Выделяем ядро

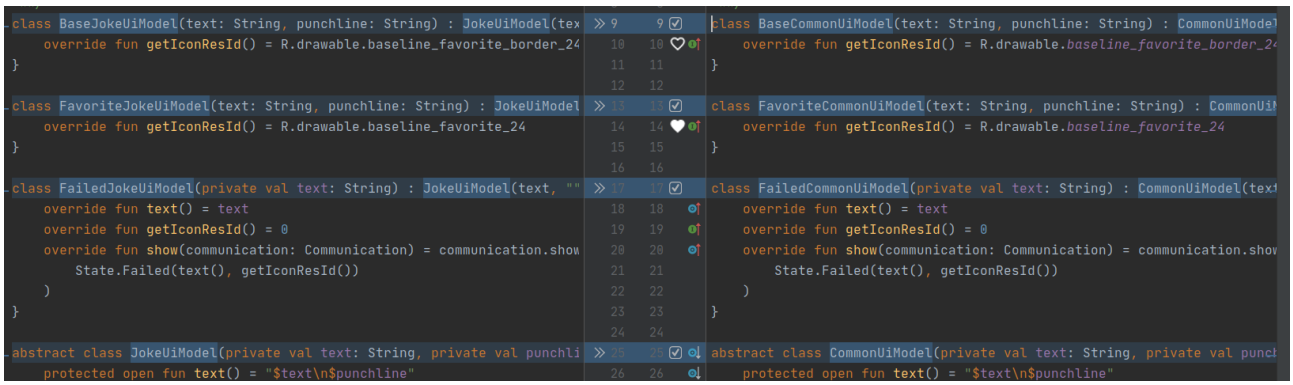
Масштабируем слои

В предыдущей лекции мы написали кастомвью которое имеет в себе 5 других вью и подходит под задачу быстро создать блок данных на юай чтобы юзер мог получить информацию: неважно, шутка или цитата, все будет работать одинаково. Теперь же нам нужно смапить кастомвью в активити с вьюмоделью. У нас уже есть вьюмодель в активити и мы можем добавить столько же кода туда, но я бы очень не хотел этого делать. И в дальнейшем вы увидете что это верное решение. Давайте не нарушать SRP и пусть 1 вьюмодель отвечает за 1 кастомвью. Сегодня эти обе кастомвью лежат в 1 активити, но завтра мы сможем захотеть изменить юай. Так что давайте все же напишем новую вьюмодель, которая делает то же самое. И так как у нас вьюмодель получает в конструктор кроме интерактора еще и комуникатор, то мы сделаем дженериком саму вьюмодель, чтобы легко переиспользовать.

Ведь посмотрите сами, наш комуникатор не зависит от бизнес сущностей. Он просто показывает стейт, каким именно образом неважно. Он работает с интерфейсами и ничего не знает о том, прогресс для шутки он отображает или для цитаты.

Единственное нам придется поменять имена JokeUiModel чтобы они были общими для всех. Итак, приступим. Но не забудьте перед тем как писать код создать новую ветку в гит. Об этом подробно можно узнать на стримах. Ведь в прошлый раз после лекции я запушил код на гитхаб.

Вот так теперь выглядит диф гита



Итак, пойдем дальше. Но перед этим давайте я внесу пару правок в активити. Как сейчас

1 линия для создания вм. ОК. 1 линия для того чтобы найти кастомвью. Ладно. И 3 метода вызываем у самой кастомвью и еще потом обсервим у вьюмодели стейт. Давайте как минимум выделим эти 3 метода в интерфейс и заинкапсулируем в кастомвью.

```
val viewModel = (application as JokeApp).viewModel

val favoriteDataView = findViewById<FavoriteDataView>(R.id.jokeFavoriteDataView)
favoriteDataView.listenChanges { isChecked ->
    viewModel.chooseFavorites(isChecked)
}
favoriteDataView.handleChangeButton {
    viewModel.changeJokeStatus()
}
favoriteDataView.handleActionButton {
    viewModel.getJoke()
}

viewModel.observe( owner: this, { state ->
    favoriteDataView.show(state)
})
```

Я выделю один общий интерфейс с именами методов независящих от того шутка или цитата.

И да, ведь в нашей вьюмодели до сих пор нарушение: паблик методы без оверрайда.

```
interface CommonViewModel {
    fun getItem()
    fun changeItemStatus()
    fun chooseFavorites(favorites: Boolean)
    fun observe(owner: LifecycleOwner, observer: Observer<State>)
}
```

И теперь кастомвью перепишем чтобы у нее был 1 метод – 1 точка изменения

```
val viewModel = (application as JokeApp).viewModel
val favoriteDataView = findViewById<FavoriteDataView>(R.id.jokeFavoriteDataView)
favoriteDataView.linkWith(viewModel)
viewModel.observe( owner: this, { state ->
    favoriteDataView.show(state)
})
```

Стало намного лучше, неправда ли? Но мы должны вьюмодели имплементировать новый интерфейс. Теперь вм выглядит таким образом.

Все что нам мешает сейчас это интерактор. Он конкретный и работает лишь с шутками. Но давайте оставим пока так и напишем просто в активити работу с новой вьюмоделью. Создаем вторую вьюмодель и пишем в активити взаимодействие с ней

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

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

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

    override fun chooseFavorites(favorites: Boolean) = interactor.getFavoriteJokes(favorites)
    override fun observe(owner: LifecycleOwner, observer: Observer<State>) =
        communication.observe(owner, observer)
}
```

Пока что вот так выглядит вьюмодель для цитат.

```
class QuoteViewModel(private val communication: Communication) : ViewModel(), CommonViewModel {

    override fun getItem() {
        TODO( reason: "Not yet implemented")
    }

    override fun changeItemStatus() {
        TODO( reason: "Not yet implemented")
    }

    override fun chooseFavorites(favorites: Boolean) {
        TODO( reason: "Not yet implemented")
    }

    override fun observe(owner: LifecycleOwner, observer: Observer<State>) =
        communication.observe(owner, observer)
}
```

Все что можем сделать без интерактора это наблюдать за стейтом. Напишем в апликейшне код создания вьюмодели и в активити сможем использовать вьюмодель.

Кстати, вам не кажется что надо переименовать класс апликейшна? Мы больше не JokeApp пишем, а нечто где можно смотреть данные и кешировать. Пока не придумал новое название, поэтому оставлю как есть. В дальнейшем переименовать будет несложно

```
class JokeApp : Application() {

    lateinit var viewModel: BaseViewModel
    lateinit var quoteViewModel : QuoteViewModel

    override fun onCreate() {
        super.onCreate()
        existing code
        quoteViewModel = QuoteViewModel(BaseCommunication())
    }
}
```

Теперь как видите очень просто добавить новый блок кода в активити

```
val quoteViewModel = (application as JokeApp).quoteViewModel
val quoteFavoriteDataView = findViewById<FavoriteDataView>(R.id.quoteFavoriteView)
quoteFavoriteDataView.linkWith(quoteViewModel)
quoteViewModel.observe( owner: this, { state ->
    quoteFavoriteDataView.show(state)
})
```

Вуаля! И этот кусок кода мы больше не будем менять. Нам сейчас нужно понять как переписать вьюмодели чтобы они работали одинаково с разными интеракторами. Но ведь сложность именно в том, что интеракторы отдают разные данные. Хотя стоп. Посмотрите на Joke класс. В нем ничего специфичного нет и он наследует мапер к общей юай модели. Так может просто дать ему иное имя? Давайте попробуем

```
sealed class CommonItem : Mapper<CommonUiModel> {
    class Success(
        private val firstText: String,
        private val secondText: String,
        private val favorite: Boolean
    ) : CommonItem() {
        override fun to(): CommonUiModel {
            return if (favorite) {
                FavoriteCommonUiModel(firstText, secondText)
            } else {
                BaseCommonUiModel(firstText, secondText)
            }
        }
    }
    class Failed(private val failure: JokeFailure) : CommonItem() {
        override fun to(): CommonUiModel {
            return FailedCommonUiModel(failure.getMessage())
```

Я также переименовал поля чтобы у нас не было завязки на шутку – текст и панчлайн. Нет, в цитате нет панчлайна. Поэтому текст первый и второй. Максимально абстрактно. Кстати, если так, то и в юай модели надо бы переименовать поля

```
abstract class CommonUiModel(private val first: String, private val second: String) {
    protected open fun text() = "$first\n$second"
```

И давайте уже напишем базовый вм который будет работать с базовым интерактором. Я просто переименовал интерактор шуток и дал общие имена методам.

```
interface CommonInteractor {
    suspend fun getItem(): CommonItem
    suspend fun changeFavorites(): CommonItem
    fun getFavorites(favorites: Boolean)
}
```

И теперь можно удалить новую вьюмодель и написать 1 базовую. Хотя стоп. Наша первая вьюмодель сразу работала с интерфейсом интерактора и ничего переписывать уже не надо.

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

    override fun getItem() {
        viewModelScope.launch(dispatcher) { this: CoroutineScope
            communication.showState(State.Progress)
            interactor.getItem().to().show(communication)
        }
    }
    override fun changeItemStatus() {
        viewModelScope.launch(dispatcher) { this: CoroutineScope
            if (communication.isState(State.INITIAL))
                interactor.changeFavorites().to().show(communication)
        }
    }
    override fun chooseFavorites(favorites: Boolean) = interactor.getFavorites(favorites)
    override fun observe(owner: LifecycleOwner, observer: Observer<State>) =
        communication.observe(owner, observer)
}
```

Теперь пофиксим апликейшн класс чтобы он отдавал этот класс обоим: и шуткам и цитатам.

У меня пока нет интеркатора и я напишу просто анонимный класс.

Но нас ничего не останавливает от того, чтобы написать базовый интерактор. В чем различие между интерактором шутки и цитаты? Если посмотреть внимательно, то никакой. Ну да, они работают с разными репозиториями, но у нас же все по чистоте написано, верно? Значит можно просто поменять название интерфейсам и в апликейшн классе просто дать внутрь нужные классы и все будет работать как ни в чем не бывало.

```
quoteViewModel = BaseViewModel(object: CommonInteractor {
    override suspend fun getItem(): CommonItem {
        TODO( reason: "Not yet implemented")
    }

    override suspend fun changeFavorites(): CommonItem {
        TODO( reason: "Not yet implemented")
    }

    override fun getFavorites(favorites: Boolean) {
        TODO( reason: "Not yet implemented")
    }
}, BaseCommunication())
```

Начнем с самого интерактора, хотя мы уже поменяли интерфейс, теперь нужно поменять класс. Там у нас репозиторий и видите как удобно что мы написали отдельный класс для данных JokeDataModel? Теперь его можно просто переименовать в нечто общее и все будет работать. Поехали!

```
interface ChangeStatus {
    suspend fun addOrRemove(id: Int, joke: CommonDataModel): CommonDataModel
}
```

Простое переименование. Идем дальше

```
interface CommonDataModelMapper<T> {
    fun map(id: Int, first: String, second: String, cached: Boolean): T
}
```

Тоже отрефакторили, идем далее

```
interface ChangeCommonItem {
    suspend fun change(changeStatus: ChangeStatus): CommonDataModel

    class Empty : ChangeCommonItem {
        override suspend fun change(changeStatus: ChangeStatus): CommonDataModel {
            throw IllegalStateException("empty change item called")
```

Не забываем убирать слово joke везде. И теперь наша общая модель стала абсолютно переиспользуемой для всех видов данных : и для шуток и для цитат.

Теперь уже спокойно можно переписать репозиторий, точнее дать новые имена.

```
class CommonDataModel(
    private val id: Int,
    private val firstText: String,
    private val secondText: String,
    private val cached: Boolean = false
) : ChangeCommonItem {
    fun <T> map(mapper: CommonDataModelMapper<T>): T {
        return mapper.map(id, firstText, secondText, cached)
    }
    override suspend fun change(changeStatus: ChangeStatus) = changeStatus.addOrRemove(id, joke: this)
    fun changeCached(cached: Boolean) = CommonDataModel(id, firstText, secondText, cached)
}
```

Вот интерфейс репозитория после изменений

```
interface CommonRepository {
    suspend fun getCommonItem(): CommonDataModel
    suspend fun changeStatus(): CommonDataModel
    fun chooseDataSource(cached: Boolean)
}
```

Осталось переименовать все остальные зависимости интерактора

```
interface FailureHandler {
    fun handle(e: Exception): Failure
}
```

Просто убираем слово Joke отовсюду где имеем общие классы и интерфейсы

```
<string name="no_cached_data">"No favorites! Add one by heart icon "
```

И можно переписать класс который показывает ошибки

```
class NoCachedData(resourceManager: ResourceManager) : BaseFailure(resourceManager) {
    override fun getMessageResId() = R.string.no_cached_data
}
```

Да, мы переписали константу в ресурсах, но в дальнейшем если все же захотим использовать разные строки можно переписать на дженерики. Позже покажу как

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

abstract class BaseFailure(private val resourceManager: ResourceManager) : Failure {
```

И наконец можно дойти до базового интерактора
Как видите я просто убрал мешающее слово и все будет работать как и прежде.

Ладно, кто-то скажет, как же так, ведь различия есть! И да, они реально есть. Они в реалм моделях и серверных моделях, также в сервисе получения. В остальном же все одинаково.

Просто потому что у нас одинаковая логика и там и там и единственное отличие лишь в сервере и в том, что нам нужны разные реалм модели для хранения и тех и других видов.

```kotlin
class BaseInteractor(
    private val repository: CommonRepository,
    private val failureHandler: FailureHandler,
    private val mapper: CommonDataModelMapper<CommonItem.Success>
) : CommonInteractor {
    override suspend fun getItem(): CommonItem {
        return try {
            repository.getCommonItem().map(mapper)
        } catch (e: Exception) {
            CommonItem.Failed(failureHandler.handle(e))
        }
    }
    override suspend fun changeFavorites(): CommonItem {
        return try {
            repository.changeStatus().map(mapper)
        } catch (e: Exception) {
            CommonItem.Failed(failureHandler.handle(e))
        }
    }
    override fun getFavorites(favorites: Boolean) =
        repository.chooseDataSource(favorites)
}
```

Теперь можем переписать апликейшн класс. Но мы забыли переименовать мапер.

```kotlin
class CommonSuccessMapper : CommonDataModelMapper<CommonItem.Success> {
    override fun map(id: Int, first: String, second: String, cached: Boolean) =
        CommonItem.Success(first, second, cached)
}
```

И теперь уже все будет красиво

```kotlin
quoteViewModel = BaseViewModel(BaseInteractor(object: CommonRepository {
    override suspend fun getCommonItem(): CommonDataModel {
        TODO( reason: "Not yet implemented")
    }
    override suspend fun changeStatus(): CommonDataModel {
        TODO( reason: "Not yet implemented")
    }
    override fun chooseDataSource(cached: Boolean) {
        TODO( reason: "Not yet implemented")
    }

},failureHandler, mapper), BaseCommunication())
```

И вот мы дошли до уровня дата слоя. Посмотрим на базовый репозиторий. Что там творится

```kotlin
interface DataFetcher {
    suspend fun getData(): CommonDataModel
}
```

Убрал слово шутка, ведь интерфейс отдает общую модель, значит и имя методу общее
Если вы внимательно посмотрите на базовый репозиторий для шутки, то поймете что можно переименовать все в общее и переиспользовать для цитат. Поехали!

```kotlin
interface CachedData : ChangeCommonItem {
    fun save(data: CommonDataModel)
    fun clear()
}
```

Здесь все теперь красиво и не завязано на шутки

```kotlin
class BaseCachedData : CachedData {
    private var cached: ChangeCommonItem = ChangeCommonItem.Empty()
    override fun save(data: CommonDataModel) {
        cached = data
    }
    override fun clear() {
        cached = ChangeCommonItem.Empty()
    }
    override suspend fun change(changeStatus: ChangeStatus): CommonDataModel {
        return cached.change(changeStatus)
    }
}
```

Главное не забудьте в апликейшн классе не переиспользовать один и тот же инстанс, иначе будет проблема. Для каждого репозитория будет свой инстанс кеш.

```kotlin
class BaseRepository(
    private val cacheDataSource: CacheDataSource,
    private val cloudDataSource: CloudDataSource,
    private val cached: CachedData
) : CommonRepository {
    private var currentDataSource: DataFetcher = cloudDataSource
    override fun chooseDataSource(cached: Boolean) {
        currentDataSource = if (cached) cacheDataSource else cloudDataSource
    }
    override suspend fun getCommonItem(): CommonDataModel = withContext(Dispatchers.IO) {
        try {
            val data = currentDataSource.getData()
            cached.save(data)
            return@withContext data
        } catch (e: Exception) {
            cached.clear()
            throw e
        }
    }
    override suspend fun changeStatus(): CommonDataModel = cached.change(cacheDataSource)
}
```

Как видите ни одного упоминания шутки. Все на интерфейсах и все будет работать как и раньше. Единственное отличие репозитория одного от другого в классах датасорсов.

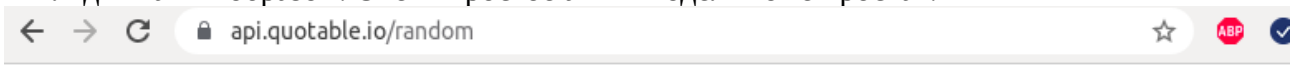
Посмотрим теперь на апликейшн класс, каким он станет сейчас

```
quoteViewModel = BaseViewModel(
    BaseInteractor(
        BaseRepository(
            object : CacheDataSource {
                override suspend fun getData(): CommonDataModel {
                    TODO( reason: "Not yet implemented")
                }
                override suspend fun addOrRemove(
                    id: Int,
                    joke: CommonDataModel
                ): CommonDataModel {
                    TODO( reason: "Not yet implemented")
                }
            },
            object : CloudDataSource {
                override suspend fun getData(): CommonDataModel {
                    TODO( reason: "Not yet implemented")
                }
            }, BaseCachedData()
        ), failureHandler, mapper
    ), BaseCommunication()
)
```

Мы почти заканчиваем. Нам осталось написать 2 класса – для кешДатаСорса и клаудДатаСорса. Давайте начнем с клаудДатаСорса. Мы в одной из лекции написали дженерик класс для работы с разными серверными моделями шуток. Нельзя использовать?

```
abstract class BaseCloudDataSource<T : Mapper<CommonDataModel>> : CloudDataSource {
    protected abstract fun getServerModel(): Call<T>
    override suspend fun getData(): CommonDataModel {
        try {
            return getServerModel().execute().body()!!.to()
        } catch (e: Exception) {
            if (e is UnknownHostException) {
                throw NoConnectionException()
            } else {
                throw ServiceUnavailableException()
```

Я просто убрал слово joke из протектед метода и все осталось как и было. Нам для работы с цитатами нужно просто написать серверную модель которая имплементирует интерфейс мапера и все будет работать. Итак, время написать уже серверную модель цитат. Апи выглядит таким образом. Очень простое апи и модель тоже простая.




```
{"_id":"NeX-2_82sj","tags":["wisdom"],"content":"Knowledge is of no value unless you put it into practice.","author":"Anton Chekhov","authorSlug":"anton-chekhov","length":57,"dateAdded":"2019-06-27","dateModified":"2019-06-27"}
```


Мы возьмем лишь те поля которые нам нужны: айди, контент и автора. Все.

```
class QuoteServerModel(
    @SerializedName( value: "_id")
    private val id: Int,
    @SerializedName( value: "content")
    private val content: String,
    @SerializedName( value: "author")
    private val author: String
) : Mapper<CommonDataModel> {
    override fun to() = CommonDataModel(id, content, author)
}
```

Теперь нам нужно написать сервис

```
interface QuoteService {

    @GET( value: "https://api.quotable.io/random")
    fun getQuote(): Call<QuoteServerModel>
}
```

И теперь напишем клаудДатаСорс

```
class QuoteCloudDataSource(private val service: QuoteService) :
    BaseCloudDataSource<QuoteServerModel>() {
    override fun getServerModel() = service.getQuote()
}
```

И мы можем убрать аноним класс из апликейшна для вьюмодели

```
quoteViewModel = BaseViewModel(
    BaseInteractor(
        BaseRepository(
            object : CacheDataSource {...},
            QuoteCloudDataSource(retrofit.create(QuoteService::class.java)),
            BaseCachedData()
        ), failureHandler, mapper
    ), BaseCommunication()
```

И последнее что нам нужно сделать: написать кешдатасорс для сохранения. Но у нас уже есть кешдатасорс для шуток. Можем ли мы переписать его таким образом чтобы он работал и с цитатами? Давайте попробуем

```
interface ChangeStatus {
    suspend fun addOrRemove(id: Int, model: CommonDataModel): CommonDataModel
}
```

Сначала уберем слово joke. После напишем абстрактный класс для того чтобы он наследовался от реалмобьекта и от мапера. Ниже покажу зачем

```
abstract class DataBaseModel : RealmObject(), Mapper<CommonDataModel>
```

Нам нужно чтобы это был реалм обьект и умел мапиться к обьекту общего вида

```
abstract class BaseCachedDataSource<T: DataBaseModel>(
    private val realmProvider: RealmProvider,
    private val mapper: CommonDataModelMapper<T>
) : CacheDataSource {

    protected abstract val dbClass : Class<T>

    override suspend fun getData(): CommonDataModel {
        realmProvider.provide().use {  it: Realm
            val list = it.where(dbClass).findAll()
            if (list.isEmpty())
                throw NoCachedDataException()
            else
                return list.random().to()
        }
    }
```

Я параметризировал кешдатасорс и сделал его абстрактным. Мы получим тип обьекта по которому ищем и смапим далее. Заметьте, я переименовал класс эксепшна NoCachedDataException вместо того чтобы использовать Joke в имени. Так же я получу в наследнике класс для поиска (да, пока что так, сложно получить из дженерика сейчас, может быть позже придумаем как). И второй метод класса будет выглядеть так.

```
override suspend fun addOrRemove(id: Int, model: CommonDataModel): CommonDataModel =
    withContext(Dispatchers.IO) {  this: CoroutineScope
        realmProvider.provide().use {  it: Realm
            val itemRealm =
                it.where(dbClass).equalTo( fieldName: "id", id).findFirst()
            return@withContext if (itemRealm == null) {
                it.executeTransaction { transaction ->
                    val newData = model.map(mapper)
                    transaction.insert(newData)
                }
                model.changeCached( cached: true)
            } else {
                it.executeTransaction {  it: Realm
                    itemRealm.deleteFromRealm()
                }
                model.changeCached( cached: false)
            }
        }
    }
```

И напишем наконец класс реалм вида для цитаты

```
open class QuoteRealmModel : DataBaseModel() {
    @PrimaryKey
    var id: Int = -1
    var content: String = ""
    var author: String = ""

    override fun to() = CommonDataModel(id, content, author, cached: true)
}
```

И еще нам нужен обратный мапер – из датамодели в реалм модель. Напишем по аналогии с мапером шутки

```
class QuoteRealmMapper : CommonDataModelMapper<QuoteRealmModel> {
    override fun map(id: Int, first: String, second: String, cached: Boolean) =
        QuoteRealmModel().also { quote ->
            quote.id = id
            quote.content = first
            quote.author = second
        }
}
```

И вот теперь можно написать 2 кешдатасорса наследника базового с дженериком

```
class JokeCachedDataSource(realmProvider: RealmProvider, mapper: JokeRealmMapper) :
    BaseCachedDataSource<JokeRealmModel>(realmProvider, mapper) {
    override val dbClass = JokeRealmModel::class.java
}
class QuoteCachedDataSource(realmProvider: RealmProvider, mapper: QuoteRealmMapper) :
    BaseCachedDataSource<QuoteRealmModel>(realmProvider, mapper) {
    override val dbClass = QuoteRealmModel::class.java
}
abstract class BaseCachedDataSource<T : DataBaseModel>(
    private val realmProvider: RealmProvider,
    private val mapper: CommonDataModelMapper<T>
) : CacheDataSource {
```

И все что нам осталось сделать это пофиксить апликейшн класс и можем запускать код

```
val quoteRepository = BaseRepository(
    QuoteCachedDataSource(realmProvider, QuoteRealmMapper()),
    QuoteCloudDataSource(retrofit.create(QuoteService::class.java)),
    BaseCachedData()
)
quoteViewModel = BaseViewModel(
    BaseInteractor(quoteRepository, failureHandler, mapper),
    BaseCommunication()
)
```

Реалм провайдер один, мапер свой. КлаудДатаСорс свой, но ретрофит один ну и кеш свой. В вм уже интерактор базовый со своим репозиторией и общим обработчиком ошибок и общим мапером успешного ответа. Ну что ж. Запустим код?

Не работает! Почему же? Из-за DataBaseModel – он наследует RealmObject но в нем нет полей. Когда библиотека пытается работать с наследниками реалм обьектов у нее ничего не получается сделать с абстрактным классом. Поэтому мы пофиксим кешдатасорс

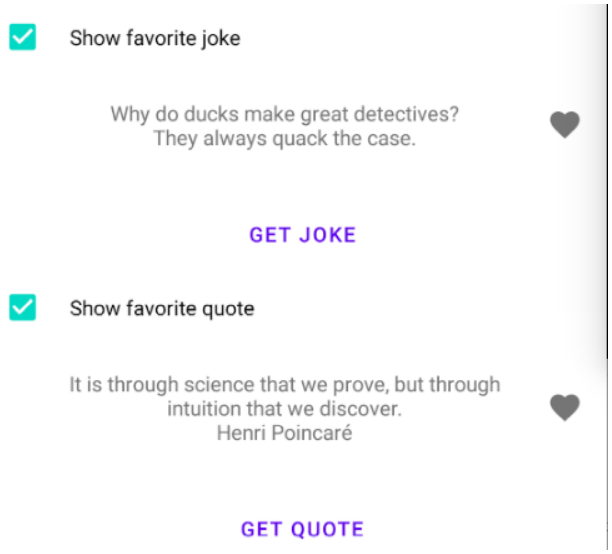
```
abstract class BaseCachedDataSource<T : RealmObject>(
    private val realmProvider: RealmProvider,
    private val mapper: CommonDataModelMapper<T>
) : CacheDataSource {

    protected abstract val dbClass: Class<T>

    override suspend fun getData(): CommonDataModel {
        realmProvider.provide().use { it: Realm
            val list = it.where(dbClass).findAll()
            if (list.isEmpty())
                throw NoCachedDataException()
            else
                return (list.random() as Mapper<CommonDataModel>).to()
```

Что я говорил про каст? Да, что это нехорошо. Надо написать полноценные маперы как мы сделали для успеха и прокинуть в конструктор. Ладно, это потом. Давайте запустим код!

Ай-яй-яй. Я неправильно написал модель для цитат. В модели шуток айди был числом, а здесь у нас строка. Как я узнал об этом? А просто поставил брейкпойнт в BaseCloudDataSource и все. Просто поменяем тип у поля айди в модели цитаты… и стоп. В классе CommonDataModel тип айди у нас числовой, значит так просто не получится. Что же делать? Параметризировать? Или написать код который преобразует строку в число?
Давайте сделаем некрасивое быстрое решение, напишем Any для типа и после пофиксим. Оказывается не все так просто, в классе кешдатасорса возникла проблема. Давайте просто напишем код который сгенерирует число. Знаю, это вовсе неправильно, но пока что так пусть будет.

```
class QuoteServerModel(
    @SerializedName( value: "_id")
    private val id: String,
    @SerializedName( value: "content")
    private val content: String,
    @SerializedName( value: "author")
    private val author: String
) : Mapper<CommonDataModel> {
    override fun to() = CommonDataModel(System.currentTimeMillis().toInt(), content, author)
}
```

Позже пофиксим чтобы было красиво. Запускаем код!
И да, если вы не удалили приложение с девайса у вас опять будет ошибка про то, что реалму нужна миграция, мы же добавили новый класс в схему.
И как видите все работает! Сначала получите шутку, потом получите цитату.
После сохраните шутку и сохраните цитату. Получите новую шутку и цитату. После выберите показывать только избранные и нажмите на кнопки повторно. Можете убрать приложение из памяти и открыть снова и проверить что в бд все записалось.



И на последок давайте пофиксим каст в кешдатасорсе. Напишем интерфейс мапера

```
interface RealmToCommonDataMapper<T : RealmObject> {

    fun map(realmObject: T): CommonDataModel
}

class JokeRealmToCommonMapper : RealmToCommonDataMapper<JokeRealmModel> {
    override fun map(realmObject: JokeRealmModel) =
        CommonDataModel(realmObject.id, realmObject.text, realmObject.punchLine, cached: true)
}

class QuoteRealmToCommonMapper : RealmToCommonDataMapper<QuoteRealmModel> {
    override fun map(realmObject: QuoteRealmModel) =
        CommonDataModel(realmObject.id, realmObject.content, realmObject.author, cached: true)
}
```

И 2 реализации. Теперь можем убрать из серверный моделей наш плохой интерфейс мапера

```
open class JokeRealmModel : RealmObject(){...}

open class QuoteRealmModel : RealmObject() {...}
```

Вуаля! И пофиксим кешдатасорс базовый чтобы работал с мапером.
После пофиксим реализации

```
class JokeCachedDataSource(
    realmProvider: RealmProvider,
    mapper: JokeRealmMapper,
    commonDataMapper: JokeRealmToCommonMapper
) :
    BaseCachedDataSource<JokeRealmModel>(realmProvider, mapper, commonDataMapper) {
    override val dbClass = JokeRealmModel::class.java
}
```

И точно так же пофиксим для цитат. После добавим кода в апликейшн класс и запустим!

```
    private val realmToCommonDataMapper: RealmToCommonDataMapper<T>
: CacheDataSource {

    protected abstract val dbClass: Class<T>

    override suspend fun getData(): CommonDataModel {
        realmProvider.provide().use {  it: Realm
            val list = it.where(dbClass).findAll()
            if (list.isEmpty())
                throw NoCachedDataException()
            else
                return realmToCommonDataMapper.map(list.random())
```

Кстати говоря. Вы заметили что у нас не более 5 зависимостей? Почти везде 3 максимум.

```
class QuoteCachedDataSource(
    realmProvider: RealmProvider,
    mapper: QuoteRealmMapper,
    commonDataMapper: QuoteRealmToCommonMapper
) :
    BaseCachedDataSource<QuoteRealmModel>(realmProvider, mapper, commonDataMapper) {
    override val dbClass = QuoteRealmModel::class.java
}
```

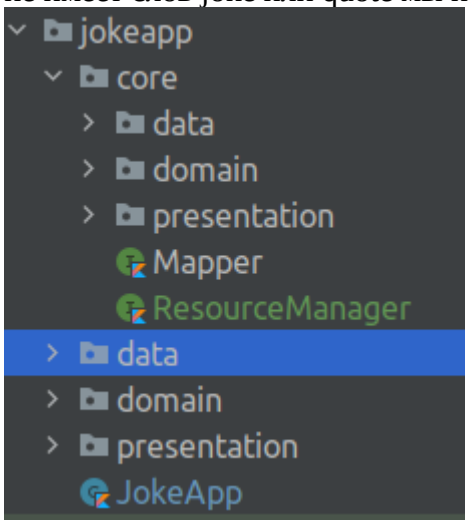
И наконец наш апликейшн класс выглядит так

```
val realmProvider = BaseRealmProvider()
val cacheDataSource = JokeCachedDataSource(realmProvider, JokeRealmMapper(), JokeRealmToCommonMapper())
val cloudDataSource = JokeCloudDataSource(retrofit.create(BaseJokeService::class.java))
```

2 Мапера, для сохранения в реалм и чтения из реалм. Вполне логично.

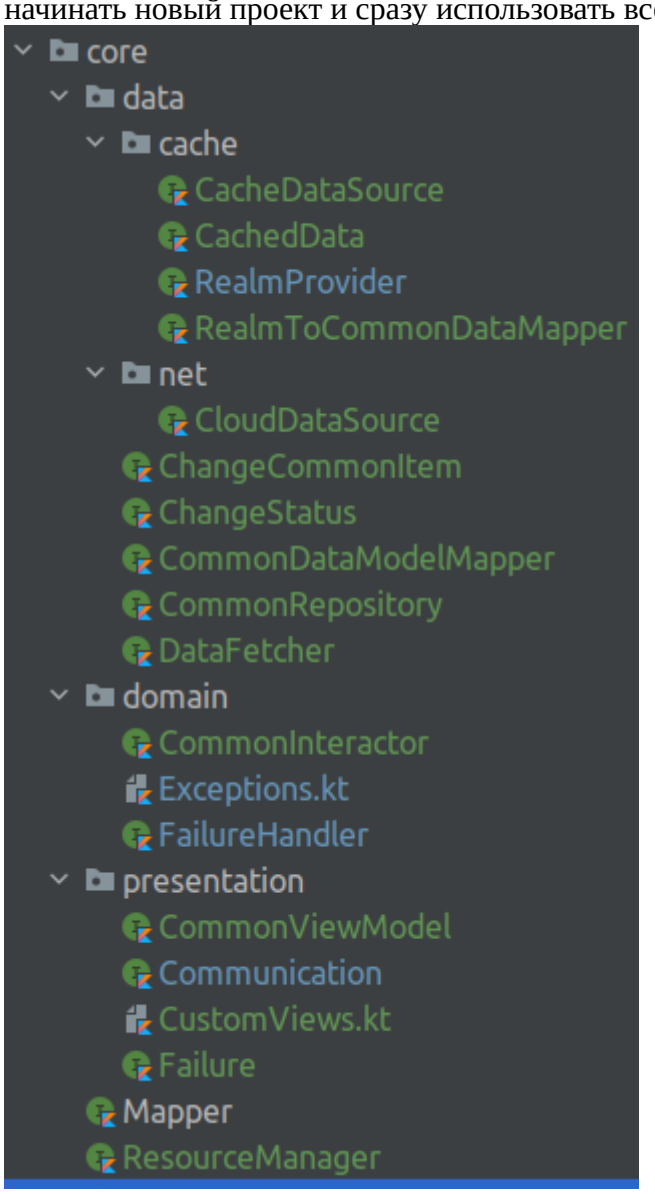
```
val quoteRepository = BaseRepository(
    QuoteCachedDataSource(realmProvider, QuoteRealmMapper(), QuoteRealmToCommonMapper()),
```

Повторно запустим код и проверим что все работает!
Действительно, без нареканий!
И последнее что нам осталось сделать это переместить все файлы в нужные пакеты. Все что не имеет слов joke или quote мы переместим в ядро - core пакет.



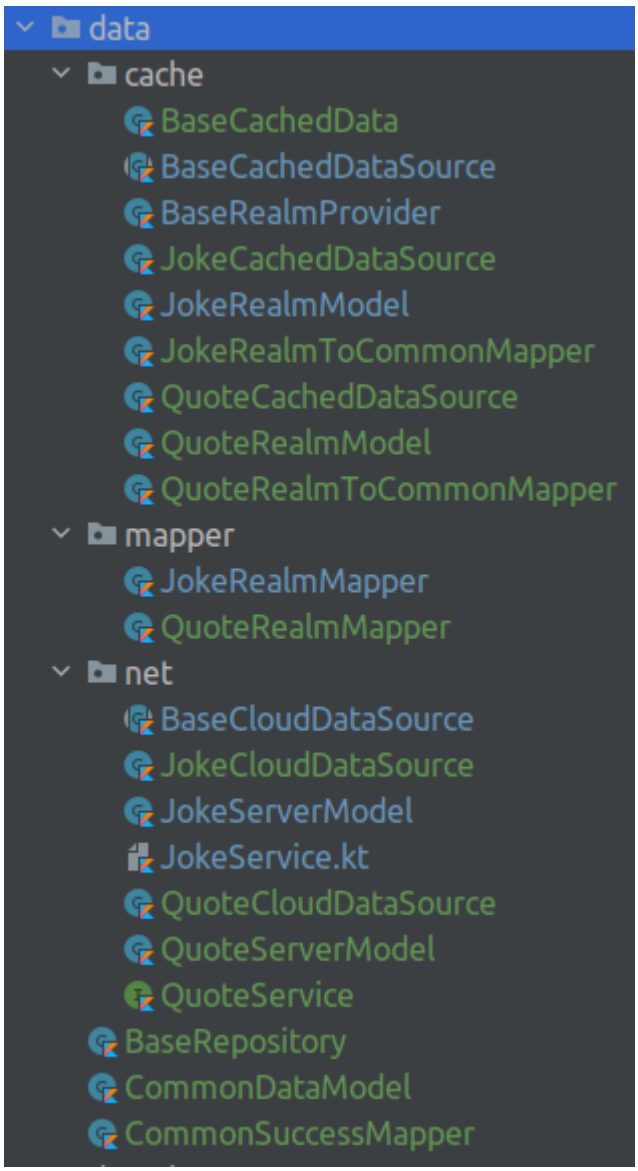
Я создал в ядре еще 3 пакета для слоев. Все что не попало в ядро ушло в пакеты слоев.

Все интерфейсы лежат теперь в ядре, все реализации вне его. Плюс ядра в том, что можно начинать новый проект и сразу использовать все классы из ядра

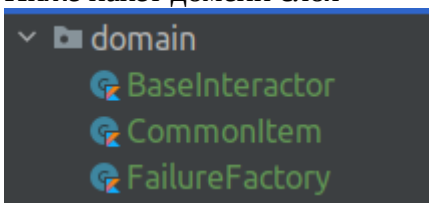


Ниже пакет дата слоя с конкретными реализациями для шуток и цитат

И здесь мы сталкиваемся с тем, что создавая проект мы указали имя проекту JokeApp и вся структура у нас содержит это слово. Мы можем поменять имя приложения из андроид манифеста, но как поменять структуру пакетов? Можно создать новый проект и скопировать все классы туда или же попробовать создать новый пакет с новым именем и переместить классы. Решайте сами как это сделать, но нам нужно новое название проекту и я пока не придумал его.
В последующих лекциях мы добавим новый функционал. Сделаем список избранных, фильтрацию если предоставляет апи параметры запроса и конечно же поиск среди избранных и разделим фичи на разные экраны.



Ниже пакет домейн слоя



Да, здесь всего 3 класса, ну и дальше презентационный слой. Вот и все.