

Clean architecture

Кульминация чистого кода

Содержание

1. Что такое чистый код
2. Что такое чистая архитектура
3. Переписываем код

1. Что такое чистый код

Итак, до сих пор мы не раз говорили о том, что писать код нужно исходя из неких принципов, иначе же ваш код будет хаосом. Можно встретить много старых проектов, где не соблюдались никакие принципы и читать такой код очень сложно и тем более поддерживать его. До сих пор мы рассмотрели следующие принципы:

1. ООП (инкапсуляция, наследование, полиморфизм)
2. SOLID (SRP, OCP, LSP, ISP, DIP)
3. KISS (KEEP IT SHORT AND SIMPLE)
4. DRY (DON'T REPEAT YOURSELF)
5. YAGNI (YOU AREN'T GONNA NEED IT)

Но кажется этого мало и кульминацией принципов написания чистого кода стало понятие чистой архитектуры. Давайте напомним что такое чистый код: код, который

1. легко читать,
2. легко его понимать без посторонней помощи, комментариев (и джавадоков?)
3. тестируемый
4. масштабируемый
5. поддерживаемый

Давайте подробнее остановимся на каждом пункте.

Код который легко читать – согласитесь, если в вашем классе более 250 линий кода, то его уже сложно читать, в нем слишком много линий. Если в вашем классе более 5 публичных методов, то уже непонятно что он делает (нарушение SRP). Если в вашем методе более условно 20 линий кода или если ваш метод принимает более 5 аргументов, все это усложняет читаемость.

Код который легко понимать – здесь вопрос о наименовании классов, интерфейсов, методов, переменных. Плохие имена убивают очень много времени на понимание сути. Имена должны говорить однозначно что они такое и что делают без лжи. Никто не должен страдать из-за того, что ваша переменная “a” непонятно где используется и что делает. Подробнее о том, как

именовать переменные можно прочитать в книге Роберта Мартина : Чистый код и не только об этом. Настоятельно рекомендую.

Код который тестируемый – это во-первых к тому, что комментарии о том, как работает ваш класс или метод не нужны, да и джавадоки тоже не нужны. Покажите как обращаться с вашим кодом в тестах. Юнит тесты созданы не только для того чтобы вы были уверены что написали корректный код, но еще и для того чтобы показать всем кто хочет использовать ваши классы как их нужно использовать. Действительно хорошо написанный юнит тест на класс стоит дороже всех возможных джавадоков и описаний на 100 страниц. Плюс вы всегда можете понять сломали ли что-либо в коде если ваши тесты перестали проходить. И речь не только о юнит тестах, есть и юай тесты (которые мы будем писать). Перед тем как рефакторить код можно написать юай тест и после рефакторинга если они проходят, значит вы справились с задачей. Так же вы можете прийти на проект где отсутствуют тесты и непонятно что как работает. Тогда вам помогут юай тесты перед рефакторингом.

Масштабируемый код – код, в котором масштабировать функционал легко и просто. Т.е. представьте что у нас в проекте не только получение данных от сервера шуток, но и другой сервер аналогично работающий. Насколько просто вам будет добавить аналогичный функционал в проект. Если вы сможете это сделать просто и быстро, значит ваш код масштабируемый.

Поддерживаемость – самое важное – насколько легко вам внедрять изменения в код. Если вам завтра скажут что нужно изменить юай и добавить кнопку или убрать ее. Сколько линий кода вы измените в проекте и сколько часов потратите на это. Если такая простая задача потребует переписывания всего кода в проекте – у вас явно неподдерживаемый код.

Расскажу одну историю про поддерживаемость. Однажды бэк разраб захотел попробовать добавить фичу в проект который я написал единолично по принципам чистой архитектуры.

У нас были 3 вкладки, нужно было добавить 4. Функционал следующий – получить список треков от сервера, закешировать при необходимости и сохранить выбор юзера. Он справился за 20 минут. При том что он не андроид разработчик и впервые видел в жизни чистую архитектуру. Но опять же – хорошие имена классов и структура проекта дали ему эту возможность.

Оценить свой код легко: сколько нужно времени чтобы новый человек на вашем месте понял что к чему и смог написать новую фичу или модифицировать существующую.

Итак, исходя из этих требований к чистому коду Роберт Мартин представил концепцию чистой архитектуры. Вы уже заметили, что в нашем проекте где мы просто получаем данные от сервера, сохраняем в бд и отображаем юзеру написано уже много классов, интерфейсов и так далее. Мы бы могли написать все это в активити на 2000 линий кода. Но тогда бы никто не смог бы ни прочитать это все ни поддерживать в дальнейшем. Да, мы уже сделали неплохо. Но можно лучше и правильнее.

2. Что такое чистая архитектура

Суть чистой архитектуры это разделение на слои. Ведь давайте подумаем : у нас есть слой презентационный – активити, вьюмодель. Где мы занимаемся лишь тем что отображаем данные юзеру и отвечаем за юай логику, когда показать ошибку, когда данные и так далее.

Так же у нас есть слой данных – наша модель (далее переименуем в репозиторий) где мы получаем данные из сети или из бд и занимаемся чисто тем, что кешируем, обрабатываем и так далее.

И самый сложный для восприятия и понимания это слой бизнес логики. Что это такое?

Смотрите, у нас есть слой данных – где просто получаем данные и немного обрабатываем их. Так же у нас есть слой презентационный, где другие данные показываем юзеру. Но как связать один слой с другим? Как данные полученные от сервера превращаются в данные которые видит юзер? На этот вопрос отвечает слой бизнес логики. Но кроме этого это тот слой, в котором описана бизнес логика. Что значит бизнес логика? В презентационном слое у нас логика отображения. Повороты, ошибки и так далее. А в слое данных у нас логика конкретно о работе с данными, как получать из сети, как хранить в бд и так далее. Но одна логика(юай слоя) без логики другой (данных) не имеет смысла. Нужен мост. Нужно связать одно с другим. Это и есть слой бизнес логики.

Чтобы вы понимали еще лучше: в нашем приложении есть некий функционал – получить шутку, получить избранную шутку, сохранить шутку в избранное, убрать из избранных. Вот это и есть бизнес логика. Запомните, то как звучит логика из уст непрограммиста : вашего начальника или гуманитария : это и есть бизнес логика. Она обезличена и не имеет в себе ничего что относится к конкретным технологиям. Бизнес логика не может звучать как логика данных – получить из реалма шутку если она есть. Никаких если. Никакого упоминания реалма или других технологий. Просто: юзер должен иметь возможность посмотреть новую шутку, добавить ее в избранные, убрать из избранных, посмотреть все избранные. Мы так же не говорим о том, как это будет выглядеть – будет ли кнопка или иконка сердечко. Это все уже логика юай слоя. Это все должен придумать дизайнер. Где хранить на сервере – серверный разработчик. Ваша же обязанность имплементировать все это – т.е. реализовать.

Задача должна выглядеть именно так – я как юзер хочу иметь возможность прочитать новую шутку. Я как юзер хочу иметь возможность добавить ее в избранное. Я как юзер хочу прочитать избранную шутку. Я как юзер хочу убрать шутку из избранных.

Хорошо, теперь, когда стало понятно что к чему, предлагаю вернуться к коду.

Мы сказали что для слоя презентационного у нас есть активити и вьюмодель (плюс комуникатор и данные юай слоя JokeUiModel). Для работы с данными у нас датасорсы и серверная модель плюс реалм модель. И все это внутри модели, которую переименуем в репозиторий. А что же тогда с логикой? Где ее писать? Как?

Интеракторы. Многие до сих пор не понимают что это такое и зачем они нужны. Но мне кажется я уже в достаточной мере представил суть и необходимость. Кроме того – если у вас есть промежуточный слой между данными и юай вы сможете подменять бизнес логику.

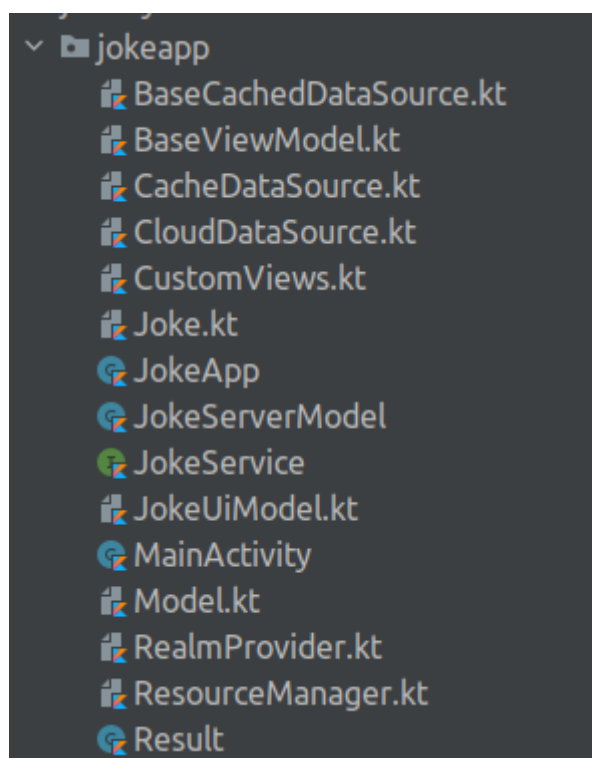
Предположим завтра вы захотите добавить покупки в проект и только те юзеры которые оформили подписку смогут сохранять данные в бд. Значит вам теперь писать логику проверки, так? А где? В юай слое? В слое данных? Репозиторий должен получать данные и заниматься только этим. Юай слой только отображением полученных данных. Для этого мы и пишем интеракторы. У вас будет 2 интерактора – для простых юзеров и для премиум юзеров. На старте приложения будет проверяться наличие подписки и использоваться нужный интерактор. Для премиум юзеров будет доступно все, а для непримиум юзеров недоступно

сохранение шуток. Вы так же сможете написать экран подписок и перекидывать непремиум юзера туда когда он попробует нажать на сердечко. А еще бывает что вам на разных экранах нужна одинаковая бизнес логика. У меня было такое, когда на разных экранах была возможность оформить подписку со скидкой. На главном экране и в меню. ЮОай разный абсолютно, но логика одинакова. Там еще таймер был по истечению времени которого предложение оформить подписку по скидке исчезало. Значит я поступил так : написал интерактор для этой логики и переиспользовал ее в 2 вьюмоделях. Вот и все. Не надо писать огромные куски кода ни в вьюмодели ни в репозитории. Для этого и есть интеракторы.

3. Переписываем код

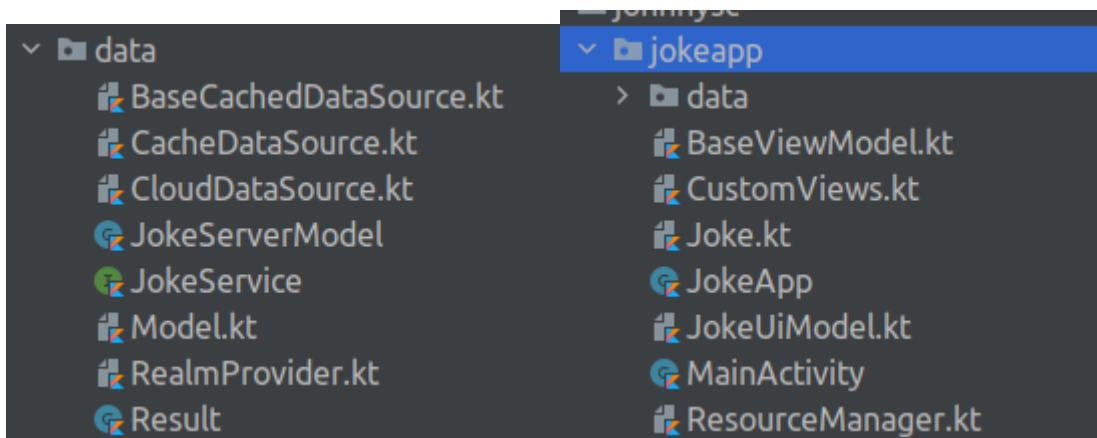
Теперь, когда я убедил вас в необходимости переписать код давайте посмотрим на то, что у нас есть. У нас огромная модель, в ней слишком много всего происходит. Нам нужно переписать код там и добавить интеракторы. Для начала переименуем в репозиторий.

И да, давайте все же отобразим структуру в проекте. Создадим пакеты, а то у нас все классы в одном пакете



Так не должно быть. Это как если бы вы на рабочем столе хранили все свои файлы. Нужна структура и мы сделаем по чистой архитектуре 3 пакета для начала: presentation, domain, data.

Все что не подпадет под эти понятия мы оставим в корне. Правый клик на jokeapp пакет и выбираем New → package. Сначала разберемся с дата слоем. Перебросим в пакет все что относится к данным. И теперь стало намного лучше, в корне осталось намного меньше классов. Теперь приступим к модели. Для начала переименуем уже в репозиторий – общепринятое название для класса который работает с данными и внутри уже инкапсулирует датасорсы (кстати погуглите патерн фасад).



Итак, вот так выглядит теперь наш интерфейс репозитория

```
interface JokeRepository {  
    suspend fun getJoke(): JokeUiModel  
    suspend fun changeJokeStatus(): JokeUiModel?  
    fun chooseDataSource(cached: Boolean)  
}
```

И сразу стоп. Методы возвращают юай модель? Это из другого слоя. Так не должно быть. Методы датаслоя должны вернуть данные дата слоя. А интерактор уже смапит их куда нужно. Поэтому нам придется переписать реализацию так же. И давайте создадим класс JokeDataModel который будет создаваться из JokeServerModel и из JokeRealmModel.

```
class JokeDataModel(  
    private val id: Int,  
    private val type: String,  
    private val text: String,  
    private val punchline: String,  
)
```

Ладно, мы написали класс для датаслоя. Оставим пока его в покое и перепишем наши JokeServerModel и JokeRealmModel чтобы они мапились к этому классу.

```
open class JokeRealmModel : RealmObject() {  
    @PrimaryKey  
    var id: Int = -1  
    var text: String = ""  
    var punchline: String = ""  
    var type: String = ""  
  
    fun toJokeDataModel() = JokeDataModel(id, type, text, punchline)  
}
```

```
data class JokeServerModel(
    @SerializedName( value: "id")
    private val id: Int,
    @SerializedName( value: "type")
    private val type: String,
    @SerializedName( value: "setup")
    private val text: String,
    @SerializedName( value: "punchline")
    private val punchline: String
) {
    fun toJokeDataModel() = JokeDataModel(id, type, text, punchline)
}
```

И мы сразу видим что один и тот же метод в 2 классах. Первое что нужно сделать это конечно же выделить метод в интерфейс, но этого мало. Давайте напишем общий дженерик метод для мапинга одних данных к другим. Принято те классы, которые могут использоваться везде в проекте выносить в некий core пакет и мы создадим мапер

```
interface Mapper<R> {
    fun to(): R
}
```

И перепишем наши классы таким образом

```
open class JokeRealmModel : RealmObject(), Mapper<JokeDataModel> {
    @PrimaryKey
    var id: Int = -1
    var text: String = ""
    var punchLine: String = ""
    var type: String = ""

    override fun to() = JokeDataModel(id, type, text, punchLine)
}
```

Согласен, выглядит пока не супер классно, но мы придумаем нечто получше в ходе рефакторинга. Теперь вернемся к нашим датасорсам. Я бы хотел чтобы их объединял один интерфейс и они оба возвращали нужные мне объекты JokeDataModel. Для этого давайте перепишем наши датасорсы и в первую очередь JokeDataFetcher

```
interface JokeDataFetcher {
    suspend fun getJoke(): JokeDataModel
}
```

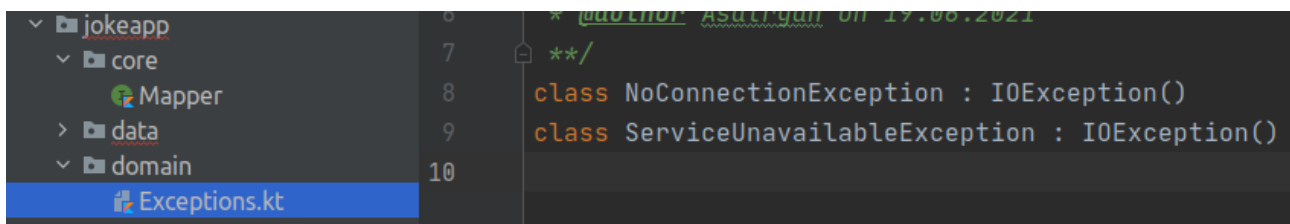
Никаких дженериков больше не нужно. Давайте для начала перепишем наш клаудДатасорс. Теперь он выглядит намного приятнее. Все что он делает это получает шутку. Вот и все.

```
interface CloudDataSource : JokeDataFetcher
```

Теперь перепишем реализацию

```
class BaseCloudDataSource(private val service: JokeService) : CloudDataSource {  
    override suspend fun getJoke(): JokeDataModel {  
        try {  
            return service.getJoke().execute().body()!!.to()  
        } catch (e: Exception) {  
            if (e is UnknownHostException) {  
                throw NoConnectionException()  
            } else {  
                throw ServiceUnavailableException()  
            }  
        }  
    }  
}
```

Здесь мы переписали следующим образом – получаем и мапим и если ошибка, то мапим ошибку из низкоуровневой – ошибки дата слоя, к ошибкам бизнес логики. Да, это абсолютно нормальный код. Ловим ошибку низкого уровня и бросаем ошибку высокого уровня. Вот:



The screenshot shows an IDE with a project structure on the left and code on the right. The project structure includes folders for 'jokeapp', 'core', 'data', and 'domain', with a file 'Exceptions.kt' selected under 'domain'. The code on the right defines two exception classes: 'NoConnectionException' and 'ServiceUnavailableException', both inheriting from 'IOException'.

```
/**  
 * @author Asutiyah on 19.08.2021  
 */  
class NoConnectionException : IOException()  
class ServiceUnavailableException : IOException()
```

Да, я создал пакет domain слоя и положил туда ошибки бизнес логики. Я хочу обрабатывать ошибку с нормальным названием NoConnectionException нежели UnknownHostException или другую. Мы всегда можем переписать датасорс, но ошибки бизнес слоя будут теми же.

Теперь же перепишем наш кешдатасорс и создадим еще одну ошибку бизнес слоя для него

```
class BaseCachedDataSource(private val realmProvider: RealmProvider) : CacheDataSource {  
    override suspend fun getJoke(): JokeDataModel {  
        realmProvider.provide().use { it: Realm  
            val jokes = it.where(JokeRealmModel::class.java).findAll()  
            if (jokes.isEmpty())  
                throw NoCachedJokesException()  
            else  
                return jokes.random().to()  
        }  
    }  
}
```

Если пусто в кеше, просто бросим исключение. Его поймает на этапе мапинга данных в бизнес слое. Кстати, заметьте, что второй метод не поменялся вовсе.

```
class ServiceUnavailableException : IOException()  
class NoCachedJokesException : IOException()
```

Идем дальше и перепишем наш репозиторий с учетом всех этих изменений

```
interface JokeRepository {
    suspend fun getJoke(): JokeDataModel
    suspend fun changeJokeStatus(): JokeDataModel?
    fun chooseDataSource(cached: Boolean)
}
```

Теперь все выглядит правильно. Репозиторий датаслоя работает с данными датаслоя. Класс:

```
class BaseJokeRepository(
    private val cacheDataSource: CacheDataSource,
    private val cloudDataSource: CloudDataSource,
    private val cachedJoke: CachedJoke
) : JokeRepository {
```

Те классы для мапинга результата больше не нужны. Правильный репозиторий работает лишь с датасорсами и с кешем, хотя CachedJoke является таким же датасорсом, он сохраняет шутку. Т.е. это мутабельный датасорс на самом деле (его тоже перепишем, возможно).

```
interface ChangeJoke {
    suspend fun change(changeJokeStatus: ChangeJokeStatus): JokeDataModel?
}
```

Итак, поменяли метод у интерфейса и можно переписывать наш репозиторий

```
class BaseJokeRepository(...) : JokeRepository {
    private var currentDataSource: JokeDataFetcher = cloudDataSource
    override fun chooseDataSource(cached: Boolean) {
        currentDataSource = if (cached) cacheDataSource else cloudDataSource
    }
    override suspend fun getJoke(): JokeDataModel = withContext(Dispatchers.IO) { this: CoroutineScope
        try {
            val joke = currentDataSource.getJoke()
            cachedJoke.saveJoke(joke)
            return@withContext joke
        } catch (e: Exception) {
            cachedJoke.clear()
            throw e
        }
    }
    override suspend fun changeJokeStatus(): JokeDataModel? = cachedJoke.change(cacheDataSource)
}
```

Выбор датасорса теперь правильный – мы реально меняем датасорсы, а не другие классы. Получение данных тоже теперь прекрасно выглядит. Помните какие раньше были классы целые? ResultHandler с 2 реализациями где было дублирование кода. Теперь все красиво. Получаем шутку, если успешно, то кешируем, если нет и у нас ошибка, то подчищаем кеш и отдаем ошибку далее вверх. Да, мне все еще не нравится то, что мы работаем с нулабл возвращаемым типом при изменении статуса шутки и мы придумаем как пофиксить. А пока давайте уже напишем наш интерактор!


```
interface JokeInteractor {

    suspend fun getJoke() : Joke

    suspend fun changeFavorites() : Joke

    suspend fun getFavoriteJokes(favorites: Boolean)

}
```

Итак, у нас 3 простых метода – получить ошибку, поменять избранные (здесь надо бы 2 метода – добавить и убрать) и получать лишь избранные шутки. Как видите работаем лишь с данными бизнес логики. Здесь вы не увидите слово кеш, потому что это понятия датаслая.

В идеале код вашего интерактора должен суметь читать неайтишник и понять суть сразу.

Давайте напишем реализацию интерактора, но для начала перепишем класс шутки

```
sealed class Joke : Mapper<JokeUiModel> {
    class Success(
        | private val text: String,
        | private val punchline: String,
        | private val favorite: Boolean
    ) : Joke() {
        override fun to(): JokeUiModel {
            return if (favorite) {
                FavoriteJokeUiModel(text, punchline)
            } else {
                BaseJokeUiModel(text, punchline)
            }
        }
    }

    class Failed(private val text: String) : Joke() {
        override fun to(): JokeUiModel {
            return FailedJokeUiModel(text)
        }
    }
}
```

Заметьте, у нас 2 вида шутки, удачная, которая мапится к простой юай модели и избранной и ошибка которая так же мапится к ошибке. Теперь уже напишем наш класс интерактора

И да, надеюсь вы не забыли переместить класс Joke в пакет domain

```
class BaseJokeInteractor(
    private val repository: JokeRepository,
    private val resourceManager: ResourceManager
) : JokeInteractor {

    override suspend fun getJoke(): Joke {
        return try {
            Joke.Success(repository.getJoke().text, repository.getJoke().punchline, favorite: false)
        } catch (e: Exception) {
            val message = when (e) {
                is NoConnectionException -> NoConnection(resourceManager).getMessage()
                is NoCachedJokesException -> NoCachedJokes(resourceManager).getMessage()
                is ServiceUnavailableException -> ServiceUnavailable(resourceManager).getMessage()
                else -> resourceManager.getString(R.string.generic_fail_message)
            }
            Joke.Failed(message)
        }
    }
}
```

Я на время открыл поля класса чтобы дописать метод. После придумаем способ мапинга. Вы посмотрите на метод. Трай кетч – мы смотрим на удобочитаемые названия исключений и получаем текст ошибки прямо из ресурсов через классы. Оставим пока старые классы и пойдем дальше.

Все же сложно писать по чистой архитектуре если ты начал со слоя данных. Давайте пойдем от слоя презентации.

```
class BaseViewModel(
    private val interactor: JokeInteractor,
    private val communication: Communication,
    private val dispatcher: CoroutineDispatcher = Dispatchers.Main
) : ViewModel() {

    fun getJoke() = viewModelScope.launch(dispatcher) { this: CoroutineScope
        communication.showState(State.Progress)
        interactor.getJoke().to().show(communication)
    }
}
```

Наша вьюмодель должна работать с интерактором, а не с репозиторием. Запомните это. Презентационный слой работает с домейн слоем, а бизнес логика со слоем данных.

Итак, получаем шутку – меняем стейт на прогресс и далее получаем шутку, мапим к юай и показываем через комуникатор стейт. Проблема с изменением статуса шутки в том, что у нас 2 стейта на юай – прогресс и непрогресс. У нас нет третьего стейта чтобы в этом случае не обрабатывать клик на изображение. Кто-то может сказать – но у нас нет изображения для случая ошибки, у нас там 0. Но это не значит что завтра мы не захотим поменять на иконку и нам не нужно завязываться на цифру 0 при проверке и тем более не нужно использовать нулабл переменные. Давайте добавим стейт и пофиксим JokeUiModel

И здесь мы получаем каскад наследования. Failed : Initial : State. Я бы не хотел чтобы стейт ошибки зависел от другого стейта. Нам нужно переписать это чтобы было независимо

```
open class Initial(private val text: String, @DrawableRes private val id: Int) : State() {
    override fun show(progress: ShowView, button: EnableView) {...}
    override fun show(textView: ShowText, imageButton: ShowImage) {...}
}

class Failed(private val text: String, @DrawableRes private val id: Int) : Initial(text, id)
```

Сделаем абстрактный класс чтобы никто не использовал его и 2 независимых наследника

```
abstract class Info(private val text: String, @DrawableRes private val id: Int) : State() {
    override fun show(progress: ShowView, button: EnableView) {...}
    override fun show(textView: ShowText, imageButton: ShowImage) {...}
}

class Initial(text: String, @DrawableRes private val id: Int) : Info(text, id)
class Failed(text: String, @DrawableRes private val id: Int) : Info(text, id)
```

И теперь уже поменяем JokeUiModel

```
class FailedJokeUiModel(private val text: String) : JokeUiModel(text, punchline: "") {
    override fun text() = text
    override fun getIconResId() = 0
    override fun show(communication: Communication) = communication.showState(
        BaseViewModel.State.Failed(text(), getIconResId())
    )
}

abstract class JokeUiModel(private val text: String, private val punchline: String) {
    protected open fun text() = "$text\n$punchline"
    @DrawableRes
    protected abstract fun getIconResId(): Int
    open fun show(communication: Communication) = communication.showState(
        BaseViewModel.State.Initial(text(), getIconResId())
    )
}
```

И теперь для стейта ошибки будет модель ошибки. Можно будет спокойно менять 0 на иконку ошибки и никто не будет на нее кликать и менять состояние шутки, которой нет.

Вернемся же в vm и пофикси́м метод смены статуса

Но перед этим упростим нам жизнь и перепишем класс стейта так чтобы легко можно было сравнить его. Создаем 3 константы и метод сравнения типа. Снаружи нужно будет просто сравнить стейт. В конкретных классах стейта просто пишем какой тип и все должно работать.

```
class Initial(text: String, @DrawableRes private val id: Int) : Info(text, id) {
    override val type = INITIAL
}

class Failed(text: String, @DrawableRes private val id: Int) : Info(text, id) {
    override val type = FAILED
}
```

```
sealed class State {
    protected abstract val type: Int
    companion object {
        const val INITIAL = 0
        const val PROGRESS = 1
        const val FAILED = 2
    }
    fun isType(type: Int): Boolean = this.type == type
}
```

Теперь можем написать метод в комуникаторе для проверки состояния

```
interface Communication {
    fun showState(state: BaseViewModel.State)
    fun observe(owner: LifecycleOwner, observer: Observer<BaseViewModel.State>)
    fun isState(type: Int): Boolean
}
```

Передадим ему тип в вьюмодели и проверим что это успешный стейт. И так выглядит класс

```
class BaseCommunication : Communication {
    private val liveData = MutableLiveData<BaseViewModel.State>()
    override fun isState(type: Int): Boolean {
        return liveData.value?.isType(type) ?: false
    }
}
```

Мы не нарушили инкапсуляцию как видите, ливдата все так же приватная. Но суть в том, что у нее может и не быть значения. Ведь если вы не успели еще засетить значение, то там нол. Именно поэтому мы воспользуемся Элвис оператором (да, прическа как у Элвис Пресли) и отдадим false если value == null.

Не передавайте null в качестве аргументов и не возвращайте null. Так же и с nullable аргументами и возвращаемым типом. Никаких null. Все проверки вовне.

И наконец напишем метод смены статуса в вьюмодели

```
fun changeJokeStatus() = viewModelScope.launch(dispatcher) { this: CoroutineScope
    if (communication.isState(State.INITIAL))
        interactor.changeFavorites().to().show(communication)
}
```

Выглядит прекрасно. Мы сравнили текущий стейт и поменяли статус. Теперь можно написать реализацию в интеракторе. Изменение статуса шутки может бросить исключение что больше нет избранных шуток, поэтому мы напишем трай кетч. Опять.

Но подождите, мы ведь точно знаем что в одном методе будет один тип ошибок, а во втором второй, зачем мы написали все 4 варианта и при этом еще и дублировали все во втором методе? Действительно, нам нужен некий класс который предоставит нужный текст ошибки и вроде как мы уже писали такой интерфейс, неправда-ли?

```

override suspend fun changeFavorites(): Joke {
    return try {
        val joke = repository.changeJokeStatus()
        Joke.Success(joke.text, joke.punchline)
    } catch (e: Exception) {
        val message = when (e) {
            is NoConnectionException -> NoConnection(resourceManager).getMessage()
            is NoCachedJokesException -> NoCachedJokes(resourceManager).getMessage()
            is ServiceUnavailableException -> ServiceUnavailable(resourceManager).getMessage()
            else -> resourceManager.getString(R.string.generic_fail_message)
        }
        Joke.Failed(message)
    }
}

```

Да, у нас есть интерфейс от которого наследуются все другие ошибки. Напишем фабрику

```

class Failed(private val failure: JokeFailure) : Joke() {
    override fun to(): JokeUiModel {
        return FailedJokeUiModel(failure.getMessage())
    }
}

```

И поменяем код в интеракторе аля мапер/фабрика

```

class GenericError(private val resourceManager: ResourceManager) : JokeFailure {
    override fun getMessage() = resourceManager.getString(R.string.generic_fail_message)
}

```

И вынесем уже when в класс

```

interface JokeFailureHandler {
    fun handle(e: Exception): JokeFailure
}

class JokeFailureFactory(private val resourceManager: ResourceManager) : JokeFailureHandler {
    override fun handle(e: Exception): JokeFailure {
        return when (e) {
            is NoConnectionException -> NoConnection(resourceManager)
            is NoCachedJokesException -> NoCachedJokes(resourceManager)
            is ServiceUnavailableException -> ServiceUnavailable(resourceManager)
            else -> GenericError(resourceManager)
        }
    }
}

```

Теперь перепишем интерактор чтобы все это было намного прекраснее

Теперь у нас есть некий мапер/фабрика для ошибок. Нам не нужно беспокоиться и думать каждый раз какой метод какие ошибки выкинет. У нас есть класс который займется этим.

Теперь можем пойти дальше и дописать последний метод в интеракторе. Он намного проще.

```

class BaseJokeInteractor(
    private val repository: JokeRepository,
    private val jokeFailureHandler: JokeFailureHandler,
) : JokeInteractor {

    override suspend fun getJoke(): Joke {
        return try {
            Joke.Success(repository.getJoke().text, repository.getJoke().punchline, favorite: false)
        } catch (e: Exception) {
            Joke.Failed(jokeFailureHandler.handle(e))
        }
    }
}

```

В нем не нужно ничего проверять, просто вызываем аналогичный метод у репозитория.

```

override fun getFavoriteJokes(favorites: Boolean) {
    repository.chooseDataSource(favorites)
}

```

Итак, в интеракторе написали методы, теперь нужно все же дописать и в репозитории

Немного пофиксим дата модель

```

class JokeDataModel(
    private val id: Int,
    val text: String,
    val punchline: String,
    val cached: Boolean = false
) : ChangeJoke {

    override suspend fun change(changeJokeStatus: ChangeJokeStatus) =
        changeJokeStatus.addOrRemove(id, joke: this)

    fun toRealm() = JokeRealmModel().also { joke ->
        joke.id = id
        joke.text = text
        joke.punchLine = punchline
    }

    fun changeCached(cached: Boolean): JokeDataModel {
        return JokeDataModel(id, text, punchline, cached)
    }
}

```

Нам нужны методы создания реалм модели и метод клонирования себя с другим статусом. Заметьте, я добавил флаг есть ли в кеше или нет. Паблик поля мы пофиксим позже. А пока посмотрим на метод в кешдатасорс. Теперь там не мапим к юай слою сразу, а даем датамодель и после уже смапим к юай слою

```

override suspend fun addOrRemove(id: Int, joke: JokeDataModel): JokeDataModel =
    withContext(Dispatchers.IO) { this: CoroutineScope
        realmProvider.provide().use { it: Realm
            val jokeRealm =
                it.where(JokeRealmModel::class.java).equalTo(fieldName: "id", id).findFirst()
            return@withContext if (jokeRealm == null) {
                it.executeTransaction { transaction ->
                    val newJoke = joke.toRealm()
                    transaction.insert(newJoke)
                }
                joke.changeCached(cached: true)
            } else {
                it.executeTransaction { it: Realm
                    jokeRealm.deleteFromRealm()
                }
                joke.changeCached(cached: false)
            }
        }
    }

```

Мы получаем на вход датамодель, пишем в бд преобразовав к реалм модели и отдаем обратно ее же но с другим статусом. Хотя можно было бы сделать и без аргумента. Просто сменить текущий статус на !себя же. Но давайте вернемся к закешированному. У нас там был null.

```

class BaseCachedJoke : CachedJoke {
    private var cached: ChangeJoke = ChangeJoke.Empty()

    override fun saveJoke(joke: JokeDataModel) {
        cached = joke
    }

    override fun clear() {
        cached = ChangeJoke.Empty()
    }

    override suspend fun change(changeJokeStatus: ChangeJokeStatus): JokeDataModel {
        return cached.change(changeJokeStatus)
    }
}

```

Теперь у нас нет нулабл поля, я создал пустую реализацию и когда у нее вызывается метод то возвращается пустота. Посмотрите на это

```

interface ChangeJoke {
    suspend fun change(changeJokeStatus: ChangeJokeStatus): JokeDataModel
}

class Empty : ChangeJoke {
    override suspend fun change(changeJokeStatus: ChangeJokeStatus): JokeDataModel {
        return JokeDataModel(id: 0, text: "", punchline: "")
    }
}

```

Мы знаем, что из юай слоя к нам придет непустая шутка и мы будем менять правильный инстанс. Но эта пустая реализация нужна просто чтобы не использовать null. Так намного приятнее работать. Согласитесь. Можно еще и в JokeDataModel сделать Empty и будет совсем красиво. Но давайте проверим теперь весь наш код и выясним где еще остались недочеты.


```

open class JokeRealmModel : RealmObject(), Mapper<JokeDataModel> {
    @PrimaryKey
    var id: Int = -1
    var text: String = ""
    var punchLine: String = ""

    override fun to() = JokeDataModel(id, text, punchLine, cached: true)
}

```

Мы никак не используем тип шутки (type) и потому убрали.

Кстати там где у нас пустая реализация можно и вовсе бросить исключение, мы ведь знаем что этот код не будет вызван и можно перестраховаться таким образом.

```

class Empty : ChangeJoke {
    override suspend fun change(changeJokeStatus: ChangeJokeStatus): JokeDataModel {
        throw IllegalStateException("empty change joke called")
    }
}

```

И давайте в конце концов пофикси́м аппликейшн класс чтобы все работало

```

val cacheDataSource = BaseCachedDataSource(BaseRealmProvider())
val resourceManager = BaseResourceManager(context: this)
val cloudDataSource = BaseCloudDataSource(retrofit.create(JokeService::class.java))
val repository = BaseJokeRepository(cacheDataSource, cloudDataSource, BaseCachedJoke())
val interactor = BaseJokeInteractor(repository, JokeFailureFactory(resourceManager))
viewModel = BaseViewModel(interactor, BaseCommunication())

```

Теперь выглядит намного лучше, без супер вложенности. Не более 3 аргументов у каждого.

Надеюсь все понимают что юнит тесты тоже придется пофиксить, но оставляю это вам.

Запустим код? Нужно проверить что мы ничего не сломали. Кстати да, по-хорошему нужны были юай тесты перед рефакторингом. Но я решил рассказать про клин сперва. Мы всегда успеем отрефакторить чужой код.

Кстати, у вас может быть проблема из-за того, что мы удалили поле тип у реалм модели. Если так, то просто удалите приложение с эмулятора/девайса и запустите с нуля. О миграциях реалм модели поговорим позже.

У нас ошибка. В одном месте забыли указать кеш и там был хардкод false в интеракторе

```

override suspend fun getJoke(): Joke {
    return try {
        val joke = repository.getJoke()
        Joke.Success(joke.text, joke.punchline, favorite: false)
    } catch (e: Exception) {
        Joke.Failure(e.message)
    }
}

```

Легко пофиксить. Выделяем переменную и отдаем поле в конструктор. Сейчас проверим работоспособность и пофикси́м инкапсуляцию.

Да, все в порядке. Теперь можем написать метод у класса вместо предоставления доступа к полям и все будет хорошо.

```
class JokeDataModel(
    private val id: Int,
    private val text: String,
    private val punchline: String,
    private val cached: Boolean = false
) : ChangeJoke {

    fun toJoke() = Joke.Success(text, punchline, cached)
```

И пофиксим в интеракторе оба метода

```
class BaseJokeInteractor(
    private val repository: JokeRepository,
    private val jokeFailureHandler: JokeFailureHandler,
) : JokeInteractor {

    override suspend fun getJoke(): Joke {
        return try {
            repository.getJoke().toJoke()
        } catch (e: Exception) {
            jokeFailureHandler.handle(e)
        }
    }
```

На самом деле должен быть мапер от одного объекта к другому и маперы должны передаваться в конструктор интерактора.

Давайте пофиксим наконец это. Напишем интерфейс мапера под класс JokeDataModel и 2 реализации для домейн слоя и для реалм

```
interface JokeDataModelMapper<T> {
    fun map(id: Int, text: String, punchline: String, cached: Boolean): T
}

class JokeSuccessMapper : JokeDataModelMapper<Joke.Success> {
    override fun map(id: Int, text: String, punchline: String, cached: Boolean) =
        Joke.Success(text, punchline, cached)
}

class JokeRealmMapper : JokeDataModelMapper<JokeRealmModel> {
    override fun map(id: Int, text: String, punchline: String, cached: Boolean): JokeRealmModel {
        return JokeRealmModel().also { joke ->
            joke.id = id
            joke.text = text
            joke.punchline = punchline
        }
    }
}
```

Вместо 2 некрасивых методов в классе я написал мапер который отдает данные в результат нужного типа и 2 реализации мапера для домейн слоя и для бд. Как теперь выглядит наш класс? Довольно прилично. Посмотрите

```
class JokeDataModel(
    private val id: Int,
    private val text: String,
    private val punchline: String,
    private val cached: Boolean = false
) : ChangeJoke {
    fun <T> map(mapper: JokeDataModelMapper<T>): T {
        return mapper.map(id, text, punchline, cached)
    }
    override suspend fun change(changeJokeStatus: ChangeJokeStatus) = changeJokeStatus.addOrRemove(id)
    fun changeCached(cached: Boolean) = JokeDataModel(id, text, punchline, cached)
}
```

У него 1 метод мапинга данных через мапер и поля приватные. Никто не может узнать об айди или тексте модели напрямую. Все кто захотят использовать мой класс будут вынуждены написать мапер для этого. Просто посмотрите как красив теперь стал интерактор

```
class BaseJokeInteractor(
    private val repository: JokeRepository,
    private val jokeFailureHandler: JokeFailureHandler,
    private val mapper: JokeDataModelMapper<Joke.Success>
) : JokeInteractor {
    override suspend fun getJoke(): Joke {
        return try {
            repository.getJoke().map(mapper)
        }
    }
}
```

Я прокидываю в конструктор мапер для преобразования модели из датаслоя в модель бизнес логики и переиспользую этот мапер в 2 местах. Теперь пофикси́м место где создавали реалм модель.

```
class BaseCachedDataSource(
    private val realmProvider: RealmProvider,
    private val mapper: JokeDataModelMapper<JokeRealmModel>
) : CacheDataSource {
    // ...
}
```

Тоже прокидываем в конструктор чтобы не создавать инстанс мапера каждый раз когда нам нужно будет сохранять шутку в кеш

```
it.executeTransaction { transaction ->
    val newJoke = joke.map(mapper)
    transaction.insert(newJoke)
}
```

И наконец пофиксим аппликейшн класс

```
val cacheDataSource = BaseCachedDataSource(BaseRealmProvider(), JokeRealmMapper())
val resourceManager = BaseResourceManager(context: this)
val cloudDataSource = BaseCloudDataSource(retrofit.create(JokeService::class.java))
val repository = BaseJokeRepository(cacheDataSource, cloudDataSource, BaseCachedJoke())
val interactor = BaseJokeInteractor(repository, JokeFailureFactory(resourceManager), JokeSuccessMapper())
viewModel = BaseViewModel(interactor, BaseCommunication())
```

По сути можно точно так же написать и другие маперы, а не тот мапер который мы изначально написали с некрасивым именем метода to(). Но оставляю это вам.

Главное чтобы вы понимали суть что у нас происходит

На активити нажимаем на кнопку получить шутку

во вьюмодели дергаем метод с корутиной, показываем прогресс и получаем у интерактора шутку

Через трай кетч получаем шутку у репозитория

В репозитории получаем у текущего датасorsa модель шутки и мапим к датаслою , если удачно то сохраняем в кеш объект и возвращаем интерактору

Интерактор мапит модель дата слоя к бизнес сущности через мапер и отдает в вьюмодель успешный ответ.

Далее вьюмодель мапит модель бизнес логики к юай слою и отдает комуникатору стейт

В активити обсервим ливдату из комуникатора и показываем юай

Звучит долго конечно. Но поверьте мне, оно того стоит. Лучше много классов и мало в нем кода нежели много кода в нескольких классах. Давайте посчитаем сколько у нас классов и интерфейсов (в конце приведу все исходники)

1. Mapper
2. BaseCachedDataSource
3. BaseCachedJoke
4. BaseRealmProvider
5. CacheDataSource
6. CachedJoke
7. ChangeJoke
8. ChangeJoke.Empty
9. ChangeJokeStatus
10. JokeRealmModel
11. RealmProvider
12. JokeDataModelMapper
13. JokeRealmMapper
14. JokeSuccessMapper
15. BaseCloudDataSource
16. CloudDataSource
17. JokeServerModel
18. JokeService
19. BaseJokeRepository
20. JokeDataFetcher

21. JokeDataModel
22. JokeRepository
23. BaseJokeInteractor
24. NoConnectionException
25. ServiceUnavailableException
26. NoCachedJokesException
27. Joke
28. Joke.Success
29. Joke.Failed
30. JokeFailureFactory
31. JokeFailureHandler
32. JokeInteractor
33. BaseCommunication
34. BaseViewModel
35. Communication
36. Show
37. ShowText
38. ShowImage
39. ShowView
40. EnableView
41. CorrectTextView
42. CorrectButton
43. CorrectImageButton
44. CorrectProgress
45. JokeFailure
46. BaseJokeFailure
47. NoConnection
48. ServiceUnavailable
49. NoCachedJokes
50. GenericError
51. JokeUiModel
52. FailedJokeUiModel
53. FavoriteJokeUiModel
54. BaseJokeUiModel
55. MainActivity
56. State
57. State.Progress
58. State.Info
59. State.Initial
60. State.Failed
61. JokeApp
62. ResourceManager
63. BaseResourceManager

Ну и конечно же под многие классы нужны юнит тесты плюс юай тесты.

На этом можно сказать что тему чистой архитектуры мы прошли. И кто-то скажет – 63 класса? Серьезно? Просто чтобы показать текст и иконку и хранить локально на мобилке? Да, друг мой. Именно так. И в последующих лекциях я покажу всю мощь чистой архитектуры, покажу что не зря мы столько абстракций создали и написали столько кода.

Далее я прикладываю код всех этих классов

```
interface Mapper<R> {
    fun to(): R
}
```

```
class BaseCachedDataSource(
    private val realmProvider: RealmProvider,
    private val mapper: JokeDataModelMapper<JokeRealmModel>
) : CacheDataSource {
    override suspend fun getJoke(): JokeDataModel {
        realmProvider.provide().use {
            val jokes = it.where(JokeRealmModel::class.java).findAll()
            if (jokes.isEmpty())
                throw NoCachedJokesException()
            else
                return jokes.random().to()
        }
    }
    override suspend fun addOrRemove(id: Int, joke: JokeDataModel):
    JokeDataModel =
        withContext(Dispatchers.IO) {
            realmProvider.provide().use {
                val jokeRealm =
                    it.where(JokeRealmModel::class.java).equalTo("id", id).findFirst()
                return@withContext if (jokeRealm == null) {
                    it.executeTransaction { transaction ->
                        val newJoke = joke.map(mapper)
                        transaction.insert(newJoke)
                    }
                    joke.changeCached(true)
                } else {
                    it.executeTransaction {
                        jokeRealm.deleteFromRealm()
                    }
                    joke.changeCached(false)
                }
            }
        }
    }
}
```

```
class BaseCachedJoke : CachedJoke {
    private var cached: ChangeJoke = ChangeJoke.Empty()
    override fun saveJoke(joke: JokeDataModel) {
        cached = joke
    }
    override fun clear() {
        cached = ChangeJoke.Empty()
    }
    override suspend fun change(changeJokeStatus: ChangeJokeStatus):
    JokeDataModel {
        return cached.change(changeJokeStatus)
    }
}
```

```

    }
}
class BaseRealmProvider : RealmProvider {
    override fun provide(): Realm = Realm.getDefaultInstance()
}

```

```

interface CacheDataSource : JokeDataFetcher, ChangeJokeStatus

```

```

interface CachedJoke : ChangeJoke {
    fun saveJoke(joke: JokeDataModel)
    fun clear()
}

```

```

interface ChangeJoke {
    suspend fun change(changeJokeStatus: ChangeJokeStatus): JokeDataModel
}

```

```

class Empty : ChangeJoke {
    override suspend fun change(changeJokeStatus: ChangeJokeStatus):
JokeDataModel {
        throw IllegalStateException("empty change joke called")
    }
}

```

```

interface ChangeJokeStatus {
    suspend fun addOrRemove(id: Int, joke: JokeDataModel): JokeDataModel
}

```

```

open class JokeRealmModel : RealmObject(), Mapper<JokeDataModel> {
    @PrimaryKey
    var id: Int = -1
    var text: String = ""
    var punchLine: String = ""
    override fun to() = JokeDataModel(id, text, punchLine, true)
}

```

```

interface RealmProvider {
    fun provide(): Realm
}

```

```

interface JokeDataModelMapper<T> {
    fun map(id: Int, text: String, punchline: String, cached: Boolean): T
}

```

```

class JokeRealmMapper : JokeDataModelMapper<JokeRealmModel> {
    override fun map(id: Int, text: String, punchline: String, cached: Boolean) =
        JokeRealmModel().also { joke ->
            joke.id = id
            joke.text = text
            joke.punchLine = punchline
        }
}

```

```
class JokeSuccessMapper : JokeDataModelMapper<Joke.Success> {
    override fun map(id: Int, text: String, punchline: String, cached: Boolean) =
        Joke.Success(text, punchline, cached)
}
```

```
class BaseCloudDataSource(private val service: JokeService) : CloudDataSource {
    override suspend fun getJoke(): JokeDataModel {
        try {
            return service.getJoke().execute().body()!!.to()
        } catch (e: Exception) {
            if (e is UnknownHostException) {
                throw NoConnectionException()
            } else {
                throw ServiceUnavailableException()
            }
        }
    }
}
```

```
interface CloudDataSource : JokeDataFetcher
```

```
data class JokeServerModel(
    @SerializedName("id")
    private val id: Int,
    @SerializedName("setup")
    private val text: String,
    @SerializedName("punchline")
    private val punchline: String
) : Mapper<JokeDataModel> {
    override fun to() = JokeDataModel(id, text, punchline)
}
```

```
interface JokeService {
    @GET("https://official-joke-api.appspot.com/random_joke/")
    fun getJoke() : Call<JokeServerModel>
}
```

```
class BaseJokeRepository(
    private val cacheDataSource: CacheDataSource,
    private val cloudDataSource: CloudDataSource,
    private val cachedJoke: CachedJoke
) : JokeRepository {
    private var currentDataSource: JokeDataFetcher = cloudDataSource
    override fun chooseDataSource(cached: Boolean) {
        currentDataSource = if (cached) cacheDataSource else cloudDataSource
    }
    override suspend fun getJoke(): JokeDataModel =
        withContext(Dispatchers.IO) {
            try {
                val joke = currentDataSource.getJoke()
            }
        }
}
```

```

        cachedJoke.saveJoke(joke)
        return@withContext joke
    } catch (e: Exception) {
        cachedJoke.clear()
        throw e
    }
}

override suspend fun changeJokeStatus(): JokeDataModel =
cachedJoke.change(cacheDataSource)
}

```

```

interface JokeDataFetcher {
    suspend fun getJoke(): JokeDataModel
}

```

```

class JokeDataModel(
    private val id: Int,
    private val text: String,
    private val punchline: String,
    private val cached: Boolean = false
) : ChangeJoke {
    fun <T> map(mapper: JokeDataModelMapper<T>): T {
        return mapper.map(id, text, punchline, cached)
    }

    override suspend fun change(changeJokeStatus: ChangeJokeStatus) =
        changeJokeStatus.addOrRemove(id, this)

    fun changeCached(cached: Boolean) = JokeDataModel(id, text, punchline,
cached)
}

```

```

interface JokeRepository {
    suspend fun getJoke(): JokeDataModel
    suspend fun changeJokeStatus(): JokeDataModel
    fun chooseDataSource(cached: Boolean)
}

```

```

class BaseJokeInteractor(
    private val repository: JokeRepository,
    private val jokeFailureHandler: JokeFailureHandler,
    private val mapper: JokeDataModelMapper<Joke.Success>
) : JokeInteractor {
    override suspend fun getJoke(): Joke {
        return try {
            repository.getJoke().map(mapper)
        } catch (e: Exception) {
            Joke.Failed(jokeFailureHandler.handle(e))
        }
    }

    override suspend fun changeFavorites(): Joke {
        return try {
            repository.changeJokeStatus().map(mapper)
        } catch (e: Exception) {

```



```

        Joke.Failed(jokeFailureHandler.handle(e))
    }
}
override fun getFavoriteJokes(favorites: Boolean) =
    repository.chooseDataSource(favorites)
}

```

```
class NoConnectionException : IOException()
```

```
class ServiceUnavailableException : IOException()
```

```
class NoCachedJokesException : IOException()
```

```
sealed class Joke : Mapper<JokeUiModel> {
    class Success(
        private val text: String,
        private val punchline: String,
        private val favorite: Boolean
    ) : Joke() {
        override fun to(): JokeUiModel {
            return if (favorite) {
                FavoriteJokeUiModel(text, punchline)
            } else {
                BaseJokeUiModel(text, punchline)
            }
        }
    }
    class Failed(private val failure: JokeFailure) : Joke() {
        override fun to(): JokeUiModel {
            return FailedJokeUiModel(failure.getMessage())
        }
    }
}

```

```
class JokeFailureFactory(private val resourceManager: ResourceManager) :
JokeFailureHandler {
    override fun handle(e: Exception) = when (e) {
        is NoConnectionException -> NoConnection(resourceManager)
        is NoCachedJokesException -> NoCachedJokes(resourceManager)
        is ServiceUnavailableException -> ServiceUnavailable(resourceManager)
        else -> GenericError(resourceManager)
    }
}

```

```
interface JokeFailureHandler {
    fun handle(e: Exception): JokeFailure
}

```

```
interface JokeInteractor {
    suspend fun getJoke(): Joke
    suspend fun changeFavorites(): Joke
}

```

```

    fun getFavoriteJokes(favorites: Boolean)
}
class BaseCommunication : Communication {
    private val liveData = MutableLiveData<State>()
    override fun isState(type: Int) = liveData.value?.isType(type) ?: false
    override fun showState(state: State) {
        liveData.value = state
    }
    override fun observe(owner: LifecycleOwner, observer: Observer<State>) =
        liveData.observe(owner, observer)
}

```

```

class BaseViewModel(
    private val interactor: JokeInteractor,
    private val communication: Communication,
    private val dispatcher: CoroutineDispatcher = Dispatchers.Main
) : ViewModel() {
    fun getJoke() = viewModelScope.launch(dispatcher) {
        communication.showState(State.Progress)
        interactor.getJoke().to().show(communication)
    }
    fun changeJokeStatus() = viewModelScope.launch(dispatcher) {
        if (communication.isState(State.INITIAL))
            interactor.changeFavorites().to().show(communication)
    }
    fun chooseFavorites(favorites: Boolean) =
        interactor.getFavoriteJokes(favorites)
    fun observe(owner: LifecycleOwner, observer: Observer<State>) =
        communication.observe(owner, observer)
}

```

```

interface Communication {
    fun showState(state: State)
    fun observe(owner: LifecycleOwner, observer: Observer<State>)
    fun isState(type: Int): Boolean
}

```

```

interface Show<T> {
    fun show(arg: T)
}

```

```

interface ShowText : Show<String>

```

```

interface ShowImage : Show<Int>

```

```

interface ShowView: Show<Boolean>

```

```

interface EnableView {
    fun enable(enable: Boolean)
}

```

```

class CorrectTextView : androidx.appcompat.widget.AppCompatTextView,
ShowText {
    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet) : super(context, attrs)
    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) : super(
        context,
        attrs,
        defStyleAttr
    )
    override fun show(arg: String) {
        text = arg
    }
}

```

```

class CorrectButton : androidx.appcompat.widget.AppCompatButton,
EnableView {
    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet) : super(context, attrs)
    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) : super(
        context,
        attrs,
        defStyleAttr
    )
    override fun enable(enable: Boolean) {
        isEnabled = enable
    }
}

```

```

class CorrectImageButton :
androidx.appcompat.widget.AppCompatImageButton, ShowImage {
    //region constructors
    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet) : super(context, attrs)
    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) : super(
        context,
        attrs,
        defStyleAttr
    )
    //endregion
    override fun show(arg: Int) {
        setImageResource(arg)
    }
}

```

```

class CorrectProgress : ProgressBar, ShowView {
    //region constructors
    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet) : super(context, attrs)
    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) : super(
        context,

```

```
    attrs,  
    defStyleAttr  
)
```

```
//endregion  
override fun show(arg: Boolean) {  
    visibility = if (arg) View.VISIBLE else View.INVISIBLE  
}  
}
```

```
interface JokeFailure {  
    fun getMessage(): String  
}
```

```
abstract class BaseJokeFailure(private val resourceManager:  
ResourceManager) : JokeFailure {  
    @StringRes  
    protected abstract fun getMessageResId(): Int  
    override fun getMessage(): String =  
resourceManager.getString(getMessageResId())  
}
```

```
class NoConnection(resourceManager: ResourceManager) :  
BaseJokeFailure(resourceManager) {  
    override fun getMessageResId() = R.string.no_connection  
}
```

```
class ServiceUnavailable(resourceManager: ResourceManager) :  
BaseJokeFailure(resourceManager) {  
    override fun getMessageResId() = R.string.service_unavailable  
}
```

```
class NoCachedJokes(resourceManager: ResourceManager) :  
BaseJokeFailure(resourceManager) {  
    override fun getMessageResId() = R.string.no_cached_jokes  
}
```

```
class GenericError(resourceManager: ResourceManager) :  
BaseJokeFailure(resourceManager) {  
    override fun getMessageResId() = R.string.generic_fail_message  
}
```

```
class BaseJokeUiModel(text: String, punchline: String) : JokeUiModel(text,  
punchline) {  
    override fun getIconResId() = R.drawable.baseline_favorite_border_24  
}
```

```
class FavoriteJokeUiModel(text: String, punchline: String) : JokeUiModel(text,  
punchline) {  
    override fun getIconResId() = R.drawable.baseline_favorite_24  
}
```

```

class FailedJokeUiModel(private val text: String) : JokeUiModel(text, "") {
    override fun text() = text
    override fun getIconResId() = 0
    override fun show(communication: Communication) =
communication.showState(
    State.Failed(text(), getIconResId())
)
}

```

```

abstract class JokeUiModel(private val text: String, private val punchline:
String) {
    protected open fun text() = "$text\n$punchline"
    @DrawableRes
    protected abstract fun getIconResId(): Int
    open fun show(communication: Communication) =
communication.showState(
    State.Initial(text(), getIconResId())
)
}

```

```

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val viewModel = (application as JokeApp).viewModel
        val button = findViewById<CorrectButton>(R.id.actionButton)
        val progressBar = findViewById<CorrectProgress>(R.id.progressBar)
        val textView = findViewById<CorrectTextView>(R.id.textView)
        val checkBox = findViewById<CheckBox>(R.id.checkBox)
        val changeButton =
findViewById<CorrectImageButton>(R.id.changeButton)
        progressBar.visibility = View.INVISIBLE
        checkBox.setOnCheckedChangeListener { _, isChecked ->
            viewModel.chooseFavorites(isChecked)
        }
        changeButton.setOnClickListener {
            viewModel.changeJokeStatus()
        }
        button.setOnClickListener {
            viewModel.getJoke()
        }
        viewModel.observe(this, { state ->
            state.show(progressBar, button, textView, changeButton)
        })
    }
}

```

```

sealed class State {
    protected abstract val type: Int
    companion object {
        const val INITIAL = 0
        const val PROGRESS = 1
    }
}

```

```

    const val FAILED = 2
}
fun isType(type: Int): Boolean = this.type == type
fun show(
    progress: ShowView,
    button: EnableView,
    textView: ShowText,
    imageButton: ShowImage
) {
    show(progress, button)
    show(textView, imageButton)
}
protected open fun show(progress: ShowView, button: EnableView) {}
protected open fun show(textView: ShowText, imageButton: ShowImage) {}

```

```

object Progress : State() {
    override val type = PROGRESS
    override fun show(progress: ShowView, button: EnableView) {
        progress.show(true)
        button.enable(false)
    }
}

```

```

abstract class Info(private val text: String, @DrawableRes private val id: Int) :
State() {
    override fun show(progress: ShowView, button: EnableView) {
        progress.show(false)
        button.enable(true)
    }
    override fun show(textView: ShowText, imageButton: ShowImage) {
        textView.show(text)
        imageButton.show(id)
    }
}

```

```

class Initial(text: String, @DrawableRes private val id: Int) : Info(text, id) {
    override val type = INITIAL
}

```

```

class Failed(text: String, @DrawableRes private val id: Int) : Info(text, id) {
    override val type = FAILED
}

```

```

class JokeApp : Application() {
    lateinit var viewModel: BaseViewModel
    override fun onCreate() {
        super.onCreate()
        Realm.init(this)
        val retrofit = Retrofit.Builder()
            .baseUrl("https://www.google.com")
            .addConverterFactory(GsonConverterFactory.create())
    }
}

```

```

        .build()
        val cacheDataSource = BaseCachedDataSource(BaseRealmProvider(),
JokeRealmMapper())
        val resourceManager = BaseResourceManager(this)
        val cloudDataSource =
BaseCloudDataSource(retrofit.create(JokeService::class.java))
        val repository = BaseJokeRepository(cacheDataSource, cloudDataSource,
BaseCachedJoke())
        val interactor = BaseJokeInteractor(repository,
JokeFailureFactory(resourceManager), JokeSuccessMapper())
        viewModel = BaseViewModel(interactor, BaseCommunication())
    }
}

interface ResourceManager {
    fun getString(@StringRes stringResId: Int) : String
}

class BaseResourceManager(private val context: Context) : ResourceManager {
    override fun getString(stringResId: Int) = context.getString(stringResId)
}

```

Попробуйте вникнуть в суть чистой архитектуры и когда поймете и примете : писать код будет в удовольствие. В следующей лекции я покажу плюсы чистой архитектуры.