Interface segregation

SOLID: ISP

О чем же этот принцип – сегрегировать интерфейсы. Что значит сегрегировать? Если простыми словами то разделять, уменьшать их. Есть более простое определение принципа – интерфейсы с меньшим количеством методов лучше чем с большим. Но это не отображает саму суть. За одно расскажу вам про принцип YAGNI – You Aren't Going Need It. Это тебе не понадобится. Не надо писать код, который не будет использоваться. И кто-то скажет – как одно сочетается с другим? Давайте посмотрим на нашем проекте JokeApp где мы нарушили принцип ISP. Это метод изменения статуса шутки.

```
class Joke(
    private val id: Int,
    private val type: String,
    private val text: String,
    private val punchline: String,
) {
    suspend fun change(cacheDataSource: CacheDataSource) = cacheDataSource.addOrRemove(id,  joke: this)
```

Вот здесь у нас нарушение в методе change. Почему? А потому что изначально в интерфейсе cacheDataSource был 1 метод, после добавился второй. И что в этом плохого? А то, что в классе Joke мы можем случайно использовать не тот метод. Смотрите сами

```
suspend fun change(cacheDataSource: CacheDataSource) = cacheDataSource.
fun toBaseJoke() = BaseJokeUiModel(text, punchline)
fun toFavoriteJoke() = FavoriteJokeUiModel(text, punchline)
```

addOrRemove(id: Int, joke: Joke)
getJoke()

В классе шутки можно поменять ее с помощью интерфейса, у которого 2 метода видимых и следовательно можно использовать не тот метод. Но среди нас нет дураков! Да-да-да. Разработчик не должен допускать даже теоретической возможности допустить ошибку. Как же нам решить эту проблему? Вот здесь мы и воспользуемся принципом разделения интерфейсов. А что если завтра мы не будем иметь кешдатасорса совсем и удалим этот интерфейс, наш класс Joke не должен пострадать из-за этого. Эти 2 вещи никак не должны быть связаны так жестко. Посмотрим на текущий интерфейс кешдатасорса

```
interface CacheDataSource {
    suspend fun getJoke() : Result<Joke, Unit>
    suspend fun addOrRemove(id: Int, joke: Joke): JokeUiModel
}
```

Как видите в нем 2 метода и сам интерфейс не наследуется ни от кого. Давайте исправим это. Вынесем метод изменения шутки в другой интерфейс (кстати, мы уже делали подобное в лекции про датасорсы в джава разделе).

```
interface ChangeJokeStatus {
    suspend fun addOrRemove(id: Int, joke: Joke): JokeUiModel
}
```

Теперь наш интерфейс делает конкретную вещь и назван соответсвенно – он меняет статус шутки посредством добавления или изменения шутки куда-то. Теперь нам нужно поменять класс Joke.

```
suspend fun change(cacheDataSource: ChangeJokeStatus) = cacheDataSource.
fun toBaseJoke() = BaseJokeUiModel(text, punchline)        addOrRemove(id: Int, joke: Joke)
```

Теперь у программиста, который будет использовать наш интерфейс нет никакой возможности ошибиться. У интерфейса изменения статуса есть лишь 1 метод и программист будет использовать конкретно его. А где сам интерфейс будет реализован неважно уже.

Давайте все же вернемся и пофиксим кешДатаСорс

```
interface CacheDataSource : ChangeJokeStatus {
    suspend fun getJoke() : Result<Joke, Unit>
}
```

И сразу видно то, что у кешдатасорса есть возможность менять статус шутки, плюс свой метод получения шутки. И как видите реализация кешдатасорса не пострадала от этого

```
class BaseCachedDataSource(private val realmProvider: RealmProvider) : CacheDataSource {

    override suspend fun getJoke(): Result<Joke, Unit> {...}

    override suspend fun addOrRemove(id: Int, joke: Joke): JokeUiModel =
        withContext(Dispatchers.IO) {...}
}
```

Давайте подытожим – теперь у нас независимый интерфейс изменения статуса шутки и его можно реализовать вне зависимости от того, есть ли у нас вообще кеш в проекте или нет. Предположим вы меняете статус просто на лету и никуда не сохраняете или сохраняете в свой сервер. Ведь как странно звучало ранее – шутка меняет свой статус посредством кешдатасорса, почему именно его? Шутку нужно уметь менять в статусе через независимый интерфейс, а не через конкретный датасорс. Класс шутки не должен знать вообще про такие вещи как датасорсы. Даже завтра вы напишете свой сервер и захотите сохранять данные на свой сервер то просто унаследуете этот интерфейс кому нужно и все будет работать.

Но я бы хотел пойти еще дальше и заострить ваше внимание на еще одном методе одного интерфейса. Просто посмотрите на эти 2 метода в 2 разных интерфейсах

```
interface CacheDataSource : ChangeJokeStatus {
    suspend fun getJoke() : Result<Joke, Unit>
}
```

```
interface CloudDataSource {
    suspend fun getJoke(): Result<JokeServerModel, ErrorType>
}
```

Ну да, 2 метода в 2 разных интерфейсах названы одинаково, но у них же разный возвращаемый тип, что мы можем поделать? Дженерики! Давайте напишем такой интерфейс

```
interface JokeDataFetcher<S, E> {
    suspend fun getJoke(): Result<S, E>
}
```

И как его переиспользовать? А вот так!

```
interface CloudDataSource : JokeDataFetcher<JokeServerModel, ErrorType>
```

```
interface CacheDataSource : JokeDataFetcher<Joke, Unit>, ChangeJokeStatus
```

Итого, у нас 2 интерфейса, у которых нет своих методов. Красота же ну!

Вот я читаю код – клаудДатаСорс и сразу вижу в той же линии : умеет получить шутку сервер модель или ошибка. КешДатаСорс умеет получать шутку плюс менять статус.

А как там конкретно уже и не важно и не нужно. Можно глянуть, но там и так все ясно. Кстати говоря о том зачем нам писать вдумчивые имена классам и интерфейсам и переменным и функциям и всему остальному. Когда у вас есть нормальные имена у всех то и джавадоки не нужны которые поясняют суть. Теперь можете посмотреть на реализации и увидете что ничего не поменялось!

```
override suspend fun getJoke(): JokeUiModel = withContext(Dispatchers.IO) {
    if (getJokeFromCache) {
        return@withContext when (val result = cacheDataSource.getJoke()) {
            is Result.Success<Joke> -> result.data.let {
                cachedJoke = it
                it.toFavoriteJoke()
            }
            is Result.Error -> {
                cachedJoke = null
                FailedJokeUiModel(noCachedJokes.getMessage())
            }
        }
    } else {
        return@withContext when (val result = cloudDataSource.getJoke()) {
            is Result.Success<JokeServerModel> -> {
                result.data.toJoke().let {
                    cachedJoke = it
                    it.toBaseJoke()
                }
            }
            is Result.Error<ErrorType> -> {
                cachedJoke = null
                val failure = if (result.exception == ErrorType.NO_CONNECTION)
                    noConnection
                else serviceUnavailable
                FailedJokeUiModel(failure.getMessage())
```

И теперь самое интересное : я могу переписать код в модели так чтобы это было красиво. Посмотрите сами как сейчас (картинка выше). И я вынесу результат над ифом

```
private interface ResultHandler<S, E> {
    fun handleResult(result: Result<S, E>): JokeUiModel
}
```

Пусть у меня будет интерфейс для разбора результата дженериком. Далее я напишу базовый абстрактный класс (все это внутри модели)

```
private abstract inner class BaseResultHandler<S, E>
    (private val jokeDataFetcher: JokeDataFetcher<S, E>) : ResultHandler<S, E> {

    suspend fun process(): JokeUiModel {
        return handleResult(jokeDataFetcher.getJoke())
    }
}
```

Так как оба датастора наследуют интерфейс получения шутки то я не смогу даже при желании вызвать другой метод у интерфейса. Он всего один. Надеюсь здесь все понятно. У базового абстрактного класса 1 метод паблик который вызывает метод у наследников с тем аргументом который придет в конструктор. Т.е. базовая логика инкапсулирована здесь. Нам не надо будет писать дублирующийся код. Идем дальше и выносим куски кода из метода в модели в классы

```
private inner class CloudResultHandler(jokeDataFetcher: JokeDataFetcher<JokeServerModel, ErrorType>) :
    BaseResultHandler<JokeServerModel, ErrorType>(jokeDataFetcher) {
    override fun handleResult(result: Result<JokeServerModel, ErrorType>) = when (result) {
        is Result.Success<JokeServerModel> -> {
            result.data.toJoke().let { it: Joke
                cachedJoke = it
                it.toBaseJoke()
            }
        }
        is Result.Error<ErrorType> -> {
            cachedJoke = null
            val failure = if (result.exception == ErrorType.NO_CONNECTION)
                noConnection
            else serviceUnavailable
            FailedJokeUiModel(failure.getMessage())
        }
    }
}
```

Просто создаем наследника от базового и передаем дженерики сервер модель и тип ошибки и просто парсим результат. То же самое напишем и для кешдатасорса. Эти классы просто будут разбирать результат который придет им в метод handleResult и все. Одна ответственность (SOLID: S). Как видите ничего сложного, написали класс и сразу же кусок кода из модели вставили в него. И теперь посмотри на то, ради чего мы все это сделали

```
private inner class CacheResultHandler(jokeDataFetcher: JokeDataFetcher<Joke, Unit>) :
    BaseResultHandler<Joke, Unit>(jokeDataFetcher) {
    override fun handleResult(result: Result<Joke, Unit>) = when (result) {
        is Result.Success<Joke> -> result.data.let {  it: Joke
            cachedJoke = it
            it.toFavoriteJoke()
        }
        is Result.Error -> {
            cachedJoke = null
            FailedJokeUiModel(noCachedJokes.getMessage())
        }
    }
}
```

Вот так выглядит теперь наш метод модели

```
override suspend fun getJoke(): JokeUiModel = withContext(Dispatchers.IO) {
    val resultHandler = if (getJokeFromCache)
        CacheResultHandler(cacheDataSource)
    else
        CloudResultHandler(cloudDataSource)
    return@withContext resultHandler.process()
}
```

Здесь мы теперь просто выбираем обработчика результата с нужным датасорсом. Ведь по сути так и должен выглядеть правильный метод. Выбираем датасорс и обрабатываем результат. Почему у нас в методе должно быть так много линий кода? Ранее было более 25 линий кода. А сейчас всего 5, и те можно написать в 2.

А самое прекрасное то, что у нас написаны были юнит тесты и можно запустить их и проверить что все работает должным образом. Но я все же недоволен тем что мы написали. Ведь каждый раз когда нужно получишь шутку мы будем создавать новые классы. Давайте уберем переменную выбора из класса и перепишем метод выбора датасорса так, чтобы по истине выбирался нужный.

Так же я убрал все обьекты ошибок туда где им место – там где они используются. И теперь мой класс модели выглядит куда лучше. У модели 3 метода значит и видеть я должен 3 метода паблик, никаких приватных методов. То что у нас классы приватные и интерфейсы это тема последующих лекций, мы обязательно вернемся к ним и перепишем все это.

И да, если вы заметили я написал ленивую инициализацию лишь для кеша, хотя мы не знаем будет ли юзер клауд юзать тоже, но пока что так. В действительности ничего страшного если вы вообще не напишете ленивую инициализацию, а сразу создадите хендлеры. Но опять же, мы их все после перепишем, не волнуйтесь об этом. И заметьте что мой текущий хендлер имеет дженерик типами <*, *> что означает “я не знаю какие типы, работай со всеми”.

Код стал намного лучше, согласитесь. Хотя мы опять же допустили 1 ошибку и наш класс хендлера имеет видимый метод кроме process. Давайте удалим ненужный интерфейс.

```
class BaseModel(
    private val cacheDataSource: CacheDataSource,
    cloudDataSource: CloudDataSource,
    private val resourceManager: ResourceManager
) : Model {
    private var cachedJoke: Joke? = null
    private val cacheResultHandler by lazy { CacheResultHandler(cacheDataSource) }
    private val cloudResultHandler = CloudResultHandler(cloudDataSource)

    private var currentResultHandler: BaseResultHandler<*, *> = cloudResultHandler

    override fun chooseDataSource(cached: Boolean) {
        currentResultHandler = if (cached) cacheResultHandler else cloudResultHandler
    }

    override suspend fun getJoke(): JokeUiModel = withContext(Dispatchers.IO) {  this: CoroutineScope
        return@withContext currentResultHandler.process()
    }

    override suspend fun changeJokeStatus(): JokeUiModel? = cachedJoke?.change(cacheDataSource)

    private
}
```

И теперь все выглядит намного лучше

```
private abstract inner class BaseResultHandler<S, E>
    (private val jokeDataFetcher: JokeDataFetcher<S, E>) {

    suspend fun process(): JokeUiModel {
        return handleResult(jokeDataFetcher.getJoke())
    }

    protected abstract fun handleResult(result: Result<S, E>): JokeUiModel
}
```

И все же мы опять создали проблему тем, что внутри модели создаем классы в полях! Это нарушение SOLID L & D. Как же нам это решить? А легко и просто. Все наши приватные классы лишь работают с закешированным обьектом шутки. Давайте просто вынесем в класс и перепишем приватные классы таким образом чтобы все приходило в конструктор.

```
interface CachedJoke {
    fun saveJoke(joke: Joke)
    fun clear()
    suspend fun change(changeJokeStatus: ChangeJokeStatus)
}
```

Да, я вынес интерфейс с 3 методами: сохранить, удалить и использовать. Метод я скопировал из класса Joke. И это опять нарушение SOLID I и поэтому надо переписать код и выделить интерфейс. Сделаем это вот таким образом

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

И наследуем его у класса Joke так же у интерфейса закешированной шутки

```
class Joke(
    private val id: Int,
    private val type: String,
    private val text: String,
    private val punchline: String,
) : ChangeJoke {
    override suspend fun change(changeJokeStatus: ChangeJokeStatus) =
        changeJokeStatus.addOrRemove(id, joke: this)
```

И у интерфейса добавим тот же метод

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

Теперь можем переписать наши классы хендлера и модель. Посмотрите как хорошо будет

```
private inner class CloudResultHandler(
    private val cachedJoke: CachedJoke,
    jokeDataFetcher: JokeDataFetcher<JokeServerModel, ErrorType>,
    private val noConnection: JokeFailure,
    private val serviceUnavailable: JokeFailure
) : BaseResultHandler<JokeServerModel, ErrorType>(jokeDataFetcher) {

    override fun handleResult(result: Result<JokeServerModel, ErrorType>) = when (result) {
        is Result.Success<JokeServerModel> -> {
            result.data.toJoke().let { it: Joke
                cachedJoke.saveJoke(it)
                it.toBaseJoke()
            }
        }
        is Result.Error<ErrorType> -> {
            cachedJoke.clear()
            val failure = if (result.exception == ErrorType.NO_CONNECTION)
                noConnection
            else serviceUnavailable
            FailedJokeUiModel(failure.getMessage())
        }
    }
}
```

Заодно я убрал поля с ошибками и получу их в конструкторе чтобы не нарушать SOLID

И теперь поменяем обработчик результата для клаудДатаСтора

```
private inner class CacheResultHandler(
    private val cachedJoke: CachedJoke,
    jokeDataFetcher: JokeDataFetcher<Joke, Unit>,
    private val noCachedJokes: JokeFailure
) : BaseResultHandler<Joke, Unit>(jokeDataFetcher) {

    override fun handleResult(result: Result<Joke, Unit>) = when (result) {
        is Result.Success<Joke> -> result.data.let {  it: Joke
            cachedJoke.saveJoke(it)
            it.toFavoriteJoke()
        }
        is Result.Error -> {
            cachedJoke.clear()
            FailedJokeUiModel(noCachedJokes.getMessage())
        }
    }
}
```

И тоже здесь убрал низкоуровневый код и инициализацию обьектов. Все в конструкторе.

И наконец пофиксим модель

```
class BaseModel(
    private val cacheDataSource: CacheDataSource,
    private val cacheResultHandler: CacheResultHandler,
    private val cloudResultHandler: CloudResultHandler,
    private val cachedJoke: CachedJoke
) : Model {
    private var currentResultHandler: BaseResultHandler<*, *> = cloudResultHandler

    override fun chooseDataSource(cached: Boolean) {
        currentResultHandler = if (cached) cacheResultHandler else cloudResultHandler
    }

    override suspend fun getJoke(): JokeUiModel = withContext(Dispatchers.IO) {  this: CoroutineScope
        return@withContext currentResultHandler.process()
    }

    override suspend fun changeJokeStatus(): JokeUiModel? = cachedJoke.change(cacheDataSource)
}
```

Красота же ну! Видите как мало кода? Так и должен выглядеть правильный класс. Работа с обьектами, а не низкоуровневый код. Ну кроме 1 условия при выборе хендлера. И да, все вложенные приватные классы я вынес изнутри и сделал публичными. Теперь пофиксим апликейшн класс (и да, тесты тоже надо будет пофиксить). И да, я забыл написать реализацию кеша шутки. Ну это просто

```
class BaseCachedJoke : CachedJoke {
    private var cached: Joke? = null
    override fun saveJoke(joke: Joke) {
        cached = joke
    }

    override fun clear() {
        cached = null
    }

    override suspend fun change(changeJokeStatus: ChangeJokeStatus): JokeUiModel? {
        return cached?.change(changeJokeStatus)
    }
}
```

Теперь можем спокойно дописать апликейшн класс

```
val cachedJoke = BaseCachedJoke()
val cacheDataSource = BaseCachedDataSource(BaseRealmProvider())
val resourceManager = BaseResourceManager( context: this)
viewModel = ViewModel(
    BaseModel(
        cacheDataSource,
        CacheResultHandler(
            cachedJoke,
            cacheDataSource,
            NoCachedJokes(resourceManager)
        ),
        CloudResultHandler(
            cachedJoke,
            BaseCloudDataSource(retrofit.create(JokeService::class.java)),
            NoConnection(resourceManager),
            ServiceUnavailable(resourceManager)
        ),
        cachedJoke
    )
)
```

Да, здесь стало многовато кода, но это проблема которая решается в следующих лекциях.

Запустим код? Проверим что все работает.

И да, все работает ровно так же как и раньше.

Теперь я предлагаю вам самим переписать тесты для модели или же начать писать их если до сих пор не начинали. Наша модель стала проще чем раньше и можно с легкостью написать юнит тесты. Ведь здесь 3 метода и они очень просты.