OpenKnights App 개발 설계서
(Compose + MVI + SharedFlow 기반)

본 문서는 DroidKnightsApp의 구조를 기반으로 하여, "OpenKnightsApp"을 Clean Architecture와 MVI(Model-View-Intent) 패턴으로 재설계한 명세서입니다. 프로젝트는 공통 UI 시스템(core-designsystem)과 비즈니스 도메인(OpenKnights 대회 운영)을 중심으로 한 app 모듈(OpenKnightsApp)로 구성됩니다.

[핵심 개요]

1. Compose + MVI + Clean Architecture 구조

- UI 상태(UiState)와 이벤트(UiEffect)를 분리해 일관된 흐름 유지
- 각 Layer를 독립적으로 관리 → 테스트 용이, 유지보수성 증가

2. 멀티 모듈 구조

- `app-openknights` / `core-designsystem` / `core-model` / `core-domain` / `core-data`
- 공통 컴포넌트와 테마를 디자인 시스템으로 분리
- Firebase 기반 서버 연동 중심

3. 공통 UI 이벤트 처리 구조

- `SharedFlow<UiEffect>`를 통해 Toast, Navigation 등 처리
- ViewModel → Composable → UI 동작 → 초기화까지 일관된 흐름 제시

4. Firebase + (Room optional)

- Firebase Auth + Firestore + Storage 기본
- 필요시 Room 캐싱 구조 확장 가능

---

1. 프로젝트 구조 개요

- 앱 이름: OpenKnightsApp
- 주요 기능: 오픈소스 경진대회 참가팀 신청, 작품 소개, 평가/심사 흐름 안내
- 사용 기술: Kotlin, Jetpack Compose, Hilt, Firebase, Coroutine, Clean Architecture, MVI
- 참조 프로젝트: DroidKnightsApp

```
OpenKnightsProject/
├── app-openknights/       # 실제 앱 로직 및 화면 구성
├── app-admin/             # 운영자 전용 관리 앱 (선택사항)
├── core-designsystem/     # 공통 UI 시스템 (Theme, Typography, Components)
├── core-model/            # 공통 데이터 모델 정의
├── core-domain/           # 비즈니스 로직 (UseCase)
├── core-data/             # Repository + Firebase 연동
└── core-common/           # 공통 유틸, 에러 처리, UI 이벤트 관리 등
```

---

2. UI Design System (core-designsystem)

KnightsTheme (Theme.kt)

- `MaterialTheme`을 래핑하여 라이트/다크 모드 대응
- 내부에서 `LocalTypography`, `LocalShape` 등을 CompositionLocal로 제공

공통 정의 요소

- `Color.kt` → 브랜드 색상 정의 (primary, secondary, error 등)
- `Type.kt` → `KnightsTypography` 데이터클래스 정의
- `Shape.kt` → 카드, 버튼의 radius 등 일관된 모양 설정

🧩 공통 컴포넌트

- **`KnightsTopAppBar`**, **`KnightsCard`**, **`KnightsButton`**, **`KnightsDialog`** 등
- 내부적으로 **KnightsTheme**을 적용하고**,** 커스터마이징 가능한 **API** 제공

- **Buttons**: `KnightsPrimaryButton`, `KnightsOutlinedButton`
- **TopAppBar**: `KnightsTopAppBar(title: String)`
- **Card**: `TeamCard`, `JudgeCard`, `ProjectCard`
- **Dialog/Popup**: `ConfirmDialog`, `ToastPopup`

패키지 구조

UI Layer (app-openknights)

- `feature-team`, `feature-submit`, `feature-evaluation`, `feature-result` 등 각 기능별 화면 모듈로 구성
- `ViewModel`은 각 feature마다 존재하며 MVI 상태 관리 책임

Domain Layer (core-domain)

- 각 유스케이스 단위로 interface 정의
  - 예: `SubmitTeamUseCase`, `EvaluateSubmissionUseCase`
- 모든 유스케이스는 core-model의 데이터 구조를 사용하며, 외부 의존 없음

Data Layer (core-data)

- Firebase Firestore 기반 Repository 구현
- Domain의 UseCase가 의존하는 Repository 인터페이스를 구현
- 필요 시 Room(Local DB)은 선택적으로 도입 가능 (예: 캐시 저장)

---

3. 화면 상태 관리 구조 (MVI + SharedFlow)

📌 패턴 요약

- `UiState`: 화면 상태 (항상 유지)
- `UiEffect`: 일회성 이벤트 (Toast, Dialog, Navigation 등)
- `UiIntent`: 사용자 동작 입력

```
// UiState 예시
data class ApplicantListState(
    val isLoading: Boolean = false,
    val applicants: List<Team> = emptyList(),
    val selectedFilter: FilterType = FilterType.ALL
)

// UiEffect 예시
sealed interface ApplicantEffect {
    object ShowSubmitSuccessToast : ApplicantEffect
    data class NavigateToDetail(val teamId: String) : ApplicantEffect
}

// UiIntent 예시
sealed interface ApplicantIntent {
    object LoadApplicants : ApplicantIntent
    data class ClickTeam(val teamId: String) : ApplicantIntent
}
```

---

4. 예시 화면 구조 - 신청자 목록 (applicant)

✅ ViewModel

```
@HiltViewModel
class ApplicantViewModel @Inject constructor(
    private val getApplicantListUseCase: GetApplicantListUseCase
) : ViewModel() {
    private val _uiState = MutableStateFlow(ApplicantListState())
    val uiState = _uiState.asStateFlow()

    private val _uiEffect = MutableSharedFlow<ApplicantEffect>()
    val uiEffect = _uiEffect.asSharedFlow()

    fun onIntent(intent: ApplicantIntent) {
        when (intent) {
            is ApplicantIntent.LoadApplicants -> load()
            is ApplicantIntent.ClickTeam -> sendEffect(ApplicantEffect.NavigateToDetail(intent.teamId))
        }
    }

    private fun load() {
        viewModelScope.launch {
            _uiState.update { it.copy(isLoading = true) }
            val data = getApplicantListUseCase()
            _uiState.update { it.copy(applicants = data, isLoading = false) }
        }
    }

    private fun sendEffect(effect: ApplicantEffect) {
        viewModelScope.launch { _uiEffect.emit(effect) }
    }
}
```

✅ Composable

```
@Composable
fun ApplicantScreen(viewModel: ApplicantViewModel = hiltViewModel()) {
    val state by viewModel.uiState.collectAsStateWithLifecycle()
```

```
    val context = LocalContext.current

    LaunchedEffect(Unit) {
        viewModel.uiEffect.collect { effect ->
            when (effect) {
                is ApplicantEffect.ShowSubmitSuccessToast -> Toast.makeText(context, "신청 완료!",
Toast.LENGTH_SHORT).show()
                is ApplicantEffect.NavigateToDetail -> navigateToDetail(effect.teamId)
            }
        }
    }

    KnightsTopAppBar(title = "신청자 목록")
    LazyColumn {
        items(state.applicants) { team ->
            TeamCard(team = team, onClick = {
                viewModel.onIntent(ApplicantIntent.ClickTeam(team.id))
            })
        }
    }
}
```

---

5. Model & Domain 계층

📦 core-model

```
data class Team(
    val id: String,
    val name: String,
    val members: List<String>,
    val projectTitle: String,
    val submittedAt: LocalDateTime
)
```

⚙️ core-domain (UseCase)

```
interface GetApplicantListUseCase {
    suspend operator fun invoke(): List<Team>
}

class GetApplicantListUseCaseImpl @Inject constructor(
    private val applicantRepository: ApplicantRepository
) : GetApplicantListUseCase {
    override suspend fun invoke(): List<Team> = applicantRepository.fetchApplicants()
}
```

---

6. Data 계층 구조 (core-data)

📁 Repository

```
interface ApplicantRepository {
    suspend fun fetchApplicants(): List<Team>
    suspend fun submitApplicant(team: Team)
}
```

📁 실제 구현

```
class ApplicantRepositoryImpl @Inject constructor(
    private val api: ApplicantApi,
    private val dao: ApplicantDao
) : ApplicantRepository {
    override suspend fun fetchApplicants(): List<Team> = api.getApplicantList().map { it.toDomain() }
    override suspend fun submitApplicant(team: Team) = dao.insert(team.toEntity())
}
```

형식과 **UI Event** 처리 구조 **(MVI + SharedFlow)**

ViewModel 책임

- `UiState`: 현재 화면의 상태 (로딩, 데이터, 오류 등)
- `UiEffect`: 일회성 이벤트 (Toast, Alert, Navigation 등)

```
sealed interface OpenKnightsUiEffect {
    data class ShowToast(val message: String) : OpenKnightsUiEffect
    object NavigateToResult : OpenKnightsUiEffect
}

@HiltViewModel
class TeamSubmitViewModel @Inject constructor(...) : ViewModel() {
    val uiState = MutableStateFlow(TeamSubmitState())
    val uiEffect = MutableSharedFlow<OpenKnightsUiEffect>()

    fun onSubmitClicked() {
        viewModelScope.launch {
            repository.submitTeam(...)
            uiEffect.emit(OpenKnightsUiEffect.ShowToast("제출 완료!"))
            uiEffect.emit(OpenKnightsUiEffect.NavigateToResult)
        }
    }
}
```

Composable에서 구독

```
val uiEffect by viewModel.uiEffect.collectAsStateWithLifecycle()

LaunchedEffect(uiEffect) {
    when (val effect = uiEffect) {
        is OpenKnightsUiEffect.ShowToast -> showToast(context, effect.message)
        is OpenKnightsUiEffect.NavigateToResult -> navController.navigate("result")
    }
}
```

테스트 형식**: core-model**

```
data class Team(
    val id: String,
    val name: String,
    val members: List<Member>,
    val appIntro: String,
    val screenShotUrls: List<String>
```

```
)

data class SubmissionEvaluation(
    val creativityScore: Int,
    val practicalityScore: Int,
    val evaluator: String
)
```

✅ 정리: 재사용 가능한 핵심 구조

계층	역할	재사용 포인트
core-designsystem	테마 + 공통 UI	KnightsTheme, KnightsButton 등 모든 앱에서 사용 가능
core-model	순수 데이터	Team, Score, Judge 등 다른 앱에서도 재사용 가능
core-domain	비즈니스 로직	UseCase는 기능 단위로 쉽게 확장 및 교체 가능
core-data	API/DB 접근	Repository 패턴으로 구성 → 교체 용이
app-feature	화면 기능 단위	MVI 구조로 명확한 역할 분리, 테스트 용이

📌 권장 개발 방법 요약

- **Compose + MVI** 구조로 모든 화면 통일
- **UiState / UiEffect / UiIntent** 삼위일체 패턴으로 구조 고정
- 공통 UI는 `core-designsystem`에 모두 배치하고 `Knights` 접두사로 명명
- 모든 비즈니스 로직은 UseCase로 추상화하여 테스트 가능하게 작성
- Repository는 반드시 인터페이스로 분리해 의존성 역전 원칙 적용
- 데이터 계층은 API/DB 분리 및 Mapper 패턴 적용

Firebase 연동

- Firebase Firestore로 참가자/팀/심사정보 저장 및 조회
- Firebase Auth (이메일 기반) 로 참가자/운영자 권한 구분 가능
- Firebase Storage로 작품 스크린샷 업로드/관리

Local DB (Room)

- 네트워크 캐시, 오프라인 모드 등 필요 시 도입 가능
- Repository에서 local → remote fallback 구조 사용

확장 방안

- 평가 위원단 전용 평가 UI (`app-admin`)
- 참가자 Push 알림 (Firebase Messaging)
- 평가 통계 시각화 (`core-chart` 모듈 분리 가능)

이 설계서는 향후 OpenKnightsApp 개발에 있어 강의, 실습, 유지보수, 기능 확장 등 모든 면에서 일관성과 재사용성을 보장하는 기준이 될 수 있습니다.

```
// ViewModel
val uiState = MutableStateFlow<UiState>(...)      // 상태 유지용
val uiEvent = MutableSharedFlow<UiEvent>()        // 일회성 이벤트 (Toast, Alert 등)

fun onIntent(intent: UiIntent) {
    when (intent) {
        is UiIntent.Refresh -> refresh()
        is UiIntent.Click -> sendToast()
    }
}

private fun sendToast() {
    viewModelScope.launch {
        uiEvent.emit(UiEvent.ShowToast("Clicked"))
    }
}
```

```
// Composable
val state by viewModel.uiState.collectAsState()
LaunchedEffect(Unit) {
    viewModel.uiEvent.collect { event ->
        when (event) {
            is UiEvent.ShowToast -> Toast.makeText(context, event.message, LENGTH_SHORT).show()
        }
    }
}
```

우송 오픈소스 경진대회 모바일앱

앱이름: **OpenKnightsApp**

참조 github repo: DroidKnightsApp

주요 data

- 대회 신청자 -> 예선 -> 본선 -> 입상
- 신청자는 team에 속한다. team은 1명부터 5명까지 가능. team은 작품에 대한 기획-의도, 앱 소개, 앱 화면 예시 등을 작성해서 제출, 신청한다.
- OpenKnights 대회는 1년에 2회 개회, 예선 신청 기간, 예선 심사 단계, 본선 대회 준비 기간, 대회 당일 등의 순서로 진행된다. 예선 심사위원 3명, 본선 심사위원 4명으로 구성.
- 각 발표 작품/팀 별 평가 점수는 1) 창의성 부문, 2) 실용성 부문 으로 나뉘며, 창의성은 독창성과 완성도, 실용성은 시장성과 기술성 지표로 나뉘어 집계된다.

OpenKnightsApp은 공통 **UI design system**을 담당하는 **module**과**,** 오픈소스 경진대회 진행을 위한 **OpenKnights app**으로 나뉘어 개발된다**.**

(core-designsystem 내 공통 Theme의 이름은 'KnightsTheme' 으로 한다.)

2단계**:** 강의를 위한 프로젝트 구조 설계

추출된 테마와 공통 컴포넌트를 바탕으로 강의 프로젝트를 다음과 같이 구조화할 수 있습니다. 이는 "작은 회사의 UI 시스템"이라는 컨셉을 명확히 보여줍니다.

```
MyLectureProject/
├── app/                  # 20개의 UI 예제를 각각의 스크린으로 구현 (소비자)
│   ├── build.gradle.kts
│   └── src/main/
│       └── ...
│
└── core-designsystem/    # 공통 UI 시스템 모듈 (제공자)
    ├── build.gradle.kts
    └── src/main/
        └── java/com/example/core/designsystem/
            ├── theme/
            │   ├── Color.kt
            │   ├── Type.kt
            │   ├── Shape.kt
            │   └── Theme.kt      # 1단계에서 통합한 테마
            └── component/
                ├── Buttons.kt    # 1단계에서 추출한 버튼
                ├── Cards.kt      # 1단계에서 추출한 카드
                └── ...           # 기타 공통 컴포넌트
```

- **core-designsystem** 모듈**:** 우리 회사만의 UI 라이브러리 역할을 합니다. 모든 색상, 타이포그래피, 모양, 공통 컴포넌트가 여기에 위치합니다.
- **app** 모듈**:** core-designsystem 모듈에 의존(implementation(project(":core-designsystem")))합니다. 20개의 예제는 이 모듈에서 제공하는 테마와 컴포넌트를 가져와 사용(소비)합니다.

3단계**: "**디자인 시스템 구축**"** 스토리라인 강의 구성

이 구조를 바탕으로 강의를 다음과 같은 스토리라인으로 진행하면 학생들의 몰입도를 높일 수 있습니다.

주차	강의 주제	핵심 내용
1-2 주차	디자인 시스템의 기초**:** 우리 앱의 얼굴 만들기	- 왜 디자인 시스템이 필요한가**?** (일관성, 재사용성, 효율성) - core-designsystem 모듈 생성 - **Theme** 정의**:** 앱의 아이덴티티가 될 색상(Color), 글꼴(Typography), 모양(Shape) 시스템 구축
3-4 주차	가장 작은 부품 만들기 **(Atomic Design)**	- 재사용 가능한 기본 컴포넌트 제작 - **Buttons, TextFields, Icons** 등 가장 작은 단위의 UI 제작 - Gemini를 활용해 기존 코드에서 공통 컴포넌트 아이디어를 얻는 방법 시연
5-6 주차	부품 조립하기**:** 카드와 다이얼로그	- 기본 컴포넌트들을 조합하여 더 복잡한 컴포넌트 만들기 - **InfoCard, ProfileView, CustomDialog** 등 제작 - 상태(State)를 고려한 컴포넌트 설계

7-12 주차	실전**!** 디자인 시스템으로 앱 화면 만들기	- app 모듈에서 20개의 예제 화면을 본격적으로 구현 - 제작된 core-designsystem의 테마와 컴포넌트만을 사용하여 화면을 구성 - 학생들에게 직접 특정 화면을 만들어보게 하는 과제 제시
13주차	디자인 시스템 확장 및 유지보수	- 새로운 요구사항 발생 시 디자인 시스템을 어떻게 업데이트하는가? - (예: 새로운 버튼 스타일 추가, 테마 색상 변경) - 변경 사항이 앱 전체에 어떻게 일관되게 적용되는지 시연

결론

Gemini CLI 도구는 코드 분석 및 생성을 통해 초기 디자인 시스템의 기반을 빠르게 마련해주는 훌륭한 파트너가 될 수 있습니다. 이 도구를 활용하여 공통 테마와 컴포넌트를 추출하고, 이를 "디자인 시스템 모듈"로 분리하여 강의를 진행하신다면, 학생들은 단편적인 Compose 예제를 넘어 확장 가능하고 유지보수하기 좋은 **UI**를 설계하는 실무적인 경험을 얻게 될 것입니다. 이는 매우 효과적이고 기억에 남는 강의가 될 것입니다.

DroidKnights 2023 앱 분석 및 강의 목표로서의 적합성

이 프로젝트는 'DroidKnights'라는 기술 컨퍼런스를 위한 공식 앱으로, 실제 출시 및 운영을 염두에 두고 설계되었기 때문에 학습용 토이 프로젝트와는 깊이가 다릅니다.

1. 명확한 모듈화 **(Modularization)** 구조

이 프로젝트의 가장 큰 장점은 기능과 관심사에 따라 프로젝트가 명확하게 분리된 멀티 모듈 구조라는 점입니다. 이는 제가 이전에 제안드렸던 core-designsystem과 app 모듈 구조의 실제적인 상위 버전이며, 강의의 최종 목표를 보여주기에 완벽합니다.

```
DroidKnights2023_App/
├── app/              # 최종 결과물을 조립하는 모듈
├── core/
│   ├── designsystem/   # ✅ 강의의 핵심! 공통 디자인 시스템 모듈
│   ├── model/          # 데이터 모델 정의
│   └── data/           # 데이터 로직 (Repository)
└── feature/
    ├── session/        # '세션' 기능 모듈
    ├── contributor/    # '기여한 사람들' 기능 모듈
    └── ...             # 기타 기능 모듈들
```

- 강의 연계 방안**:**
  - 초반(1~6주차): core/designsystem 모듈을 집중적으로 구축합니다. 학생들이 직접 Color.kt, Type.kt, Component.kt 등을 채워나가게 합니다.
  - 중반(7~12주차): feature 모듈을 하나씩 만들어 나갑니다. 각 feature 모듈은 core/designsystem 모듈에 의존하여, 미리 만들어 둔 UI 컴포넌트를 가져와 화면을 조립하는 경험을 하게 합니다.
  - 후반(13주차): app 모듈에서 모든 feature 모듈들을 합쳐 하나의 완성된 앱으로 만드는 과정을 보여주며 프로젝트를 마무리합니다.

2. 체계적인 디자인 시스템 **(core/designsystem)**

프로젝트 내에 core-designsystem 모듈이 독립적으로 존재합니다. 이는 "우리 회사만의 UI 라이브러리"라는 컨셉을 학생들에게 직관적으로 보여줄 수 있는 최고의 구조입니다.

- 내부 구성 요소**:**
  - theme: Color, Typography, Shape 등 앱의 전체적인 룩앤필을 결정하는 요소들이 체계적으로 정의되어 있습니다. (Light/Dark 테마 포함)

    - component: KnightsTopAppBar, KnightsCard, KnightsButton 등 앱 전반에서 사용되는 커스텀 UI 컴포넌트들이 들어있습니다. 네이밍 규칙(Knights-)을 통해 앱 고유의 컴포넌트임을 명확히 한 점도 좋은 학습 포인트입니다.
- 강의 연계 방안:
    - "오늘은 우리 회사의 기본 버튼인 KnightsButton을 만들어 보겠습니다." 와 같이 역할극처럼 강의를 진행하며 학생들의 몰입을 유도할 수 있습니다.
    - 학생들이 20개의 예제에서 UI 패턴을 추출하고, Knights- 접두사를 붙여 자신만의 컴포넌트를 core/designsystem에 추가하고, 이를 feature 모듈에서 사용하는 과정을 직접 실습하게 합니다.

3. 현대적인 아키텍처와 기술 스택

이 프로젝트는 Android 개발의 최신 트렌드를 잘 따르고 있어, 학생들이 현업에서 사용되는 기술을 자연스럽게 습득하게 할 수 있습니다.

- 아키텍처: UI Layer(Compose) - Domain Layer - Data Layer로 나뉘는 **Clean Architecture**를 기반으로 한 MVI(Model-View-Intent) 패턴을 적용했습니다.
- 주요 기술 스택:
    - **100% Kotlin + Coroutines:** 비동기 처리
    - **Jetpack Compose:** UI 구현
    - **Hilt:** 의존성 주입 (DI)
    - **Retrofit + Moshi:** 네트워크 통신
    - **Github Actions:** CI/CD 자동화
- 강의 연계 방안:
    - Compose UI 강의가 메인이지만, "좋은 UI는 좋은 설계에서 나온다"는 점을 강조하며 왜 모듈을 나누고, 왜 Hilt 같은 도구를 사용하는지 자연스럽게 설명할 수 있습니다.
    - UI(View)와 데이터(ViewModel)를 분리하는 것의 중요성을 실제 코드를 통해 보여줄 수 있습니다.

최종 제안: 강의 로드맵

이 앱을 최종 목표로 설정하고, 다음과 같이 강의 로드맵을 구성하면 매우 효과적일 것입니다.

1. 시즌 **1:** 디자인 시스템 파운데이션 **(1~6**주차**)**
    - 목표: core-designsystem 모듈 완성하기
    - 활동: 색상/타이포그래피 정의, 기본 컴포넌트(버튼, 카드, 앱바 등) 제작, 테마(Theming) 적용
2. 시즌 **2:** 기능 단위 화면 개발 **(7~12**주차**)**
    - 목표: feature 모듈 2~3개 완성하기 (예: 세션 목록, 발표자 소개)
    - 활동: core-designsystem의 컴포넌트를 가져와 실제 화면을 조립. ViewModel을 연동하여 상태에 따라 UI가 바뀌는 동적인 화면 구현.
3. 시즌 **3:** 앱 완성 및 배포 준비 **(13~14**주차**)**
    - 목표: app 모듈에서 모든 기능 통합하기
    - 활동: Navigation을 이용해 화면 간 이동 구현, 최종 앱 빌드 및 완성.

이 DroidKnights 2023 앱은 기술적으로나 구조적으로나 매우 잘 만들어진 프로젝트입니다. 이 앱의 구조를 목표로 삼아 리버스 엔지니어링 하듯 하나씩 같이 만들어나가는 방식으로 강의를 구성하신다면, 학생들은 단편 지식을 넘어 하나의 완성도 있는 제품을 만드는 값진 경험을 하게 될 것입니다.

DroidKnights App UI Event 처리 아키텍처 분석

요구사항

- 공통 UI 컴포넌트(Alert, Toast 등)를 ViewModel → SharedFlow → Composable 패턴으로 처리하는 구조
- 여러 화면에서 공통 UI 컴포넌트를 사용하고, 각 화면의 View와 상호작용할 수 있는 공통된 메커니즘

분석 결과

DroidKnights 앱은 **ViewModel → StateFlow → Composable** 패턴을 사용하여 위 요구사항을 만족합니다. `StateFlow`는 `SharedFlow(replay=1)`의 특성을 가지므로 사실상 동일한 목적의 패턴으로 볼 수 있습니다.

이 구조는 UI의 상태를 나타내는 **`UiState`**와 Toast, Snackbar, Dialog와 같은 일회성 이벤트를 나타내는 **`UiEffect`**를 분리하여 관리합니다. 이를 통해 상태 관리 코드를 단순화하고 예측 가능하게 만듭니다.

핵심 구현 코드 (`feature/session` 모듈 예시)

1. ViewModel: UI Effect 이벤트 발생

ViewModel은 사용자의 액션에 따라 `UiEffect`를 `StateFlow`에 발행(emit)합니다.

`SessionDetailViewModel.kt`

```
@HiltViewModel
class SessionDetailViewModel @Inject constructor(...) : ViewModel() {
    // ... UiState 관련 코드 생략 ...
    // 1. 일회성 이벤트를 처리하기 위한 StateFlow 선언
    private val _sessionUiEffect = MutableStateFlow<SessionDetailEffect>(SessionDetailEffect.Idle)
    val sessionUiEffect = _sessionUiEffect.asStateFlow()

    // 2. 북마크 버튼 클릭 시 Toast 메시지(팝업)를 띄우라는 Effect 발생
    fun toggleBookmark() {
        // ...
        viewModelScope.launch {
            // ...
            _sessionUiEffect.value = SessionDetailEffect.ShowToastForBookmarkState(!bookmark)
        }
    }

    // 3. Composable에서 Effect 소비 후, 초기 상태로 되돌리기 위한 함수
    fun hidePopup() {
        viewModelScope.launch {
            _sessionUiEffect.value = SessionDetailEffect.Idle
        }
    }
}

// Effect 종류를 정의하는 Sealed Interface
sealed interface SessionDetailEffect {
    data object Idle : SessionDetailEffect
```

```
    data class ShowToastForBookmarkState(val bookmarked: Boolean) : SessionDetailEffect
}
```

2. Composable (Screen): UI Effect 구독 및 처리

Composable은 ViewModel의 `UiEffect` `StateFlow`를 구독하고, 특정 Effect가 발생했을 때 해당하는 UI(Toast, Popup 등)를 한 번만 보여줍니다.

SessionDetailScreen.kt

```
@Composable
internal fun SessionDetailScreen(
    sessionId: String,
    viewModel: SessionDetailViewModel = hiltViewModel(),
) {
    // ... UiState 구독 코드 생략 ...
    // 1. ViewModel의 UiEffect를 구독
    val effect by viewModel.sessionUiEffect.collectAsStateWithLifecycle()

    // ...
    Box {
        // ... 화면 컨텐츠 UI ...
        // 2. 특정 Effect가 발생했을 때만 해당하는 Composable을 띄움
        if (effect is SessionDetailEffect.ShowToastForBookmarkState) {
            SessionDetailBookmarkStatePopup(
                bookmarked = (effect as SessionDetailEffect.ShowToastForBookmarkState).bookmarked
            )
        }
    }

    // 3. Effect가 발생하면, 일정 시간 후 초기화하여 중복 표시를 방지
    LaunchedEffect(effect) {
        if (effect is SessionDetailEffect.ShowToastForBookmarkState) {
            delay(1000L) // 1초 동안 팝업을 보여줌
            viewModel.hidePopup() // ViewModel의 Effect를 초기 상태로 변경
        }
    }
}
```

결론 및 적용 방안

DroidKnights 앱의 `UiEffect` 패턴은 ViewModel과 View 간의 상호작용을 명확하게 분리하고, 일회성 UI 이벤트를 안정적으로 처리하는 훌륭한 예시입니다.

다른 앱에서도 이 구조를 적용하려면,

1. 각 화면의 ViewModel에 `UiState`와 `UiEffect`를 정의합니다.
2. 사용자 인터랙션에 따라 ViewModel에서 `UiEffect`를 발생시킵니다.
3. Composable에서는 `LaunchedEffect`를 사용하여 `UiEffect`를 구독하고, 해당하는 UI(Toast, Dialog 등)를 그리거나 화면 이동과 같은 Side Effect를 처리합니다.
4. Effect를 소비한 후에는 반드시 ViewModel의 `UiEffect`를 초기 상태로 되돌려야 합니다.

DroidKnights App 디자인 시스템 분석

개요

DroidKnights 앱은 `core/designsystem` 모듈을 통해 앱의 전체적인 디자인 통일성을 유지합니다. 이 모듈은 색상, 타이포그래피, Shape, 공통 컴포넌트를 중앙에서 관리하여 재사용성을 높이고 일관된 사용자 경험을 제공합니다.

핵심 구성 요소

1. 테마 (`Theme.kt`)

- **`KnightsTheme`**: Material3의 `MaterialTheme`을 기반으로 앱의 기본 테마를 정의합니다. `darkColorScheme`과 `lightColorScheme`을 별도로 정의하여 라이트/다크 모드를 완벽하게 지원합니다.
- **`CompositionLocalProvider`**: `LocalTypography`, `LocalShape` 등을 앱 전역에 제공하여 어떤 Composable에서든 일관된 디자인 요소를 쉽게 사용할 수 있도록 합니다.

```
// core/designsystem/src/main/java/com/droidknights/app/core/designsystem/theme/Theme.kt

@Composable
fun KnightsTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit,
) {
    val colorScheme = if (darkTheme) DarkColorScheme else LightColorScheme

    // ...
    CompositionLocalProvider(
        LocalDarkTheme provides darkTheme,
        LocalTypography provides Typography,
        LocalShape provides KnightsShape(),
    ) {
        MaterialTheme(
            colorScheme = colorScheme,
            content = content,
        )
    }
}

// 실제 사용 예시
object KnightsTheme {
    val typography: KnightsTypography
        @Composable
        get() = LocalTypography.current

    val shape: KnightsShape
        @Composable
        get() = LocalShape.current
}
```

2. 색상 (KnightsColor.kt)

- 앱에서 사용되는 모든 색상을 KnightsColor object에 상수로 정의하여 관리합니다.
- 이를 통해 색상의 일관성을 유지하고, 필요 시 손쉽게 전체 테마의 색상을 변경할 수 있습니다.

```
// core/designsystem/src/main/java/com/droidknights/app/core/designsystem/theme/KnightsColor.kt

object KnightsColor {
    @Stable
    val White = Color(0xFFFFFFFF)

    @Stable
    val Black = Color(0xFF000000)
    // ... 다양한 색상 정의 ...
}
```

3. 타이포그래피 (Type.kt)

- KnightsTypography 데이터 클래스를 사용하여 display, headline, title 등 다양한 텍스트 스타일을 체계적으로 정의합니다.
- 각 스타일에 맞는 fontSize, fontWeight, lineHeight 등을 미리 지정해두어 일관된 텍스트 위계를 제공합니다.

```
// core/designsystem/src/main/java/com/droidknights/app/core/designsystem/theme/Type.kt

internal val Typography = KnightsTypography(
    displayLargeR = SansSerifStyle.copy(
        fontSize = 57.sp,
        lineHeight = 64.sp,
    ),

    // ... 다양한 타이포그래피 스타일 정의 ...
)

@Immutable

data class KnightsTypography(
    val displayLargeR: TextStyle,
    // ...
)
```

4. 공통 컴포넌트 (component 패키지)

- KnightsCard, KnightsTopAppBar 등 앱 전반에서 재사용되는 UI 컴포넌트를 제공합니다.
- 이 컴포넌트들은 내부적으로 KnightsTheme의 색상과 타이포그래피를 사용하여 디자인 일관성을 유지합니다.

KnightsCard.kt

```
@Composable
fun KnightsCard(
    modifier: Modifier = Modifier,
    color: Color = MaterialTheme.colorScheme.surface, // 테마의 surface 색상 사용
    content: @Composable () -> Unit,
```

```
) {
    Surface(
        // ...
        color = color,
        shape = RoundedCornerShape(32.dp), // 일관된 Shape 적용
        content = content,
    )
}
```

KnightsTopAppBar.kt

```
@Composable
fun KnightsTopAppBar(
    // ...
) {

    // ...
    Text(
        text = stringResource(id = titleRes),
        style = KnightsTheme.typography.titleSmallM, // KnightsTheme의 타이포그래피 사용
        modifier = Modifier.align(Alignment.Center)
    )
    // ...
}
```

적용 방안

새로운 앱을 개발할 때 `core/designsystem` 모듈의 구조를 참고하여 다음과 같이 적용할 수 있습니다.

1. **`designsystem`** 모듈 생성: 앱의 핵심 디자인 요소를 관리할 별도의 모듈을 만듭니다.
2. 테마, 색상, 타이포그래피 정의: `Theme.kt`, `Color.kt`, `Type.kt` 파일을 생성하여 앱의 브랜드에 맞는 디자인 시스템을 구축합니다.
3. 공통 컴포넌트 개발: 버튼, 카드, 칩, 탑바 등 앱 전반에서 사용될 공통 컴포넌트를 `component` 패키지 내에 개발합니다. 이 때, 정의된 테마와 색상, 타이포그래피를 사용하도록 구현합니다.
4. **`KnightsTheme`** 적용: 모든 화면의 최상위 Composable을 `KnightsTheme`으로 감싸서 앱 전체에 일관된 디자인을 적용합니다.

DroidKnights App `core/model` 및 `core/domain` 모듈 분석

개요

DroidKnights 앱은 클린 아키텍처(Clean Architecture) 원칙에 따라 `core/model`과 `core/domain` 모듈을 분리하여 앱의 핵심 비즈니스 로직과 데이터 구조를 체계적으로 관리합니다.

1. `core/model` 모듈: 순수 데이터 구조 정의

역할

앱의 핵심 비즈니스 개념을 나타내는 순수한 데이터 구조(엔티티)를 정의합니다. 이들은 어떠한 비즈니스 로직이나 프레임워크 의존성도 가지지 않으며, 단순히 데이터를 담는 컨테이너 역할을 합니다.

특징

- `data class`로 구성되며, 앱의 모든 계층(UI, Domain, Data)에서 공유될 수 있는 가장 기본적인 데이터 형태입니다.
- 데이터의 형태와 속성만을 정의하며, 데이터를 조작하는 행위(메서드)는 포함하지 않습니다.

핵심 구현 코드 예시 (`model-session` 모듈)

`Session.kt`

```
package com.droidknights.app.core.model.session

import kotlinx.datetime.LocalDateTime

data class Session(
    val id: String,
    val title: String,
    val content: String,

    val speakers: List<Speaker>,
    val tags: List<Tag>,
    val room: Room,
    val startTime: LocalDateTime,
    val endTime: LocalDateTime,
    val isBookmarked: Boolean
)
```

`Speaker.kt`

```
package com.droidknights.app.core.model.session

data class Speaker(
    val name: String,
    val introduction: String,
    val imageUrl: String,
)
```

2. `core/domain` 모듈: 핵심 비즈니스 로직 정의 (Use Cases)

역할

앱의 핵심 비즈니스 규칙과 로직을 캡슐화합니다. `model`에서 정의된 엔티티를 사용하여 특정 비즈니스 목표를 달성하는 작업을 정의합니다. "어떤 작업을 수행할 것인가?"에 대한 질문에 답하는 계층입니다.

특징

- 유스케이스**(Use Case)**: 특정 비즈니스 시나리오를 구현하는 클래스입니다. 예를 들어, `BookmarkSessionUseCase`는 세션을 북마크하는 비즈니스 로직을 담당합니다.
- 프레임워크 독립적: 이 계층은 UI, 데이터베이스, 네트워크 등 외부 프레임워크에 의존하지 않습니다. 오직 `model` 계층과 `data` 계층의 인터페이스에만 의존합니다.
- **`api`** 모듈 분리: `domain-session-api`와 같이 `api` 모듈을 별도로 두어 유스케이스의 인터페이스를 정의하고, `domain-session`과 같이 `api`가 없는 모듈에서 해당 인터페이스의 구현체를 제공합니다. 이는 의존성 역전 원칙(Dependency Inversion Principle)을 따르며, 상위 모듈(예: `feature` 모듈)이 하위 모듈(예: `data` 모듈)의 구체적인 구현에 의존하지 않고 인터페이스에만 의존하도록 하여 결합도를 낮춥니다.

핵심 구현 코드 예시 (`domain-session` 모듈)

`BookmarkSessionUseCase.kt` (API 인터페이스)

```
package com.droidknights.app.core.domain.session.usecase.api
interface BookmarkSessionUseCase {
    suspend operator fun invoke(sessionId: String, bookmark: Boolean)
}
```

`BookmarkSessionUseCaseImpl.kt` (구현체)

```
package com.droidknights.app.core.domain.session.usecase
import com.droidknights.app.core.data.session.api.SessionRepository
import com.droidknights.app.core.domain.session.usecase.api.BookmarkSessionUseCase
import javax.inject.Inject

internal class BookmarkSessionUseCaseImpl @Inject constructor(
    private val sessionRepository: SessionRepository,
) : BookmarkSessionUseCase {
    override suspend operator fun invoke(sessionId: String, bookmark: Boolean) =
        sessionRepository.bookmarkSession(sessionId, bookmark, )
}
```

3. `domain`과 `model`이 분리된 이유

`domain`과 `model` 모듈이 분리된 것은 클린 아키텍처(Clean Architecture) 또는 계층형 아키텍처(Layered Architecture)의 원칙을 따르기 위함입니다.

1. 관심사 분리 **(Separation of Concerns)**:
    - `model`은 "무엇(What)"을 다룰 것인지(데이터 구조)에 집중합니다.
    - `domain`은 "어떻게(How)" 비즈니스 로직을 수행할 것인지에 집중합니다.
    - 이렇게 분리함으로써 각 계층의 책임이 명확해지고, 코드의 가독성과 유지보수성이 향상됩니다.

2. 재사용성:
    - `model`은 앱의 모든 계층에서 재사용될 수 있는 순수한 데이터 구조를 제공합니다.
    - `domain`의 유스케이스는 UI나 데이터 소스가 변경되어도 핵심 비즈니스 로직은 변하지 않으므로, 다른 플랫폼이나 UI 프레임워크에서도 재사용될 수 있습니다.

3. 테스트 용이성:
    - `model`은 단순한 데이터 클래스이므로 쉽게 테스트할 수 있습니다.

- `domain`의 유스케이스는 외부 의존성(데이터베이스, 네트워크 등)을 인터페이스로 추상화하므로, 단위 테스트 시 Mock 객체를 사용하여 쉽게 테스트할 수 있습니다.

4. 유연성 및 확장성:

   - 데이터 소스(예: 로컬 DB, 원격 API)가 변경되더라도 `domain` 계층은 `data` 계층의 인터페이스에만 의존하므로, `data` 계층의 구현만 변경하면 됩니다. `domain` 계층은 영향을 받지 않습니다.
   - 새로운 비즈니스 로직이 추가될 때 `domain` 계층에 새로운 유스케이스를 추가하면 됩니다.

결론적으로, `domain`과 `model`의 분리는 앱의 아키텍처를 견고하고 유연하게 만들며, 대규모 앱 개발 및 장기적인 유지보수에 매우 유리합니다. 발표자(contributor)와 발표 내용(session)이 중요한 데이터 모델인 것은 맞지만, 이 모델들을 가지고 어떤 행위를 할 것인지(예: 세션 북마크, 발표자 목록 가져오기)는 `domain` 계층에서 정의하고 구현하는 것이 바람직합니다.

DroidKnights App `core/data` 모듈 분석

개요

`core/data` 모듈은 앱의 데이터 계층을 담당하며, `domain` 계층에서 정의된 유스케이스가 필요로 하는 데이터를 제공합니다. 이 모듈은 데이터 소스(네트워크, 로컬 데이터베이스, 파일 등)로부터 데이터를 가져오고, 저장하며, 조작하는 역할을 합니다.

핵심 구성 요소

1. Repository 패턴

- `SessionRepository`와 같은 인터페이스를 통해 데이터 소스에 대한 추상화를 제공합니다. `domain` 계층은 이 Repository 인터페이스에만 의존하므로, 실제 데이터 소스가 변경되어도 `domain` 계층은 영향을 받지 않습니다.
- `api` 모듈(`data-xxx-api`)에 Repository 인터페이스를 정의하고, 실제 구현체는 `data-xxx` 모듈에 위치시켜 의존성 역전 원칙을 따릅니다.

2. 데이터 소스 캡슐화

- `SessionApi`, `SessionPreferencesDataSource`와 같이 실제 데이터 소스와 상호작용하는 구체적인 구현을 캡슐화합니다. `domain` 계층은 이들의 존재를 알 필요가 없습니다.

3. 데이터 변환 (Mapper)

- 필요한 경우, 네트워크나 로컬 데이터베이스에서 가져온 데이터를 `core/model`에서 정의된 도메인 모델로 변환하는 로직을 포함할 수 있습니다. (예: `it.toData()` 호출)

핵심 구현 코드 예시 (`data-session` 모듈)

`SessionRepository.kt` (API 인터페이스)

```
package com.droidknights.app.core.data.session.api
import com.droidknights.app.core.model.session.Session
```

```
import kotlinx.coroutines.flow.Flow

interface SessionRepository {
    suspend fun getSessions(): List<Session>
    suspend fun getSession(sessionId: String): Session
    fun getBookmarkedSessionIds(): Flow<Set<String>>
    suspend fun bookmarkSession(sessionId: String, bookmark: Boolean)
    suspend fun deleteBookmarkedSessions(sessionIds: Set<String>)
}
```

SessionRepositoryImpl.kt (구현체)

```
package com.droidknights.app.core.data.session
import com.droidknights.app.config.api.DroidknightsBuildConfig
import com.droidknights.app.core.data.session.api.SessionApi
import com.droidknights.app.core.data.session.api.SessionRepository
import com.droidknights.app.core.data.session.mapper.toData
import com.droidknights.app.core.datastore.session.api.SessionPreferencesDataSource
import com.droidknights.app.core.model.session.Session
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.filterNotNull
import kotlinx.coroutines.flow.first
import javax.inject.Inject

internal class SessionRepositoryImpl @Inject constructor(
    private val sessionApi: SessionApi,
    private val sessionDataSource: SessionPreferencesDataSource,
    private val droidknightsBuildConfig: DroidknightsBuildConfig,
) : SessionRepository {
    private var cachedSessions: List<Session> = emptyList()
    private val bookmarkIds: Flow<Set<String>> = sessionDataSource.bookmarkedSession
    override suspend fun getSessions(): List<Session> {
        return sessionApi.getSessions(url = droidknightsBuildConfig.sessionsDataUrl())
            .map { it.toData() }
            .also { cachedSessions = it }
    }

    override suspend fun getSession(sessionId: String): Session {
        val cachedSession = cachedSessions.find { it.id == sessionId }
        if (cachedSession != null) {
            return cachedSession
        }
        return getSessions().find { it.id == sessionId }
            ?: error("Session not found with id: $sessionId")
    }
    override fun getBookmarkedSessionIds(): Flow<Set<String>> {
        return bookmarkIds.filterNotNull()
    }
    override suspend fun bookmarkSession(sessionId: String, bookmark: Boolean) {
        val currentBookmarkedSessionIds = bookmarkIds.first()
        sessionDataSource.updateBookmarkedSession(
            if (bookmark) {
                currentBookmarkedSessionIds + sessionId
            } else {
```

```
            currentBookmarkedSessionIds - sessionId
        }
    )
  }
  override suspend fun deleteBookmarkedSessions(sessionIds: Set<String>) {
    val currentBookmarkedSessionIds = bookmarkIds.first()
    sessionDataSource.updateBookmarkedSession(
        currentBookmarkedSessionIds - sessionIds
    )
  }
}
```

적용 방안

새로운 앱을 개발할 때 `core/data` 모듈의 구조를 참고하여 다음과 같이 적용할 수 있습니다.

1. **`data`** 모듈 생성: 앱의 데이터 접근 로직을 관리할 별도의 모듈을 만듭니다.
2. **Repository** 인터페이스 정의: 각 도메인(예: 세션, 발표자)에 대한 Repository 인터페이스를 `data-xxx-api` 모듈에 정의합니다. 이 인터페이스는 `domain` 계층에서 사용될 것입니다.
3. **Repository** 구현체 개발: `data-xxx` 모듈에 Repository 인터페이스의 구현체를 개발합니다. 이 구현체는 실제 네트워크 통신, 로컬 데이터베이스 접근, 파일 읽기/쓰기 등 데이터 소스와의 상호작용 로직을 포함합니다.
4. 데이터 변환: 필요한 경우, 데이터 소스에서 가져온 데이터를 `core/model`에서 정의된 도메인 모델로 변환하는 매퍼(Mapper)를 구현합니다.