

멀티 모듈 (Modular Architecture)

강사 룡

앱의 규모가 커지면, 필연적으로 기능별 / 계층별로 모듈을 분리해야할 필요가 생깁니다. 멀티 모듈로의 분리를 통해서 우리는 Android 프로젝트를 더 아키텍처 기반으로 만들고 아울러 빌드 속도도 크게 개선할 수 있습니다. 이번 강의에서는 구체적으로 모듈은 어떻게 분리하면 좋은가, 모듈화 도중에 겪는 문제는 어떻게 해결해야 하는가를, 여러 팀들을 컨설팅 해 온 경험을 토대로 해설합니다.

TABLE OF CONTENT.

1

모듈을 나누기 위한
기본 지식

2

모듈화 과정에서
겪는 문제들

모듈을 나누기 위한 기본 지식.

- 모듈이란 무엇인가?
- 모듈은 왜 나뉘야 하는가?
- 모듈을 어떻게 나눌 것인가?
- 모듈은 언제 나뉘야 하는가?
- 모듈을 나누는 절차
- 모듈화의 중요 원칙



모듈이란 무엇인가?

Module (n.)

A module is a chunk of code that has a well defined scope, logically justified dependencies, a clear owner, and is meant to be reusable across several of apps.

잘 정의된 범위, 논리적으로 타당한 의존성들, 분명한 담당자를 갖고 있으며, 앱들에서 재사용되도록 의도된 코드의 덩어리.

- 예: Account 모듈 - 로그인/아웃, 사용자 아이덴티티 선택, 인증 관련 코드가 모여있는 곳
- 여기서 얘기하는 모듈은 Dagger module이 아닌 gradle / bazel module을 의미
- 모듈은 앱을 구성하는 벽돌들과 같이 사용됨

모듈은 왜 나눠야 하는가?

프로젝트 코드베이스의 복잡도를 낮게 유지 → 프로젝트가 커지고 팀원이 늘어나도 생산성 저하를 방지할 수 있으므로

모듈화를 통해서 내부 코드들 사이의 의존도를 낮출 수 있음 → 코드 복잡도를 낮추고, 테스트 친화성을 높임

개발자의 자율성 증대
(Contributor autonomy)

- 자기 모듈들의 ownership을 갖고 담당할 수 있으므로
- 외부 모듈의 영향으로부터 잘 격리된 코드를 이용해서 독립적으로 일할 수 있으므로

빌드 속도!
- 개발자 행복 증진

- 병렬 처리로 인한 빌드 속도 증가 (특히 incremental build 속도)
- 매우 처리가 무거운 일부 gradle plugin의 영향을 특정 모듈로 국한 시킬 수 있음 (eg. Realm)
- 각 개발자가 보고 있는 모듈 이외의 타 모듈을 dummy로 바꿔서 빌드 속도 개선 가능

Play Feature Delivery
(구 Dynamic Feature Module)
- 사용자 행복 증진

- 중요성이 있지만 널리 사용되지는 않는 기능이라면 필요할 때만 다운로드, 필요 없을 때는 언인스톨 가능
- 앱의 크기와 시작 시간을 함께 줄일 수 있음

모듈을 어떻게 나눌 것인가?

중요한 질문들

- ✓ 이 모듈의 역할은 무엇인가?
 - 어떤 서비스 / 기능을 제공하는가?
 - 다른 모듈이 어떻게 사용해야 하는가?
- ✓ 이 모듈은 복잡성을 캡슐화 하고 있는가?
- ✓ 이 코드들은 모듈 하나로 묶으면 충분한가? 아니면 좀 더 나뉘 여지가 있는가?
- ✓ 이 코드들은 여러 모듈들이어야 하는가? 아니면 조합이 가능한가?

모듈을 어떻게 나눌 것인가?



캡슐화

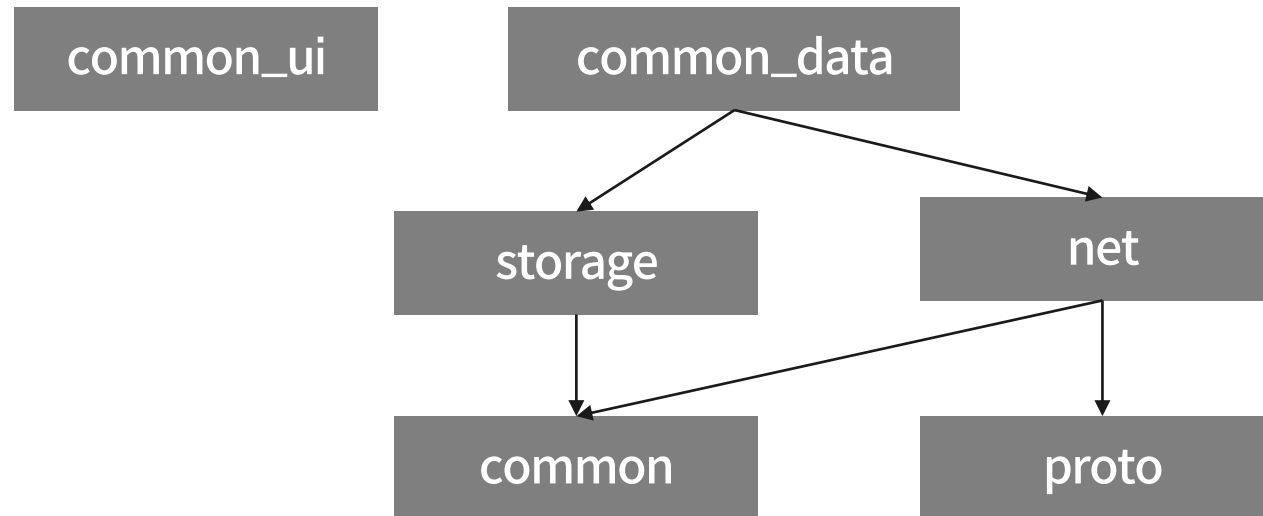
만약 모듈 A가 모듈 B의 내부적인 동작에 강하게 의존하고 있다면,
A를 B의 복잡성으로부터 격리하지 못한 것.
따라서,

- 모듈 B의 API가 더 잘 정의될 필요가 있거나
- 모듈 B는 A의 일부가 되어야 함

모듈을 어떻게 나눌 것인가?

“

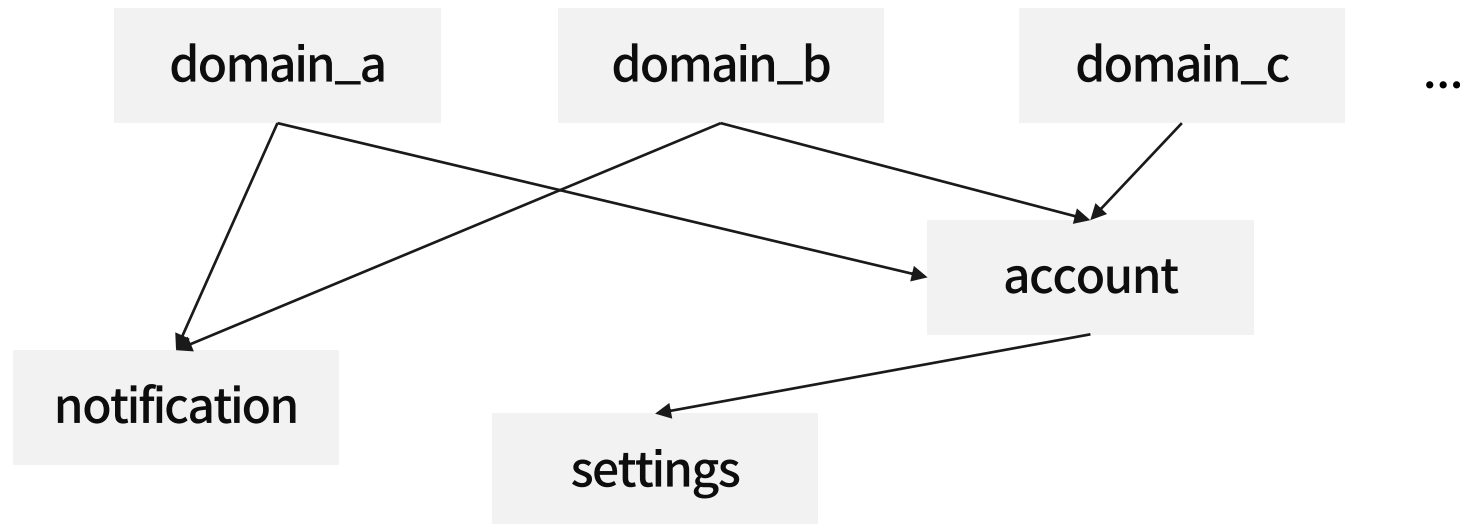
재사용성 높은 기능에 따른 분리



모듈을 어떻게 나눌 것인가?

“

도메인 지식에 따른 분리




여기서 퀴즈!

“

이 화면에는 몇 개의 모듈이
사용되고 있을까요?





PUBG Mobile

KRAFTON, Inc.
In-app purchases



4.3★
1M reviews ⓘ

10M+
Downloads

16+
Rated for
16+ ⓘ

Install

ⓘ Google Play Services will install additional components (11 MB) needed to use this app




About this game →

4th Anniversary 4U!

- ✓ 각 모듈에 다른 owner가 지정되어 있음
- ✓ 각자의 모듈이 다른 종류의 데이터를 취급
- ✓ 각자의 모듈이 다른 의존성을 가짐
- ✓ 사용자 인터렉션의 종류도 다름





PUBG Mobile

KRAFTON, Inc.

In-app purchases



4.3 ★
1M reviews ⓘ

10M+
Downloads

16+
Rated for
16+ ⓘ

Install

ⓘ Google Play Services will install additional components (11 MB) needed to use this app

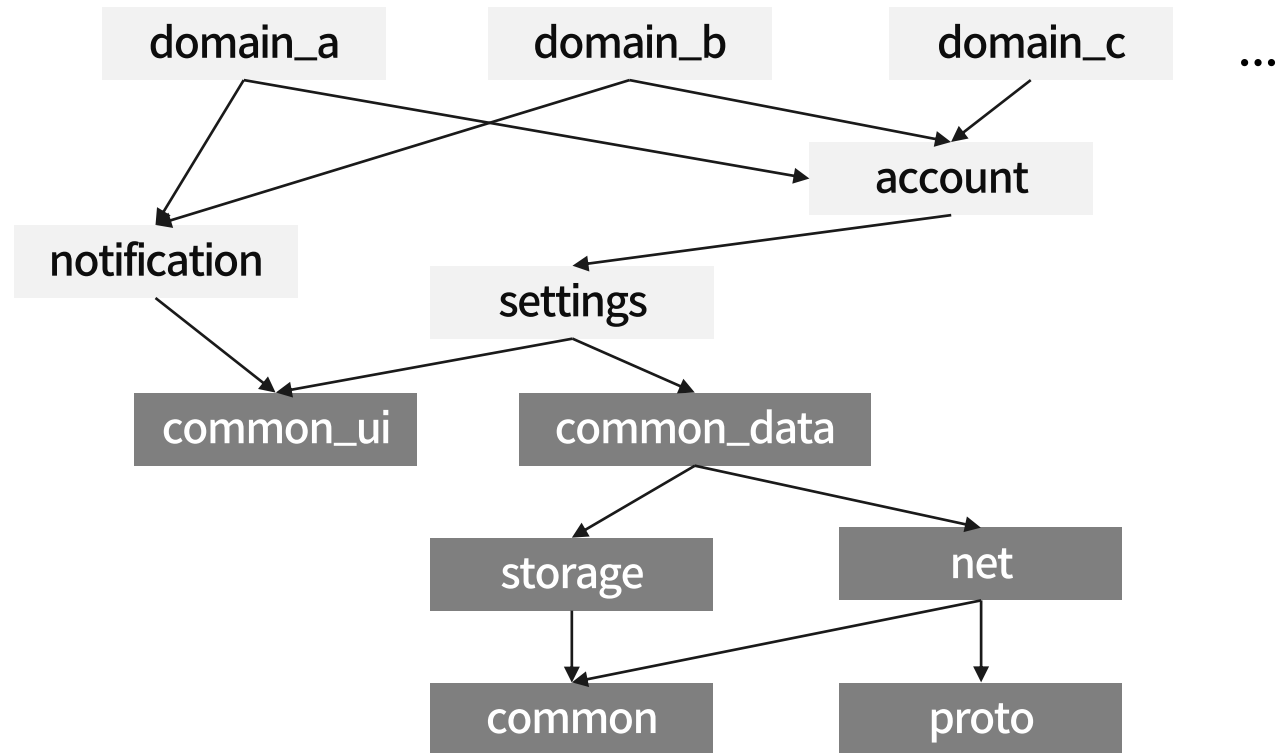



About this game

4th Anniversary 4U!

“

합치면?



여러 앱을 컨설팅해오면서 깨달은 것들

- ✓ 기존에 이미 분류해 놓은 package 구조를 그대로 모듈로 옮기는 건 보통 결과가 좋지 못함
- ✓ 모듈을 단순히 피처나 팀에 따라서 나누는 것은 이상적인 모듈 구조를 만들지 못한다.
 - 대신, 각 모듈은 코드베이스 내에서의 "시스템"을 대변해야 한다
 - 피처 코드는 그것이 상호작용하는 여러 시스템에 걸쳐서 적절히 흩어져 있어야 한다
- ✓ 하나의 모듈이 너무 많은 역할을 하고 있다면, 더 분리해야 한다는 신호일 수 있음
- ✓ 하나의 모듈이 오직 하나의 다른 플레이어에 의해서 사용된다면 모듈로 분리시킬 필요가 없다는 증거일 수 있음
- ✓ 모듈의 사이즈는 모듈 분류의 기준이 될 수 없음

모듈을 나누는 기준 - 총정리!

| | |
|-----------------------------------|---|
| 도메인 지식 (domain knowledge) | 하나의 모듈은 그룹화가 될만큼 명확한 분량의 도메인 지식을 캡슐화 해야 한다 |
| 잘 정의된 범위 (well-defined scope) | 좋은 범위는 새로운 기능 조각이 하나의 모듈에 속할지 그렇지 않을지를 애매모호하지 않게 해줘야 한다 |
| 재사용성 (reusability) | 모듈이 제공하는 기능은 여러 종류의 앱에서 사용될 수 있어야 한다 |
| 선택가능성 (optionality) | 모듈이 제공하는 기능은 특정 앱에서 활성화 될 수도 있고 아닐 수도 있다 |
| 의존성 관리 (dependency management) | 새로운 모듈은 기존 모듈과 확연히 다른 의존성 요구사항을 갖는다 |
| 책임조직 (accountability) | 장기간 이 모듈을 책임질 의지가 있는 팀이 존재한다 |

그럼 모듈은 언제 나눠야 하는가?

“

ASAP

- ✓ 모듈을 나누는 데에 드는 노력이 크지 않음
- ✓ 모듈화는 늦을 수록 painful 해진다
- ✓ 모듈화를 통해서 이전에 발견하지 못했던 아키텍처 상의 문제를 찾아서 해결할 수 있고, 잘 캡슐화된 구조를 유지할 수 있다
- ✓ 모듈화를 통해서 더 테스트하기 용이한 구조를 유지할 수 있다

모듈을 나누는 절차

1. 모듈로 나눌 수 있을 듯한 후보 코드들을 정한다
2. 해당 코드들의 의존성 이슈를 해결한다
 - a. 불필요한 의존성은 없애거나, 다른 곳으로 이전
 - b. 만약 아직 모듈화 되어 있지 않은 의존성이 발견 됐다면, 중단 후 일단 해당 부분을 먼저 모듈화
3. 목표 모듈이 interface / impl module 두 개로 나뉘어야 하는지 결정
4. 모듈을 위한 DI 설정 작성
5. 어느 정도 커버리지를 만족할 때까지 단위 테스트 추가
6. README.md 파일을 작성하고 담당자를 명시
7. 모듈을 작성하고 모든 코드를 이동!

모듈화의 중요 원칙



**최소한의 테스트 커버리지를 유지할 것!
그리고 그 책임은 모듈 담당자에게..**

- ✓ 테스트를 기본적으로 담보해야, 모듈의 API들이 외부 모듈에서 문제없이 사용할 수 있다는 확신을 줄 수 있음
- ✓ 담당자 이외의 개발자가 모듈을 수정할 때도, 원래 의도와 다른 방향으로 잘못 수정하는 일을 막을 수 있다

모듈화 과정에서 겪는 문제들.

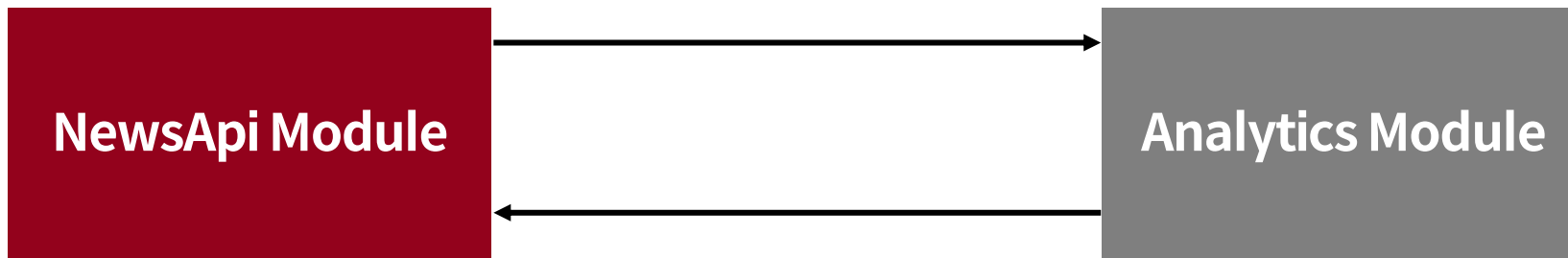
- 문제점1 순환 참조
- 문제점2 거꾸로 된 의존성
- 의존성 역전의 원칙
- 멀티모듈을 위한 언어 / 빌드툴 차원의 지원
- 문제점3 없어야 할 의존성이 존재
- 문제점4 god object에 대한 의존성
- 문제점5 너무 많은 / 큰 유틸리티 클래스들
- 문제점6 정적인 유틸리티 함수들
- 문제점7 부수효과로 인한 테스트의 어려움
- 리소스의 모듈화



문제점

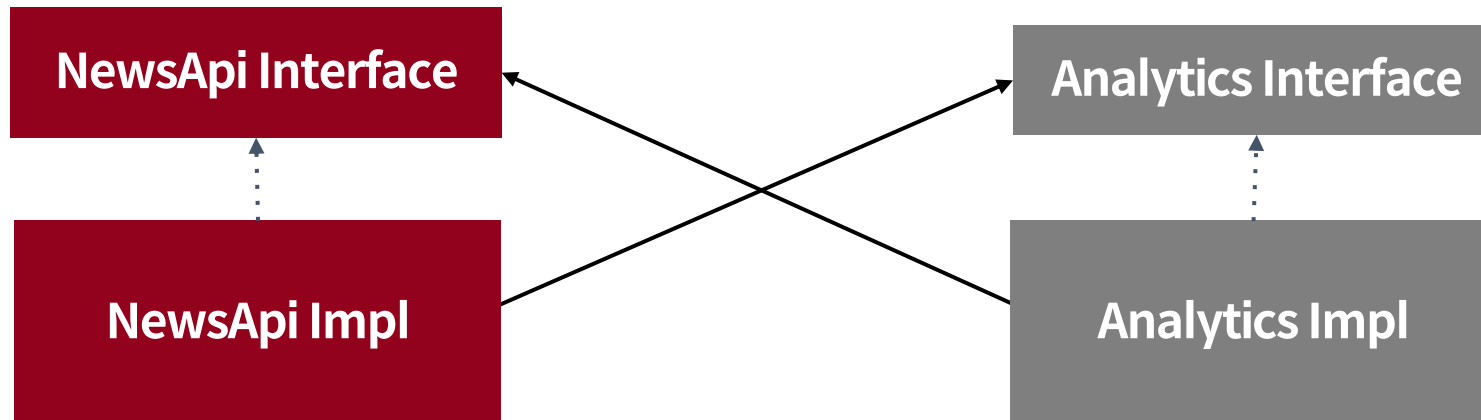
: 순환 참조 (Circular Dependencies)

- gradle도, bazel도 모듈이 서로를 참조할 수 없음
- 그러나.. 처음 모듈을 분리하면 엄청난 양의 순환 참조를 마주하게 됨..



해결책 : 인터페이스 참조

- 각 모듈을 interface와 그 구현체의 쌍으로 분리, 각 모듈의 구현체는 상대의 인터페이스 모듈을 참조하도록 수정
- 모듈화 전에는 딱히 interface 분리가 없어도 (동작에는) 문제가 없었기 때문에 무의식적으로 인터페이스 구분이 되어 있지 않은 부분을 많이 존재했으나, 이를 명확히 구분
- Bonus: 각 모듈은 명시적인 public API를 제공하게 됨



문제점

: 거꾸로 된 의존성 (Inverse Dependencies)

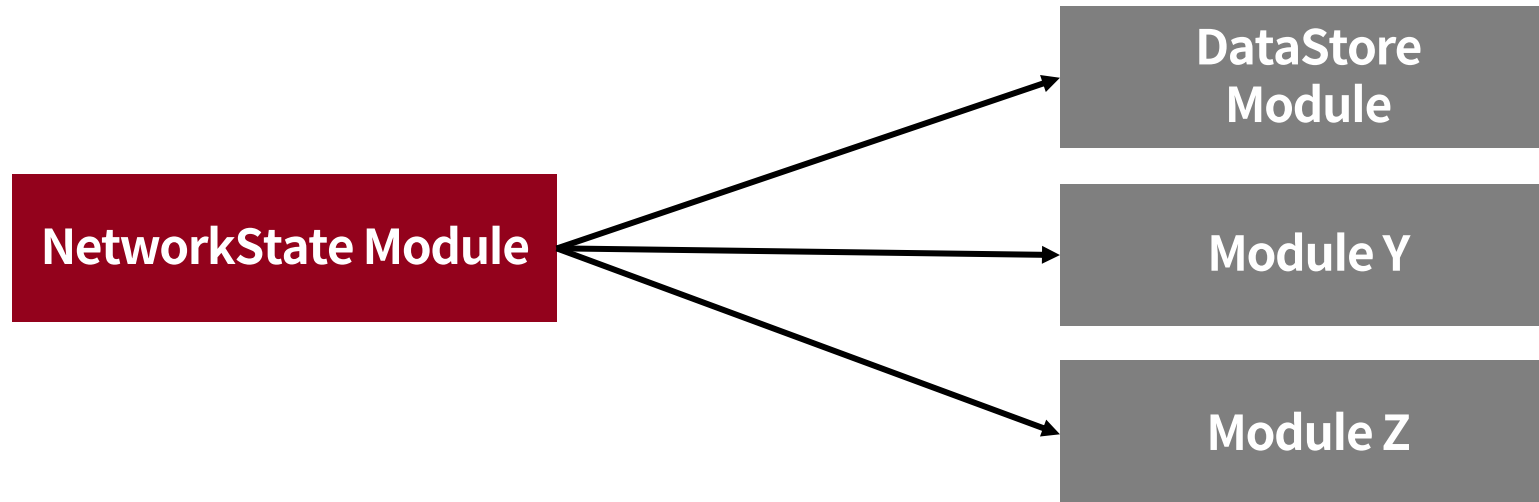
- Eg. Network의 상황의 변경을 감지하는 클래스
- 동작에도, 설계상으로도 별 문제가 없어보이는 코드이지만..

```
class NetworkStateChangeReceiver : BroadcastReceiver {  
    ...  
    override fun onReceive(context: Context, intent: Intent) {  
        updateCachedNetworkInfo(context)  
        NetworkQualityMonitor.refresh(context)  
        if (LocalDataStore.get().checkLiteModeEnabled()) {  
            LocalDataStore.get().clearSessionTimestamp()  
        }  
    }  
}
```

문제점

: 거꾸로 된 의존성 (Inverse Dependencies)

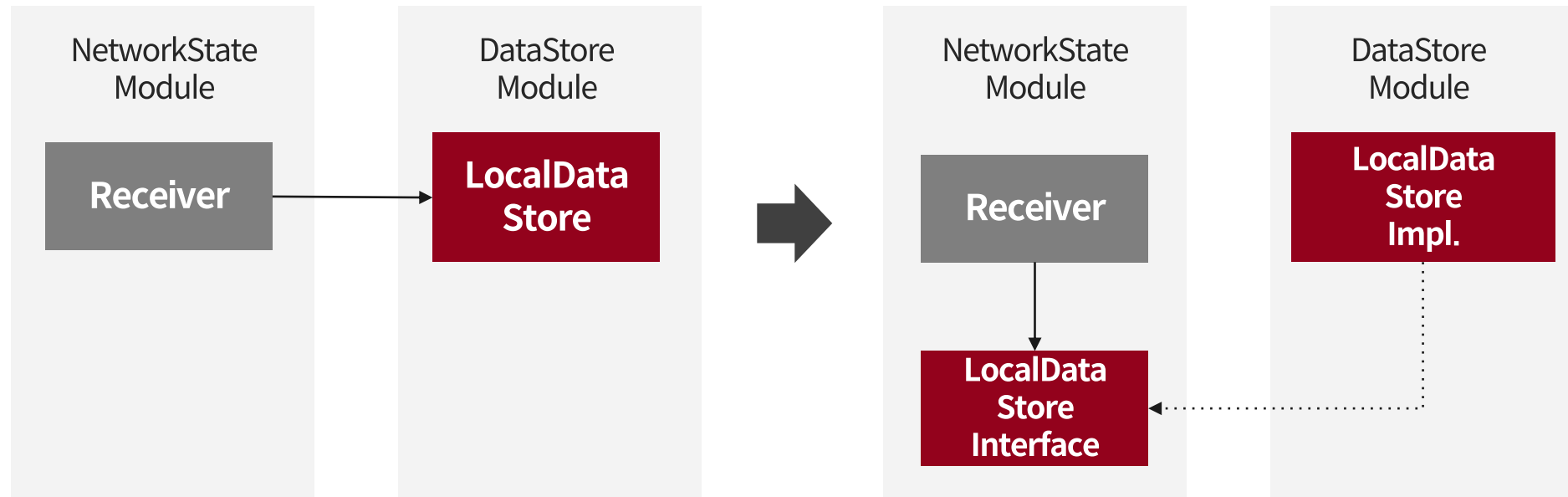
- 하지만 멀티모듈 상황에서는, NetworkState 모듈이 DataStore 모듈에 의존하게 됨
- 개념적으로는 NetworkState가 이에 의존할 이유가 없기 때문에 직관적이지도 이해하기도 어려움 → 멀티모듈의 관계 그래프가 필요 이상으로 복잡해지고, 이해하기도 어려워짐
- 확장에도 문제가 생김: 네트워크 상태가 바뀔 때의 동작이 추가될 때마다 그 동작을 갖고 있는 모듈에 대한 의존성이 늘어남



문제점

: 의존성 역전의 원칙 (Dependency Inversion Principle)

- 원칙1) 추상화된 모듈은 구현 모듈에 의존해서는 안 된다.
- 이 원칙이 깨져 있는 상황이라면, 구현 클래스의 인터페이스를 만들어 상위 모듈로 옮기는 형태로 의존성의 방향을 반대로 바꾸면 해결됨



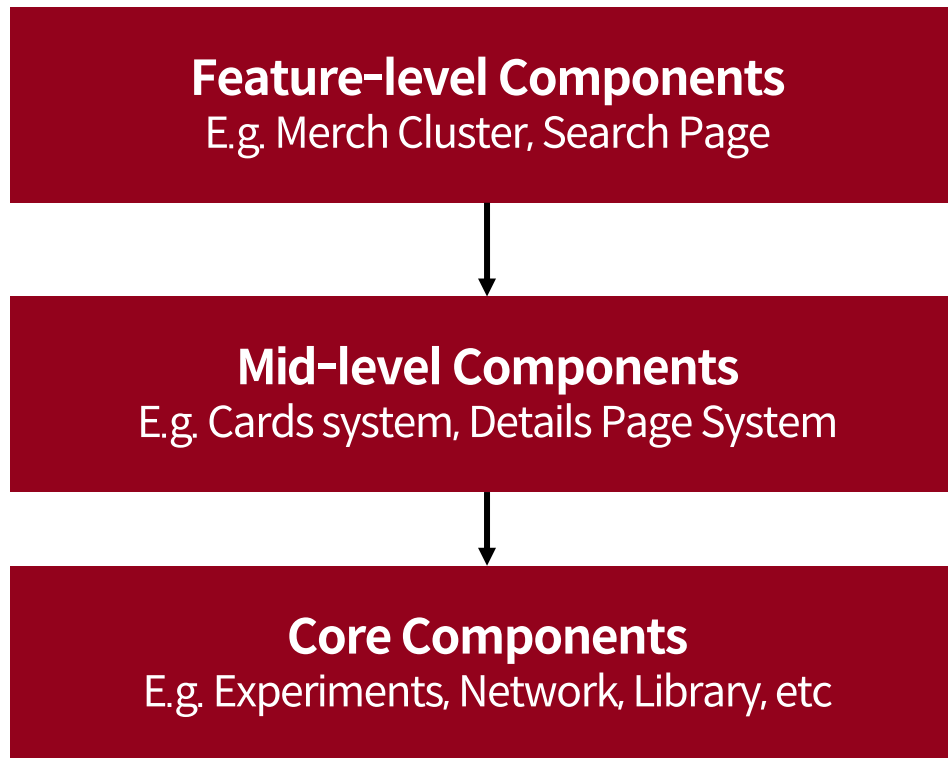
문제점

: 의존성 역전의 원칙 (Dependency Inversion Principle)

```
interface NetworkStateChangeListener {  
    fun onNetworkStateChanged(context: Context)  
}  
  
fun addNetworkStateChangeListener(listener: NetworkStateChangeListener) {  
    ...  
}  
  
class NetworkStateChangeReceiver : BroadcastReceiver {  
    override fun onReceive(context: Context, intent: Intent) {  
        ...  
        broadcastNetworkChange(context)  
    }  
}
```

- 네트워크 변화를 감지하고 싶은 다른 모듈에서 호출하기 위한 API를 추가
- 의존관계의 역전이 일어남
→ 의존성의 방향이 바람직하고 직관적
- 테스트 가능한 코드가 됨
 - NetworkStateChangeReceiver를 DataStore와 독립적으로 테스트 가능
 - DataStore는 NetworkStateChangeReceiver의 mock object를 만들어서 사용할 수 있음

바람직한 의존성 방향



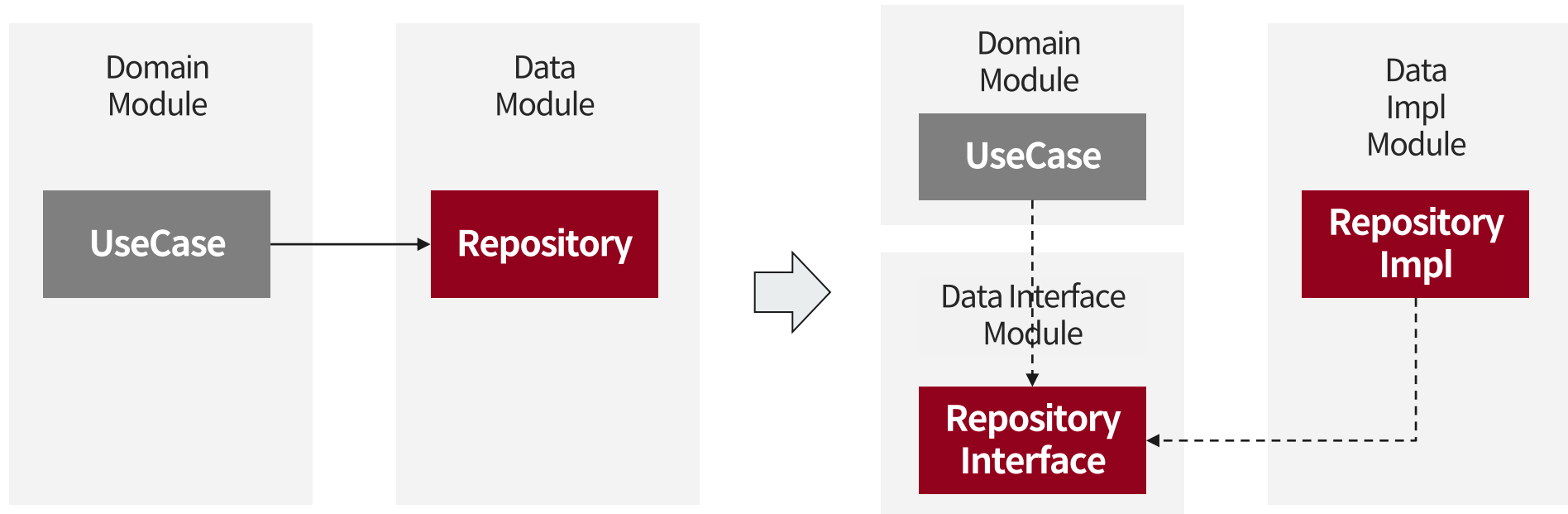
“

의존성이 흘러가는 직관적인
방향으로 모델링 해야 함



의존성 역전의 원칙 (2)

- 2) 상위 모듈은 하위 모듈의 무엇도 직접 임포트하면 안 된다. 상위/하위 모두 인터페이스에 의해서만 각자에 대한 의존성을 가진다
- 이 경우는 의존관계 역전의 결과로 하위 계층이 상위 계층에 의존하게 되는 것이 아님
→ 상위, 하위 계층이 꼭 필요한 인터페이스만 참조하여 캡슐화가 강화됨



의존성 역전의 원칙 (Dependency Inversion Principle)

```
// in Domain Module
class FooUseCase @Inject internal constructor(repository: FooRepository) {
    fun executeFoo() {
        repository.foo()
    }
}

// in Data Interface Module
interface FooRepository {
    fun foo()
}

// in Data Implementation Module
internal class FooRepositoryImpl : FooRepository {
    override fun foo() { ... }
}
```

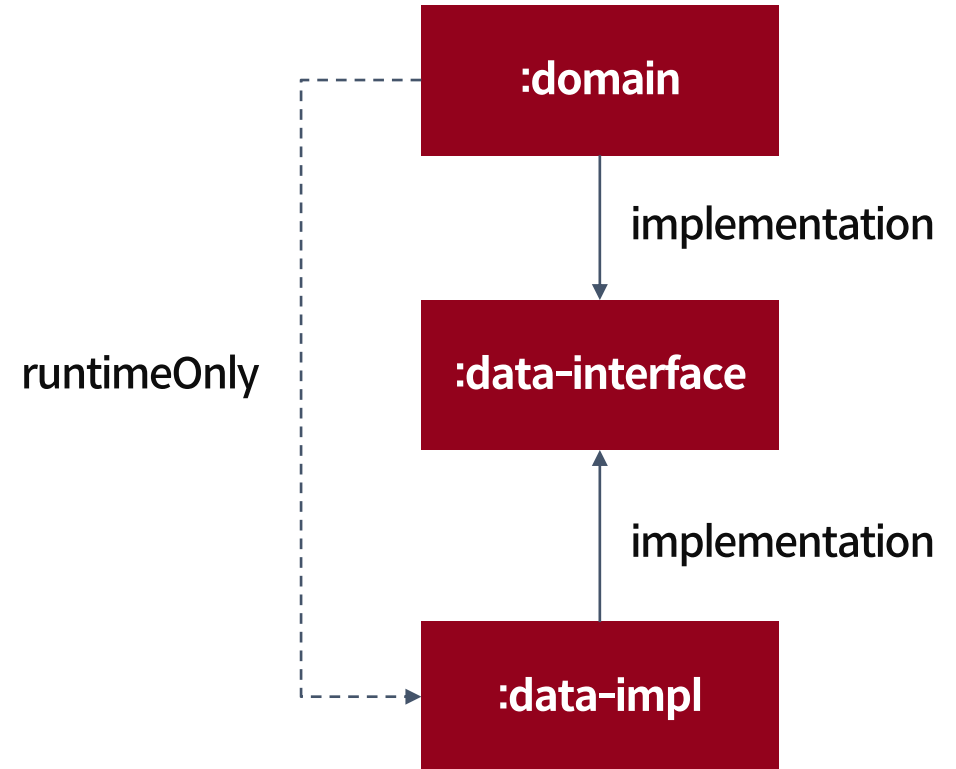
- 상하위 모듈이 모두 중간 모듈의 인터페이스에만 의존함
- Bonus 2: Data Impl 모듈이 수정 되어도 타 모듈들은 다시 빌드되지 않음

```
// domain/build.gradle
implementation project(":data-interface")
runtimeOnly project(":data-impl")

// data-interface/build.gradle 에는 타 모듈에 의존하지 않음

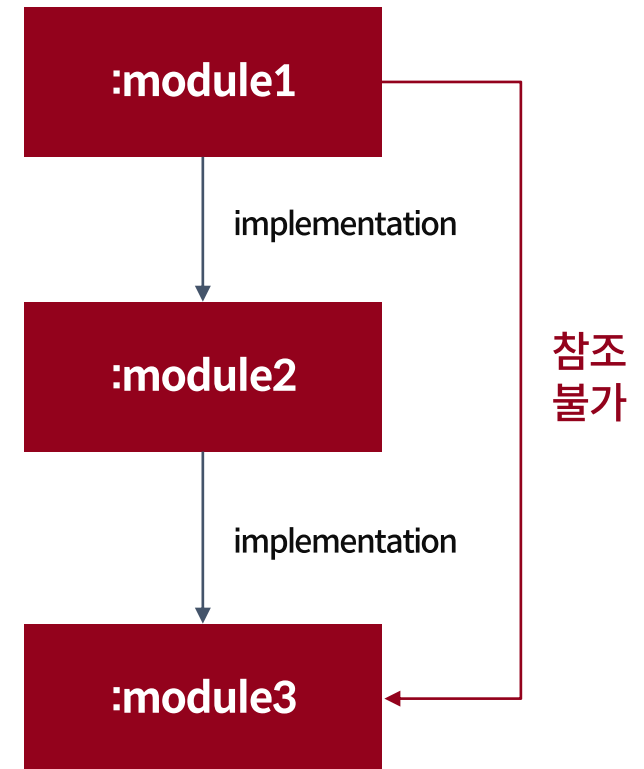
// data-impl/build.gradle
implementation project(":data-interface")

// data-impl/.../di/DataModule.kt
@Module
@InstallIn(SingletonComponent::class)
object DataModule {
    @Provides
    fun provideBarRepository(): BarRepository =
        BarRepositoryImpl()
}
```



멀티모듈을 위한 언어 / 빌드툴 차원의 지원

- Kotlin**
- `private < protected < internal < public`
※ **internal**과 `public`의 차이는? → 다른 모듈에서 접근하지 못하게 함
 - Java의 디폴트 가시성 (`package-private`)과는 달리, 모듈 간의 가시성만을 구분
-
- Gradle**
- `compile` (현재는 `api`) → `implementation`로의 변경
 - ✓ `api`: 내 모듈의 API로서 포함 (상위 모듈에서도 참조 가능)
 - ✓ `implementation`: 내 모듈 내부 구현으로서 포함 (상위 모듈에서 참조 불가)
 - 상위 모듈이 자신의 손자 모듈을 볼 수 없음
 - 캡슐화 강화
 - 손자 모듈의 변경이 일어나도 할아버지 모듈은 재컴파일 할 필요 없음
 - `runtimeOnly`: 컴파일 의존성 트리에서 배제하고 최종 apk 생성시에 통합



문제점 : 없어야 할 의존성이 존재 (예1)

- 모듈로 나누기 전까지는 문제의식 없이 쉽게 사용되는 안티패턴
- 예: 앱의 메인 페이지의 특정 Intent를 얻기 위해 MainActivity의 메소드를 호출해야 하는 경우 → 어떤 페이지/모듈들도 MainActivity에 의존성을 가질 가능성이 있음
- 어떤 비즈니스 로직도 특정 Activity에 의존하면 안 되기 때문에 심각한 아키텍처 상의 문제

```
fun showInstallMessage(title: String, isUpdate: Boolean, ...) {  
    ...  
    clickIntent = MainActivity.getDownloadIntent(context)  
    ...  
}
```

문제점

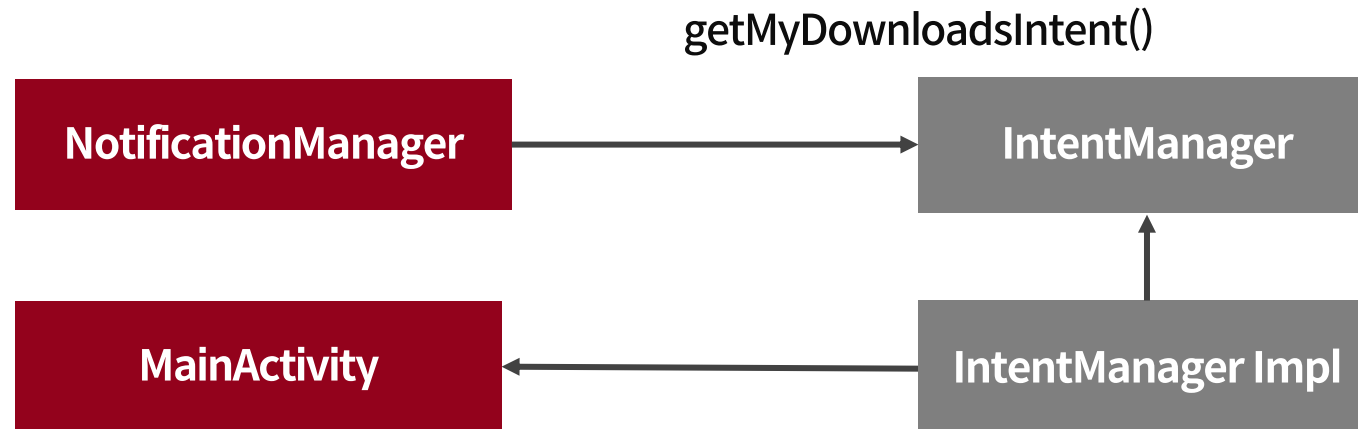
: 없어야 할 의존성이 존재 (예2)

- 탭 레이어에서 자주 보이는 문제 - 최상위 Fragment의 멤버를 호출
- UI 코드는 자신이 PageFragment 안에서 표시될 것이라고 가정해서는 안 됨에도 불구하고 잘못 호출 → 사실은 여기서 간헐적으로 에러가 나고 있었으나 재현율이 낮아서 수정되지 못하고 있었음-_-;

```
override fun onBylinesClick(info: BylineEntryInfo) {  
    try {  
        val clickIntent = getIntent(info)  
        context.startActivity(clickIntent)  
    } catch (e: ActivityNotFoundException) {  
        containerFragment.getPageFragmentHost()  
            .showErrorDialog(context.getString(info.getErrorMessageld()))  
    }  
}
```

해결책 : 새로 발견된 역할을 담당해줄 모듈을 생성

- 앞서 예는 Activity/Fragment가 더 쪼개질 수 있음을 보여주는 예라고 할 수 있음
- 첫번째 예를 위해서는, 전역적으로 모든 Intent 들을 관리해주는 IntentManager를 추가 해서 해결 가능
- Tip: Navigation 문제도 같은 방법으로 해결 가능



문제점

: god object에 대한 의존성

- 멀티모듈에서는 단순히 MyApplication의 하나의 메소드를 호출하는 행위가, :app 모듈을 의존함을 의미함 → 순환 참조
- Application를 호출하는 클래스들을 테스트하게 어렵게 만듦
→ mock을 만든다해도 Application 클래스 내의 수많은 메소드에 대해 모두 만들어야..

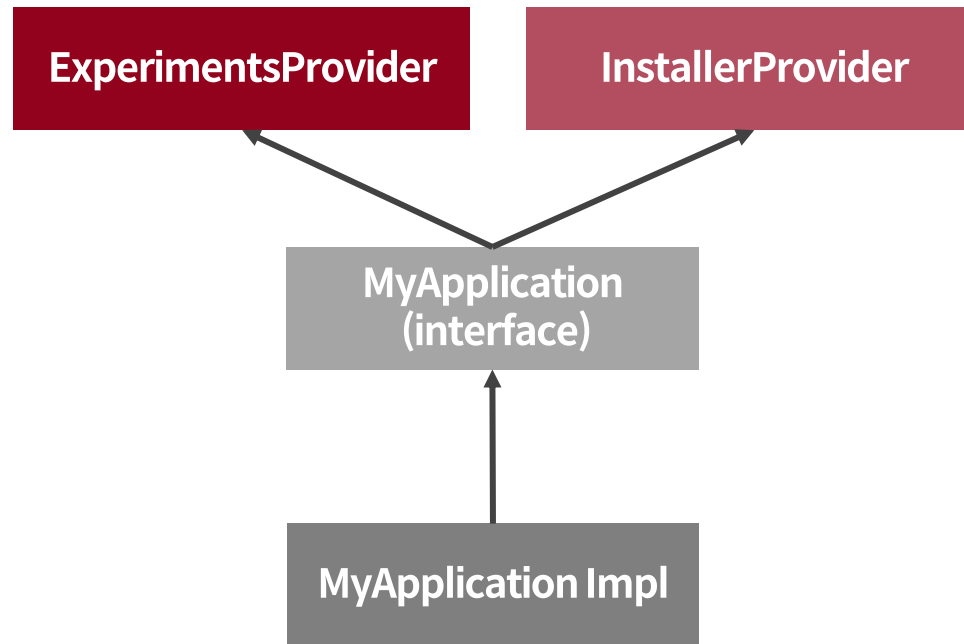
```
class MyApplication : Application {  
    fun getPackageManager(): PackageManager  
    fun getLibraries(): List<Library>  
    fun getExperiment(): Experiment  
    fun getInstaller(): Installer  
    ...  
}
```

// 앱의 전역에서 이런 식으로 호출됨..

```
MyApplication.get().getPackageManager()  
MyApplication.get().getExperiment()  
...
```

해결책 1단계 : god object를 쪼개기

- 첫번째 문제를 해결 - 더 이상 Application 객체에 직접 의존할 필요가 없어짐



해결책 2단계 : 의존성 주입 적용

```
@Inject
lateinit var packageManager: PackageManager

@Inject
lateinit var dataLoader: DataLoader

class Foo @Inject constructor(
    private val packageManager: PackageManager,
    private val dataLoader: DataLoader,
    ...
)
```

- DI 프레임워크가 알아서 인스턴스를 주입할 수 있도록 수정
- Application에 대한 의존성을 완전히 분리
- Hilt 등을 통해 쉽게 테스트 설정 가능

문제점

: 너무 많은 유틸리티 클래스들

- AccountUtils, AdUtils, AppRestartScheduler, AuthTokenUtils, AutoUpdateUtils, ArrayUtils, CachedLocationAccess, DateTimeUtils, DeviceConfigurationHelper, DeviceManagementHelper, ErrorResourceHelper, IntentUtils, LocationHelper, MainThreadStack, MathUtils, Md5Utils, ...
- 시스템이 커지면서 끝없이 새로 생기는 유틸리티들..
- 왜 문제인가?
 - ✓ 하나의 유틸리티 클래스에 접근해도 유틸리티 모듈에 대한 의존성을 갖게 됨
→ 그것도 전체 유틸리티 클래스에 의존
 - ✓ 각 유틸리티 클래스들이 갖고 있는 추가 의존성들도 함께 갖게 됨
→ 필연적으로 순환 참조를 야기
 - ✓ 관리 책임(accountability)의 문제: 이 많은 클래스들을 대체 누가 관리하나?

문제점 : 너무 많은 유틸리티 클래스들

- 오래된 프로젝트의 경우 유틸리티의 숫자뿐 아니라 사이즈도 문제가 될 수 있음
- 심지어 특정 유틸리티 하나의 사이즈가 너무 큰 문제도 생김
→ 카테고리를 나누기 애매해서 그냥 Utils class에 집어넣은 메소드들이 시간이 지나다보면 엄청난 숫자가 됨
- 왜 문제인가?
 - ✓ 특정 유틸리티 메소드 하나를 호출함에도 유틸리티 클래스 전체에 대한 의존성을 갖게 됨
 - ✓ 범용 유틸리티 특성상 유틸리티가 네트워크 모듈 등 여러 곳에 의존성을 갖는 경우도 많음
→ 추가 의존성으로 의존성 그래프가 걷잡을 수 없이 복잡해짐
→ 네트워크를 사용하지 않음에도 해당 모듈에 대한 의존성을 함께 갖게 되어 버림

해결책1

: 유틸리티 클래스들을 해체

- 할 수 있는 한 유틸리티 패턴을 지양
 - ✓ 실제로 대부분의 유틸리티 메소드들은 일부 호출자에 의해서만 사용됨
 - ✓ 각 메소드의 reference들을 파악해서 많은 곳에서 호출되지 않는 경우라면 호출하는 쪽으로 이전해서 private 혹은 internal로 만들
- 큰 유틸리티 클래스, 모듈을 더 잘게 쪼갬
 - ✓ 큰 유틸리티 클래스는 더 의미있는 작은 단위로 쪼갬
 - ✓ 쪼갬 클래스들은 적절한 모듈로 재배치
 - ✓ 일반적인 “Utils” 모듈은 앱의 비즈니스 로직과 전혀 관계 없는 정말 일반적인 것들만으로 구성
 - Eg. ArrayUtils, ListUtils, ...

그럼 어디까지 나눠야 하나??

“

큰 유틸리티 모듈을 작은 것들로 나누는 게 좋다면,
상당히 많은 양의 모듈이 될 수도 있는데 괜찮을까??

이에 대한 대답은 대부분의 경우 Yes! 임



문제점 : 정적인 유틸리티 함수들

- 앞서 문제에서 본 것처럼 Application을 이용한 접근은 바람직하지 않음
- 정적 메소드의 경우, 의존성 주입의 대상이 안 됨
- 정적 메소드는 mock을 통한 테스트도 어려움

```
class Utils {  
  companion object {  
    fun isFeature1Enabled(): Boolean =  
      MyApplication.get().getExperiments().isEnabled(MyExperiments.FEATURE1)  
      && !MyApplication.get().getPolicies().isFeatureProhibited()  
    ...  
  }  
}
```


해결책 : 정적 클래스들을 싱글톤으로 전환

- Application 객체가 아닌 의존성 주입을 통해 인스턴스를 얻을 수 있음
- 호출하는 쪽은 쉽게 mock을 만들어 테스트 할 수 있다

```
class Utils(  
    private val experimentProvider: ExperimentsProvider,  
    private val policies: Policies  
) {  
  
    fun isStickyWifiEnabled(): Boolean =  
        experimentProvider.getExperiments().isEnabled(MyExperiments.FEATURE1)  
        && !policies.isFeatureProhibited()  
    }  
}
```

문제점

: 부수효과(side effect)로 인한 테스트의 어려움

- 테스트 케이스 안에서 LoggingContext와 FrontendList를 어떻게 mock으로 교체할 수 있을까?
- 게다가 각각의 클래스가 다른 모듈에 속해 있다면?

```
fun bindView(document: Document) {  
    ...  
    val feList = FrontendList(document)  
  
    val loggingContext = LoggingContext()  
    loggingContext.logImpression(this)  
}
```

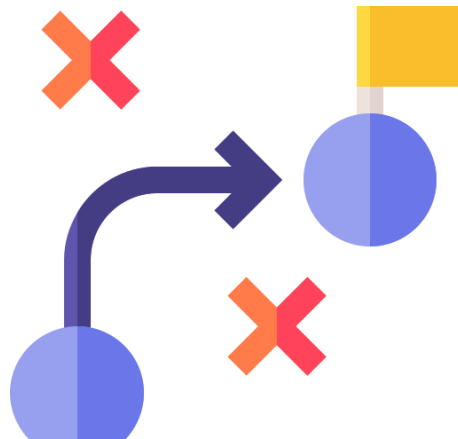
해결책 : 팩토리 메소드 추가

- 팩토리 클래스들을 통해서 fake / mock 오브젝트 생성
- Bonus: 필드 주입을 완전히 막을 수 있음. 보통 팩토리들은 생성자 주입을 통해서 의존성을 얻기 때문

```
fun bindView(document: Document) {  
    ...  
    val feList = frontendListFactory.create(document)  
  
    val loggingContext = loggingContextFactory.create()  
    loggingContext.logImpression(this)  
}
```

그 외 : 리소스의 모듈화

- 종종 리소스들의 분류와 기능(피처)상의 분류가 일치하지 않는 경우가 있음
- Playbook
 - ✓ 리소스가 모듈 A 한 군데에서만 사용된다면 → A로 이동
 - ✓ 리소스가 모듈 A, B에서 사용되지만, 모듈 B가 A에 의존한다면 → A로 이동
 - ✓ 리소스가 모듈 A, B에서 사용되지만 두 모듈 사이에 의존 관계가 없다면,
 - 논리적으로 리소스가 A에 속한다면 A로 이동 후, 모듈 B가 A에 의존하도록 의존성 추가
 - 특정 모듈에 속해야 할 개연성이 떨어진다면, common 리소스 모듈로 이동



결론

“

모듈화에서 생기는 문제들은 보통
큰 스케일의 아키텍처에서 나오는 문제가 불거진 증상

→ 해결책은 각 시스템과 모듈의 책임을
명확히 정의해주는 것!



수고하셨습니다!

