# **Правила архитектуры android**

Module или Модуль = gradle module

DiModule = Модуль для скоупа

**Важное примечание:**

* Функционал не должен нарушать правила
* Если не получается реализовать функционал по правилам:
  + Проверить актуальные правила, возможно они были изменены.
  + Обратиться за помощью к старшему группы/эксперту/тимлиду/архитектору
    - Они должны помочь построить архитектуру так, чтобы она вписывалась в правила.
    - Если все вплоть до тимлида или архитектора не могут вписать функционал в правила, значит нужно расширить правила таким образом, чтобы существующий функционал не перестал им удовлетворять и новый функционал имел возможность быть реализованным по правилам.
  + Функционал обязан удовлетворять правилам. Правила могут быть расширены для нового функционала, но только с согласия тимлида и архитектора.

**Список сущностей для функционала:**

* DiModule
* IFeature (Router, View, Presenter)
* Router : IFeature.Router
* Fragment : IFeature.View
* Presenter : DisposablePresenter, IFeature.Presenter
  + ErrorHandler
* Interactor
* IFeatureState : IState (доработать без зависимости от RxJava)
  + FeatureState : AbstractState<StateModel>, IFeatureState
  + CommandFeatureState
  + ObserverFeatureState
* Repository
  + RequestExecutor
  + ErrorTransformer
* Mapper
* Api

**Зависимости модулей (все через implement)**

* App
  + Mediator
    - Feature1
      * CoreFeature
        + api CoreCommon
      * Domain1
        + CoreDomain

api CoreCommon

* + - * Domain2
        + CoreDomain
    - Feature2
      * CoreFeature
      * Domain1
        + CoreDomain
      * Domain2
        + CoreDomain

FeatureModule не зависит от другого FeatureModule

FeatureModule может зависеть от нескольких DomainModule

DomainModule может зависеть от других DomainModule (следить чтобы не было циклических зависимостей)

Mediator зависит от всех Feature и Domain(нужно ли?) модулей.

**Что содержат модули**

* App
  + Application
  + RootActivity (когда будет SingleActivity)
  + AppModule
* Mediator
  + Launcher (Chain с чего начать приложение)
  + Router (реализации IRouter для всех фич)
* Feature
  + public
    - DiModule
    - IRouter (список выходов из фичи)
    - Screen
    - Data
  + internal
    - IFeature (View, Presenter, Router - выходы из фрагмента)
    - Fragment
    - Presenter
    - Interactor
    - \* State
* CoreFeature
  + IView
  + BaseFragment
  + IPresenter
  + DisposablePresenter
  + IRouter
  + Result
  + DomainError
  + UtilsFeature
* Domain
  + public
    - DiModule (устанавливает зависимости для IRepository)
    - IRepository
    - Entity Models (Модели возвращаемые из IRepository)
    - \* CustomExpectedException
  + internal
    - Repository
    - \* CustomErrorTransformer
    - Mapper
    - Api
* CoreDomain
  + RequestExecutor
  + ErrorTransformer
  + IMapper
  + Cache (прорабатывается)
  + UtilsDomain
* CoreCommon
  + ExpectedException
  + Common Gradle Dependencies (toothpick, coroutines...)
  + UtilsCommon
  + AppState (прорабатывается / сущность для публикации данных на все приложение. Одна фича публикует данные по ключу, другая при необходимости подписывается на этот ключ и получает свежие данные)

\* присутствует по необходимости.

**Работа с DI:**

Порядок наследования скоупов

* AppScope - общие зависимости необходимые всему функционалу приложения (2-3 фичи не считаются всем приложением)
* FragmentScope - зависимости необходимые для работы конкретной фичи

Фиче модуль может состоять из нескольких фрагментов.

Зависимости фрагментов нужно проектировать таким образом, как будто каждый фрагмент является отдельным независимым модулем. В крайнем случае смотри про Зависимые фрагменты.

DomainModule

* публикует наружу свой DiModule, в котором описан граф построения его зависимостей.
* не имеет собственного скоупа, только DiModule.
* может использовать для построения своих зависимостей только иньекции из AppScope, он ничего не знает про Фичу в которой будет использоваться.
* FragmentScope устанавливает в себя необходимые DiModule (Свой и нужных DomainModule'й)

FragmentModule

* скоуп привязан к ЖЦ фрагмента
* родительским скоупом является AppScope или другой FragmentScope, фрагмента непосредственно связанного с текущим.
* два фрагмента могут быть зависимыми или независимыми.

Независимые фрагменты

* Независимый фрагмент - самостоятельная боевая единица, для работы которой нужны только входящие данные.
* Данные для работы экранов передаются через аргументы фрагмента и устанавливаются зависимостями в его DiModule, для получения их в презентере.
* Фрагмент имеет свой инстанс интерактера с бизнес логикой, которая не хранит состояние нужное другому фрагменту.
* Запросы апи независимые и кешируются на время жизни этого фрагмента.

Зависимые фрагменты

* Зависимые фрагменты - экипаж танка, боевой единицей являются только вместе.
* Фрагмент может иметь родительским скоупом другой фрагмент только если эти фрагменты связаны одной неразрывной функциональностью.
* Зависимые фрагменты делятся на master и slave. Граф зависимостей выглядит как AppScope - FragmentScopeMaster - FragmentScopeSlave
* Например: Экран списка и экран с подробностями выбранного элемента. т.е. экраны которые нелогичны для одиночного использования.
* Под нелогичностью подразумевается, что экран не может использоваться в другом месте сам по себе из-за ограниченности информации, а не потому что он принадлежит тому же функционалу, что и предыдущий фрагмент. И что его работа невозможна без одного и тогоже инстанса какой-либо сущности, что и у предыдущего экрана.

**Работа с данными и фрагментами**

1. Передача данных фрагменту на вход

* Presenter(VM) имеет данные и передает их в роутер в виде примитивов или общедоступных Parcelable объектов.
* Реализация Router создает из полученных параметров объект Data и передает его в Command.
* Объект Data лежит рядом со Screen и является Parcelable.
* Data принимается в Fragment.newInstance и сохраняется в аргументы.
* Fragment открывает свой Scope и устанавливает в него модуль, в конструктор которого передает Data из аргументов.
* Далее Data получает Presenter(VM) через инъекцию.

2. Обновление данных в рантайме на разных фрагментах на одном экране

* Если необходимо обновлять одни и те же данные на 2 и более фрагментах, которые являются связанными, то это делается через общий инстанс State или Interactor, в зависимости от типа данных.
* Если это не связанные фрагменты, то смотри пункт 3.

3. Актуализировать данные на уровне всего приложения (Прорабатывается)

* Для того, чтобы обновить какие-либо данные и обновленное значение было доступно на уровне всего приложения, в AppScope будет добавлен в зависимости необходимый механизм. AppState (пока пример)
* Получаем инъекцией в Interactor реализацию AppState.
  + AppState представляет из себя Map<Type, DataWrapper<\*>>
  + Ключем к данным является их тип/класс.
  + Данные обернуты в класс DataWrapper<Type> который умеет кастить данные в нужный тип. (Или хранится как Any, а кастом занимается сам AppState)
  + Проверка целостности и соответствия типов данных с ключем осуществляется в реализации AppState.
* Через метод update(type, data) обновляем данные.
* Через метод get<type>() получаем актуальные данные сразу нужного типа
* Через метод obsetve<type>() получаем цепочку для подписки на данные приведенные к нужному типу. (разобраться как сделать подписки для корутин или без зависимости от rxJava)

4. Кеширование данных (прорабатывается

* Существует 2 типа кеширования (имеется ввиду по жц)
* Глобальное кеширование - сохраняются в инстанс глобального кеша и хранятся все время жизни приложения.
  + Не злоупотребляем, т.к. он ограничен по памяти и если все будут в него сохранять данные, то более старые будут вычищаться.
* Локальное кеширование - инстанс кеша на уровне FragmentScope, данные хранятся пока не выйдем с экрана.(пока не закроется скоуп)

5. Сохранение данных на устройстве

* Сохранение данных на устройстве осуществляется через зависимости от специальных DomainModule предназначенных для работы с базой данных, файловой системой, общими префами и прочими хранилищами на устройстве.

6. Восстановление данных

* Данные переданные между фрагментами сохраняются в аргументы
* Сохранение State реализуется самостоятельно через onSaveInstanceState (прорабатывается)
* Сохранение и восстановление глобального кеша разруливает Activity (прорабатывается)
* Так же на activity AppState. (прорабатывается)
* Локальный кеш не сохраняется.