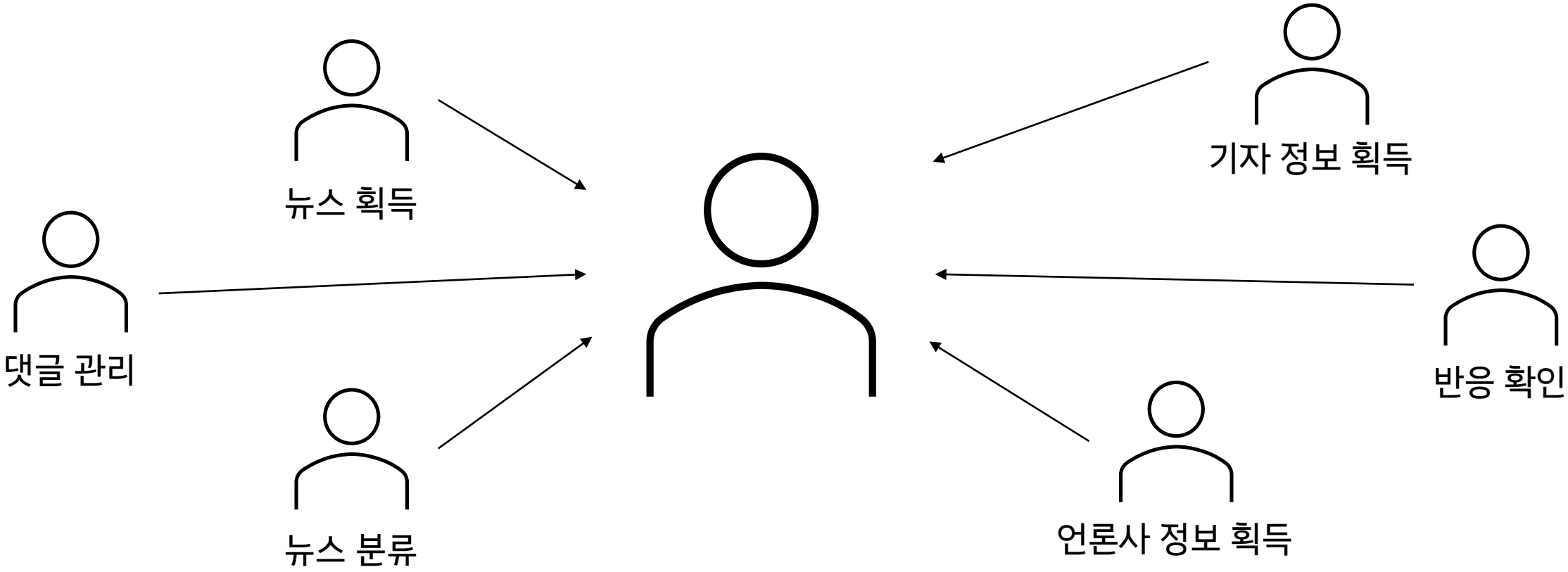
관심사 분리 원칙

- 관심사가 많아질수록, 작업은 복잡해지기 마련

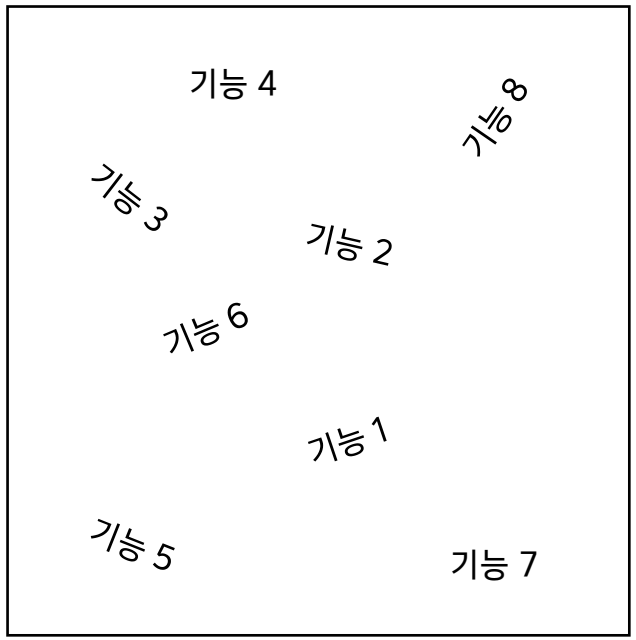


관심사 분리 원칙

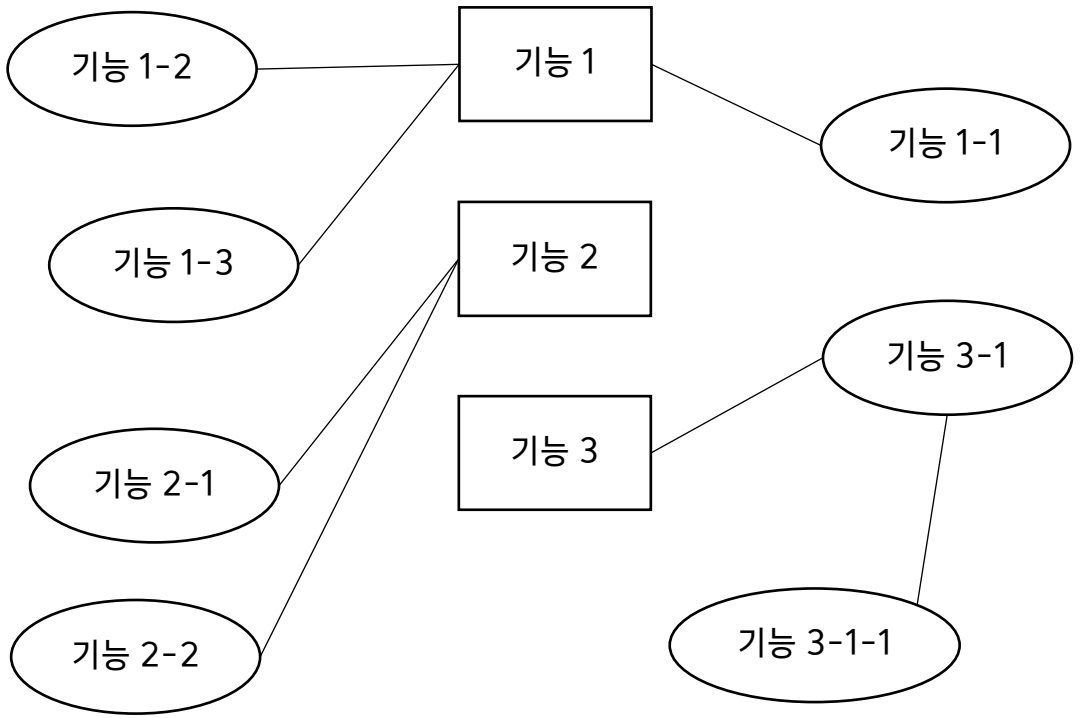
- 서로 관련있는 관심사들을 모아서 따로 처리하게 만들자!



관심사 분리 원칙

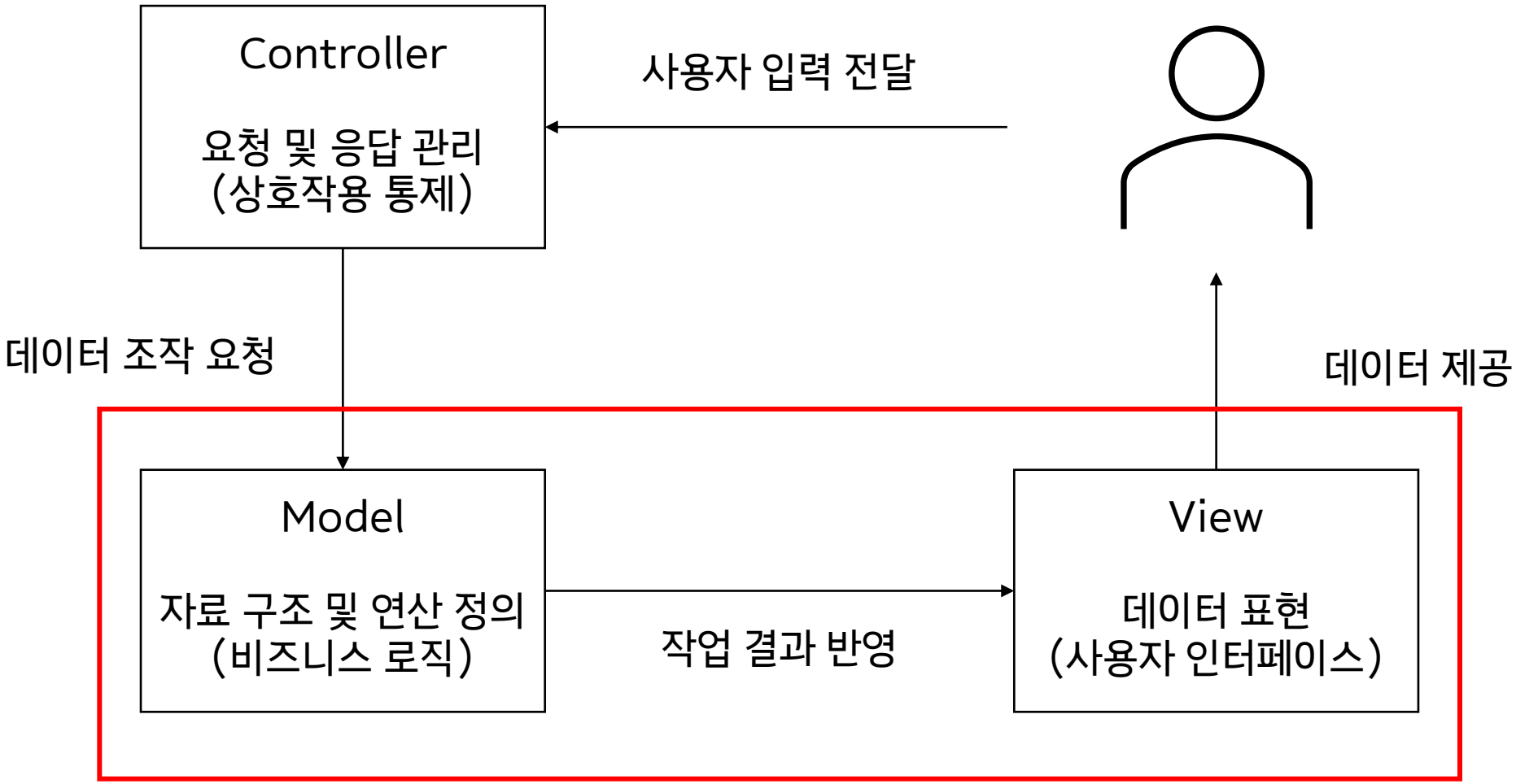


결합도 ↑ - 응집도 ↓

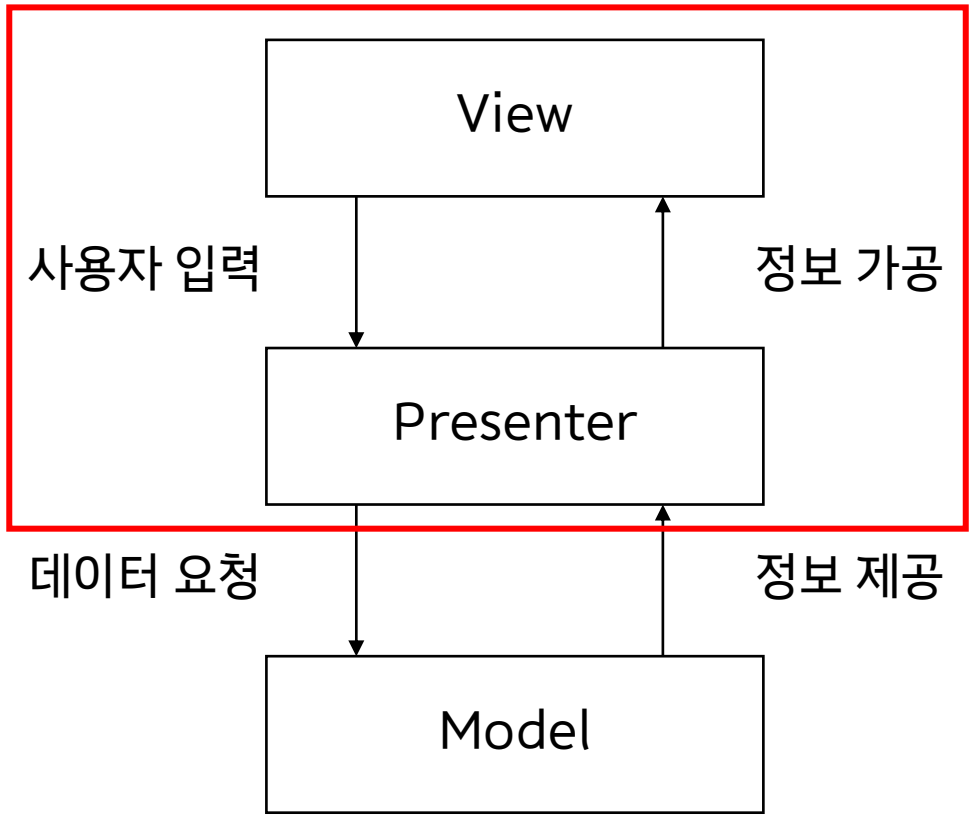


결합도 ↓ - 응집도 ↑

MVC, MVP, MVVM 패턴의 이해

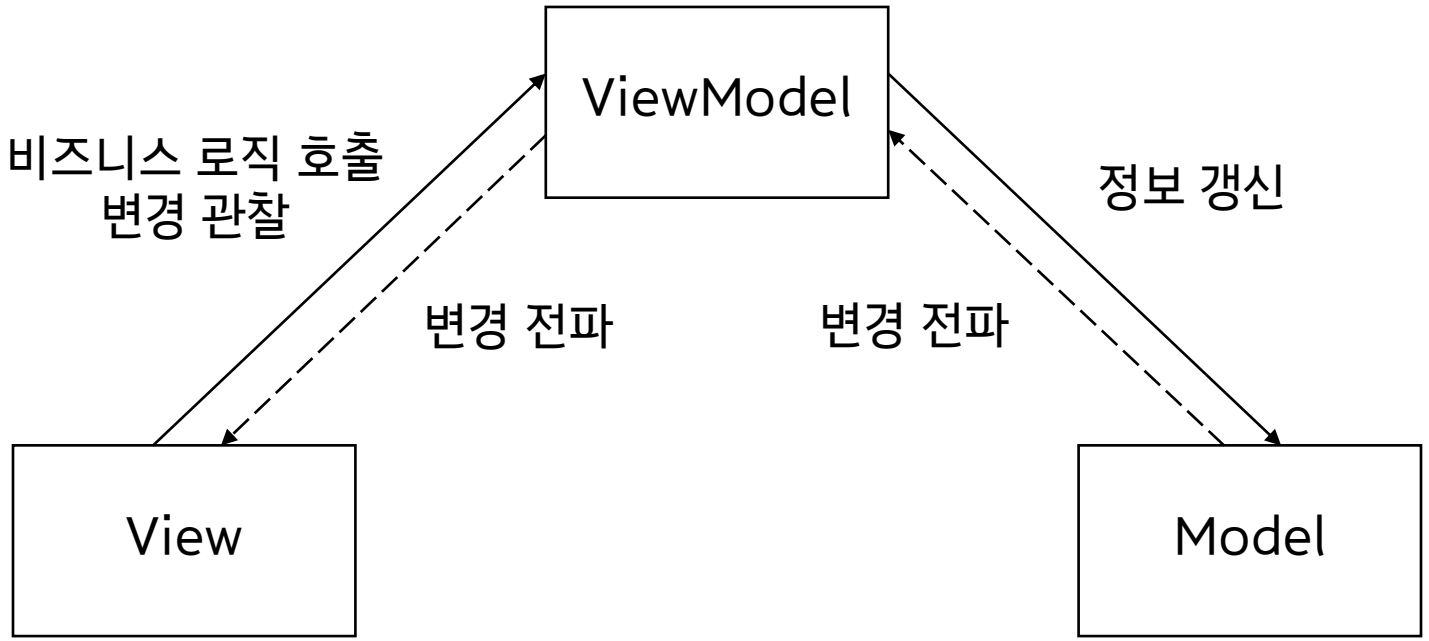


MVC, MVP, MVVM 패턴의 이해



- View와 Model 간 의존성 X
- Model 단위 테스트 O
- Controller의 역할 또한 Presenter가 담당하게 됨

MVC, MVP, MVVM 패턴의 이해

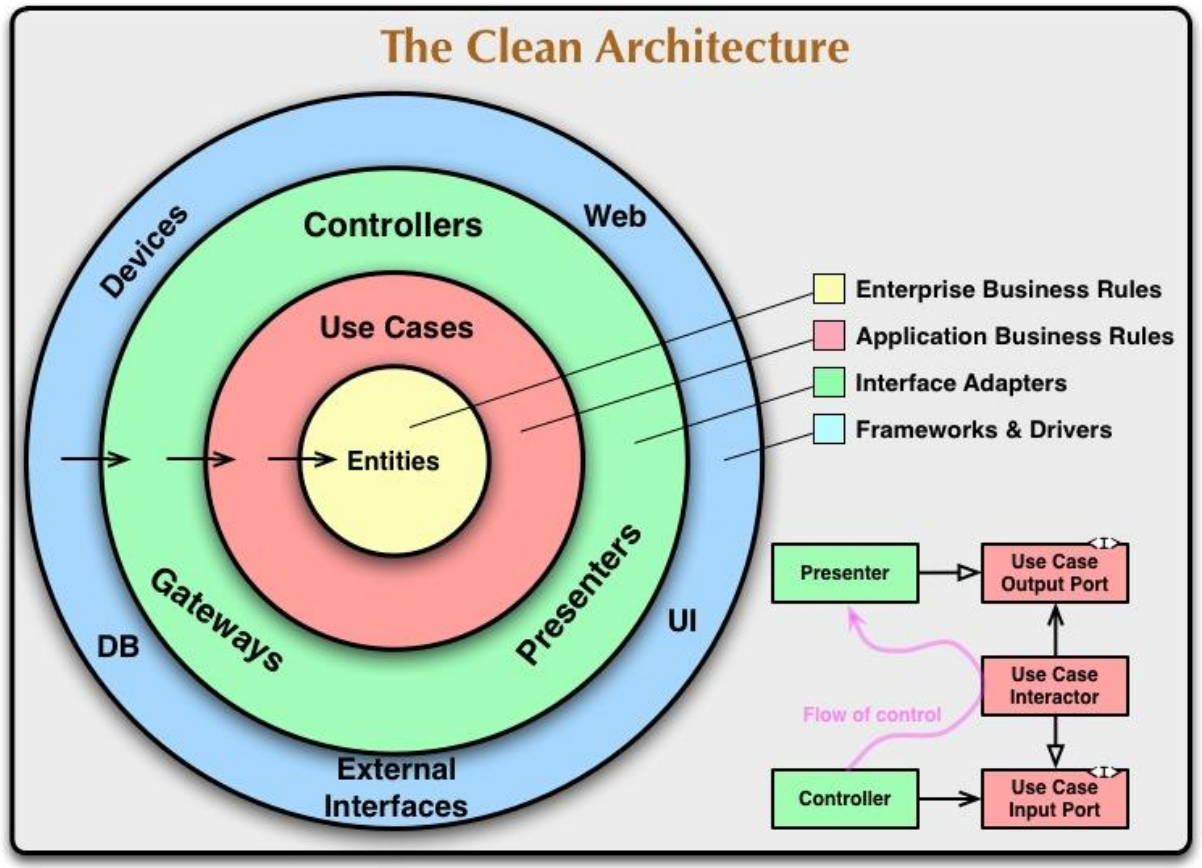


MVC, MVP, MVVM 패턴의 이해

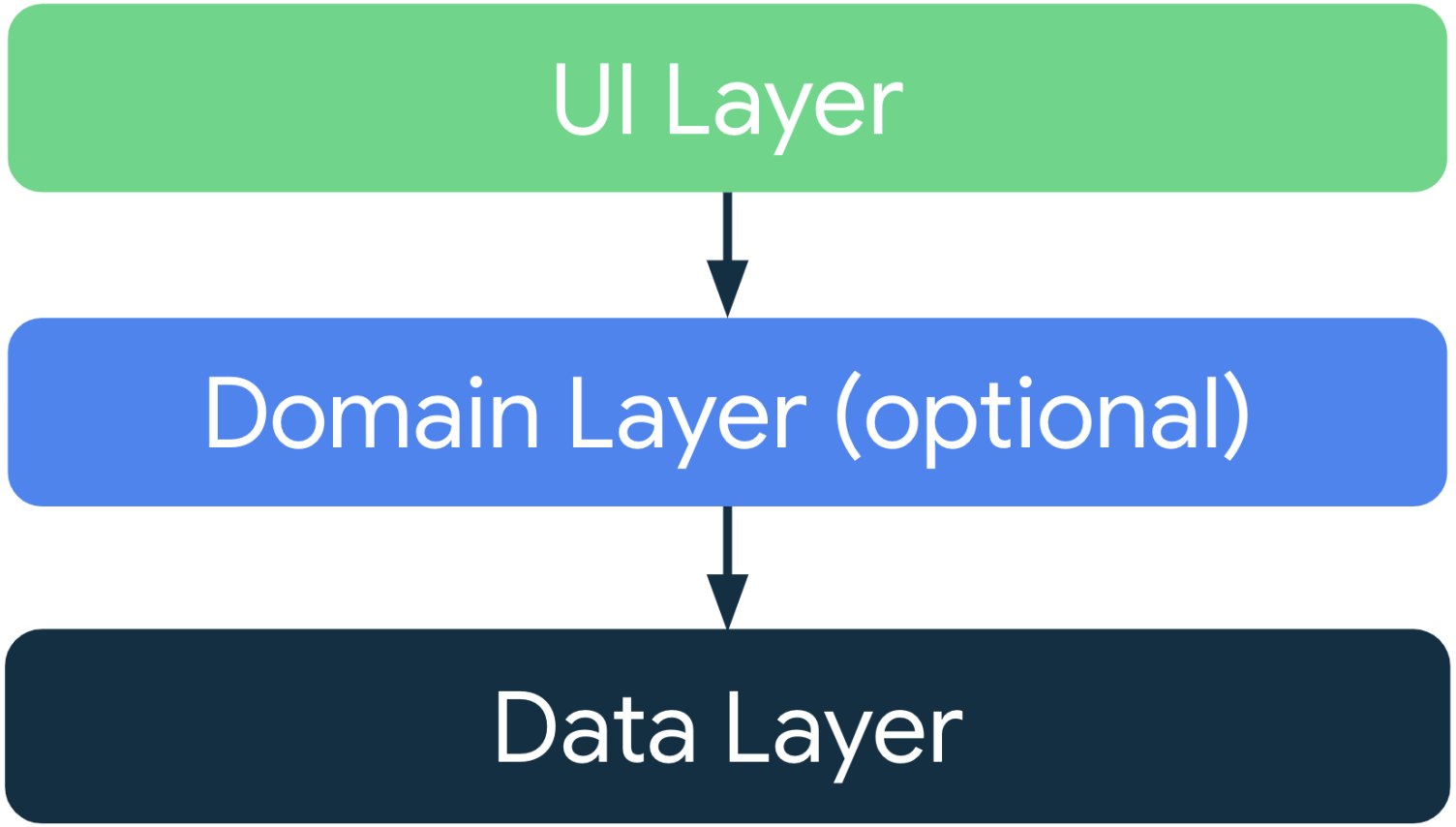
- Android에서도 ViewModel이라는 이름의 객체 제공
  - 이 둘은 이름만 같은 별개의 개념(AAC / MVVM ViewModel)
  - AAC ViewModel로도 MVVM 패턴 달성 가능

- MVVM의 의의는 View와 Model 간의 의존성을 낮추는 것!
  - 획득한 데이터를 표현하는 것에 집중하는 View
  - 데이터를 처리하는 것에 집중하는 Model
  - 데이터와 비즈니스 로직을 중간에서 관리하는 ViewModel

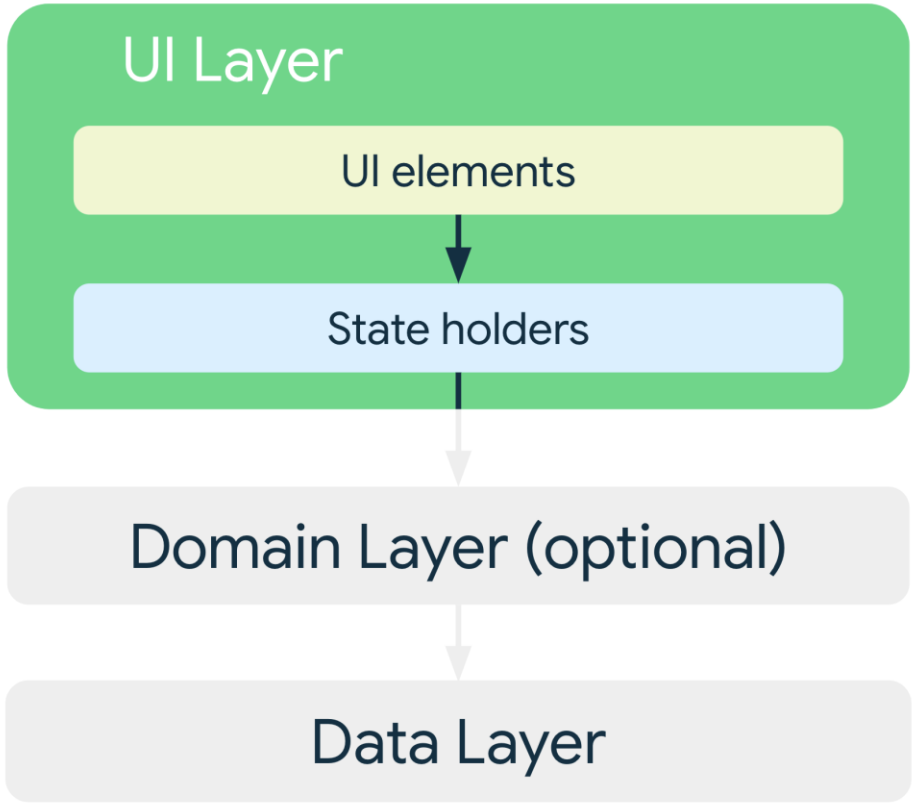
Clean Architecture

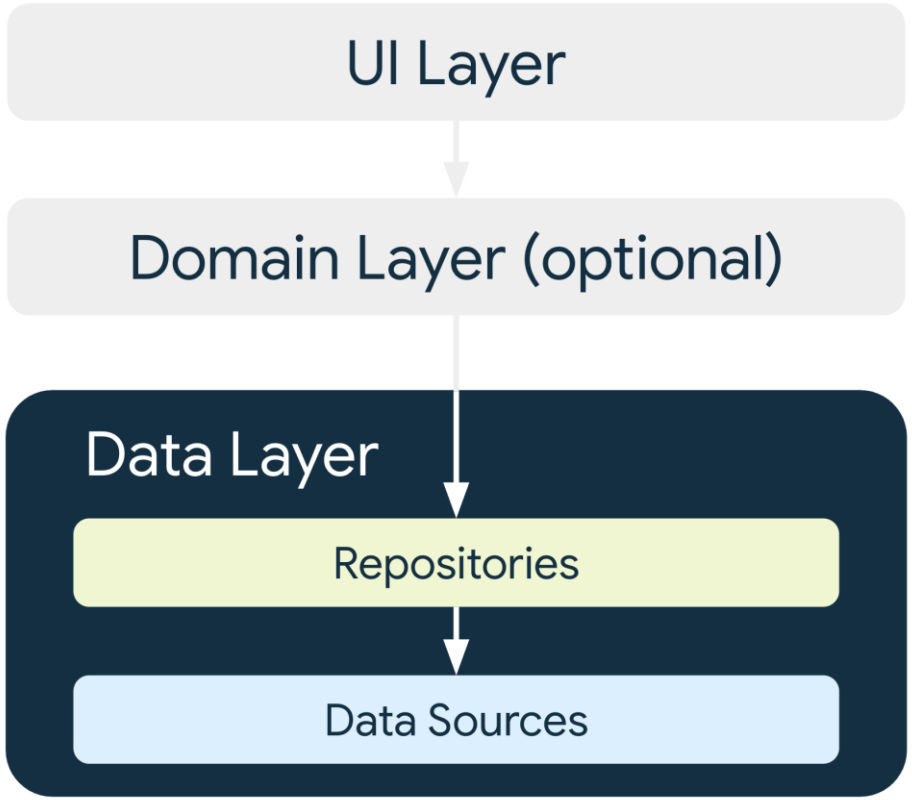


Google이 제시하는 Android 앱 권장 아키텍처

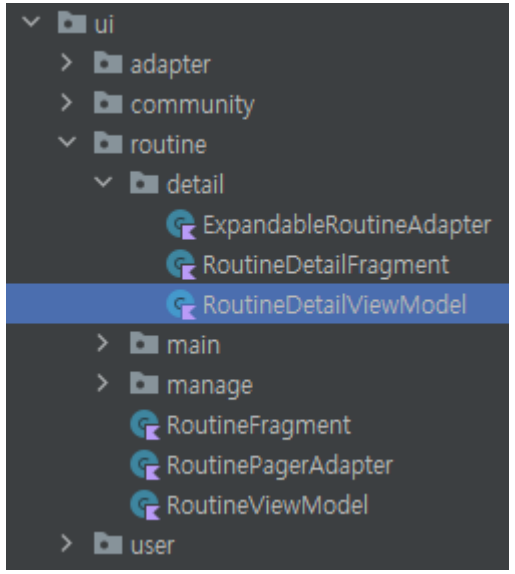


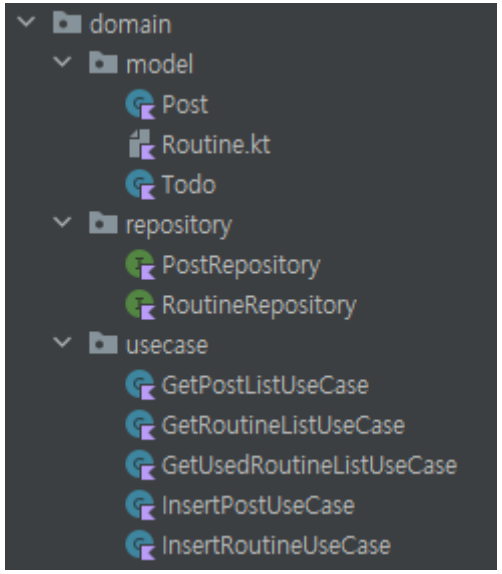
Google이 제시하는 Android 앱 권장 아키텍처

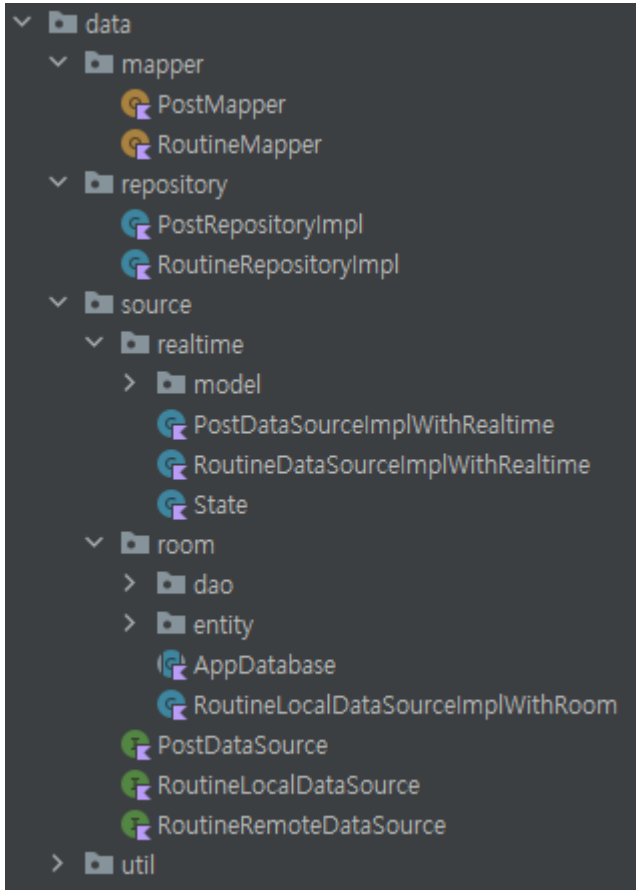




실제 적용 사례


ui
adapter
community
routine
detail
ExpandableRoutineAdapter
RoutineDetailFragment
RoutineDetailViewModel
main
manage
RoutineFragment
RoutinePagerAdapter
RoutineViewModel
user


domain
model
Post
Routine.kt
Todo
repository
PostRepository
RoutineRepository
usecase
GetPostListUseCase
GetRoutineListUseCase
GetUsedRoutineListUseCase
InsertPostUseCase
InsertRoutineUseCase


data
mapper
PostMapper
RoutineMapper
repository
PostRepositoryImpl
RoutineRepositoryImpl
source
realtime
model
PostDataSourceImplWithRealtime
RoutineDataSourceImplWithRealtime
State
room
dao
entity
AppDatabase
RoutineLocalDataSourceImplWithRoom
PostDataSource
RoutineLocalDataSource
RoutineRemoteDataSource
util

실제 적용 사례

```kotlin
interface RoutineLocalDataSource {
    suspend fun insert(routine: RoutineWithTodo)
    suspend fun update(routine: RoutineWithTodo)
    suspend fun delete(routine: RoutineWithTodo)

    fun getRoutineList(): Flow<List<RoutineWithTodo>>
}
```

```kotlin
interface RoutineRemoteDataSource {
    suspend fun insert(routineWithTodo: RealtimeDBModelRoutineWithTodo)
    suspend fun delete(routineWithTodo: RealtimeDBModelRoutineWithTodo)

    suspend fun fetchRoutine(id: String): State<RealtimeDBModelRoutineWithTodo>
}
```

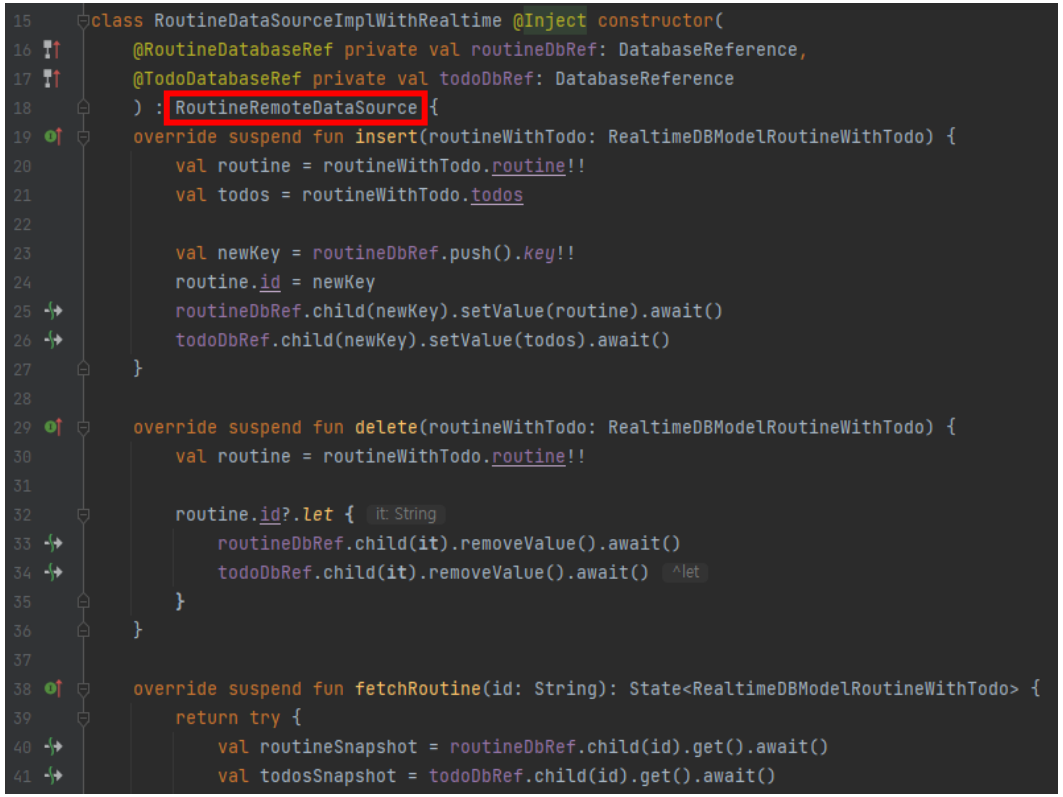
- 애플리케이션 외부에서 데이터를 얻어오는 역할
- 실제 데이터를 얻어오는 코드를 작성하여 이 인터페이스를 구현
- Repository 구현체에서 해당 인터페이스를 사용

실제 적용 사례

```
class RoutineLocalDataSourceImplWithRoom @Inject constructor(
    private val dao: RoutineDao
    ) : RoutineLocalDataSource {

    override suspend fun insert(routine: RoutineWithTodo) {
        dao.insert(routine)
    }

    override suspend fun update(routine: RoutineWithTodo) {
        dao.update(routine)
    }

    override suspend fun delete(routine: RoutineWithTodo) {
        dao.delete(routine)
    }

    override fun getRoutineList(): Flow<List<RoutineWithTodo>> {
        return dao.getRoutinesWithTodos()
    }
}
```

- 애플리케이션 외부에서 데이터를 얻어오는 역할
- 실제 데이터를 얻어오는 코드를 작성하여 이 인터페이스를 구현
- Repository 구현체에서 해당 인터페이스를 사용

실제 적용 사례



```
class RoutineDataSourceImplWithRealtime @Inject constructor(
    @RoutineDatabaseRef private val routineDbRef: DatabaseReference,
    @TodoDatabaseRef private val todoDbRef: DatabaseReference
    ) : RoutineRemoteDataSource {
    override suspend fun insert(routineWithTodo: RealtimeDBModelRoutineWithTodo) {
        val routine = routineWithTodo.routine!!
        val todos = routineWithTodo.todos

        val newKey = routineDbRef.push().key!!
        routine.id = newKey
        routineDbRef.child(newKey).setValue(routine).await()
        todoDbRef.child(newKey).setValue(todos).await()
    }

    override suspend fun delete(routineWithTodo: RealtimeDBModelRoutineWithTodo) {
        val routine = routineWithTodo.routine!!

        routine.id?.let { it: String
            routineDbRef.child(it).removeValue().await()
            todoDbRef.child(it).removeValue().await()  ^let
        }
    }

    override suspend fun fetchRoutine(id: String): State<RealtimeDBModelRoutineWithTodo> {
        return try {
            val routineSnapshot = routineDbRef.child(id).get().await()
            val todosSnapshot = todoDbRef.child(id).get().await()
```

- 애플리케이션 외부에서 데이터를 얻어오는 역할
- 실제 데이터를 얻어오는 코드를 작성하여 이 인터페이스를 구현
- Repository 구현체에서 해당 인터페이스를 사용

실제 적용 사례

```kotlin
interface RoutineRepository {
    suspend fun insert(routine: Routine)
    suspend fun deleteInLocal(routine: Routine)

    suspend fun upload(routine: Routine)
    suspend fun deleteInRemote(routine: Routine)

    fun getAllRoutinesFromLocal(): Flow<List<Routine>>

    suspend fun fetchRoutine(id: String): Routine
}
```

- DataSource로부터 획득해온 데이터를 **도메인 모델**로 변환
- 애플리케이션이 **비즈니스 로직**에 사용할 수 있는 형태로 가공
- 알맞은 Mapper 작성 필요
- 각 UseCase에 데이터 제공

실제 적용 사례

```
class RoutineRepositoryImpl @Inject constructor(
    private val localDataSource: RoutineLocalDataSource,
    private val remoteDataSource: RoutineRemoteDataSource
) : RoutineRepository {
    override suspend fun insert(routine: Routine) {
        localDataSource.insert(
            RoutineMapper.fromRoutineToRoutineWithTodo(routine)
        )
    }

    override suspend fun deleteInLocal(routine: Routine) {
        localDataSource.delete(
            RoutineMapper.fromRoutineToRoutineWithTodo(routine)
        )
    }

    override suspend fun upload(routine: Routine) {
        remoteDataSource.insert(
```

- DataSource로부터 획득해온 데이터를 **도메인 모델**로 변환
- 애플리케이션이 **비즈니스 로직**에 사용할 수 있는 형태로 가공
- 알맞은 Mapper 작성 필요
- 각 UseCase에 데이터 제공

실제 적용 사례

```kotlin
object RoutineMapper {
    fun fromRoutineWithTodoToRoutine(routineWithTodo: RoutineWithTodo): Routine {...}

    fun fromRoutineToRoutineWithTodo(routine: Routine): RoutineWithTodo {
        val todos = routine.todos.map { it: Todo
            RoomEntityTodo(
                dateTime = it.dateTime,
                importance = it.importance,
                description = it.description,
                achieved = it.achieved)
        }.toMutableList()

        val term = when (routine.term) {
            Term.DAILY -> 0
            Term.WEEKLY -> 1
            Term.MONTHLY -> 2
            Term.YEARLY -> 3
            Term.NONE -> 4
        }

        val roomEntityRoutine = RoomEntityRoutine(
            name = routine.name,
            term = term,
            isUsed = routine.isUsed)

        return RoutineWithTodo(roomEntityRoutine, todos)
    }
```

- DataSource로부터 획득해온 데이터를 **도메인 모델**로 변환
- 애플리케이션이 **비즈니스 로직**에 사용할 수 있는 형태로 가공
- 알맞은 Mapper 작성 필요
- 각 UseCase에 데이터 제공

실제 적용 사례

```
class GetUsedRoutineListUseCase @Inject constructor(
    private val repository: RoutineRepository
    ) {
    operator fun invoke(): Flow<List<Routine>> {
        return repository.getAllRoutinesFromLocal().map {  it: List<Routine>
            it.filter { item -> item.isUsed }
        }
    }
}
```

```
class InsertRoutineUseCase @Inject constructor(
    private val repository: RoutineRepository
    ) {
    suspend operator fun invoke(routine: Routine) {
        repository.insert(routine)
    }
}
```

- 작업 처리의 최소 단위
- ViewModel에서 사용하기 좋게 데이터를 가공하여 전달
- 구조만 보고도 어떤 서비스인지 알 수 있도록!

실제 적용 사례

```
18    @HiltViewModel
19    class RoutineDetailViewModel @Inject constructor(
20        getUsedRoutineListUseCase: GetUsedRoutineListUseCase,
21        private val insertRoutineUseCase: InsertRoutineUseCase,
22        @IoDispatcher private val ioDispatcher: CoroutineDispatcher
23    ) : ViewModel() {
24        private val _usedRoutines = getUsedRoutineListUseCase().onStart { emptyList<Routine>() }.asLiveData()
25        val usedRoutines: LiveData<List<Routine>> get() = _usedRoutines
26
27        fun insertNewRoutine(routine: Routine) {
28            viewModelScope.launch(ioDispatcher) {  this: CoroutineScope
29                insertRoutineUseCase(routine)
30            }
31        }
32    }
```

정리

- AAC ViewModel 사용
  - MVVM ViewModel을 사용했다고는 보기 힘듬
- Google이 제시하는 권장 아키텍처 적용
- 해당 아키텍처는 이른바 ‘클린 아키텍처’에 기반