UiTests, MockDataSource

Как проверить функционал в 1 клик

После предыдущей лекции я немного порефакторил код, чтобы комуникации не были видны снаружи. Для этого я сделал интерфейс, который имплементирует комуникация и также вьюмодель. И при попытке получить данные из комуникаций я отдаю через методы вьюмодели. Весь код доступен на гитхаб, комит называется little refactoring, made core module. Также чтобы не инициализировать по 3 поля я вынес в некий модуль ядра все что может переиспользоваться другими модулями для вьюмоделей

```
class JokesAndQuotesApp : Application() {

    private val viewModelsFactory by lazy {
        ViewModelsFactory(MainModule(coreModule), JokesModule(coreModule), QuotesModule(coreModule))
    }

    private lateinit var coreModule: CoreModule

    override fun onCreate() {
        super.onCreate()
        coreModule = CoreModule( context: this)
    }

    fun <T : ViewModel> get(modelClass: Class<T>, owner: ViewModelStoreOwner): T =
        ViewModelProvider(owner, viewModelsFactory).get(modelClass)
}
```

Вот так выглядит апликейшн класс, я даже сделал приватным фабрику и 1 метод получения вьюмодели.

Теперь же у меня встает такой вопрос,

как мне проверять работоспособность приложения?

Каждый раз запускать проект и кликать по всем кнопкам? А что если в этот момент у меня на сервере произойдет ошибка? Или сервер просто будет недоступен? Как же быть тогда?

Ответ на этот вопрос : UI tests юай тесты! Раньше мы писали юнит тесты на котлин классы и проверяли просто логику кода. Но как мы можем проверить именно юай? Ведь сейчас нужно открыть приложение и самому кликать на вью и проверять глазами. Было бы классно если бы мы могли нажать на 1 кнопку и чтобы кто-то другой проверял все по написанному нами 1 раз сценарию. Для этого есть библиотека Espresso и она на старте проекта уже внедрена в градл.

Но перед тем как писать юай тест нам нужно сделать так, чтобы приложение получало данные не из сети, а из так называемых моков : хардкода если хотите. Ведь мы не можем на юай тесте ждать пока от сервера придет ответ и то не факт что он будет успешен. Мы хотим имитировать сервер, чтобы он отдавал предсказуемый результат каждый раз.

Итак, давайте взглянем на класс CloudDataSource который отдает данные от сервера. У нас получение данных рандомного вида и потому давайте напишем дубликат этого класса, но будем отдавать хардкод обьекты с рандомным айди и текстом

```
class JokeCloudDataSource(private val service: BaseJokeService) :
    BaseCloudDataSource<JokeServerModel, Int>() {
    override fun getServerModel() = service.getJoke()
}
```

Давайте напишем такой же класс, но с префиксом Mock который будет обозначать – (для юай тестов) ненастоящие данные.

```
class MockJokeCloudDataSource() : BaseCloudDataSource<JokeServerModel, Int>() {

    private var id: Int = -1
    override fun getServerModel(): Call<JokeServerModel> {
        return object : SimpleCall<JokeServerModel> {
            override fun execute(): Response<JokeServerModel> {
                ++id
                return Response.success(
                    JokeServerModel(
                        id,
                        type: "mockType",
                        text: "mock text $id",
                        punchline: "mock punchline $id"
```

Здесь я просто инкрементирую айдишник при каждом получении шутки. Плюс у нас не сама модель, а Call потому я написал некий интерфейс SimpleCall в котором все методы которые не нужны в этом случае возвращают пустоту или ошибку.

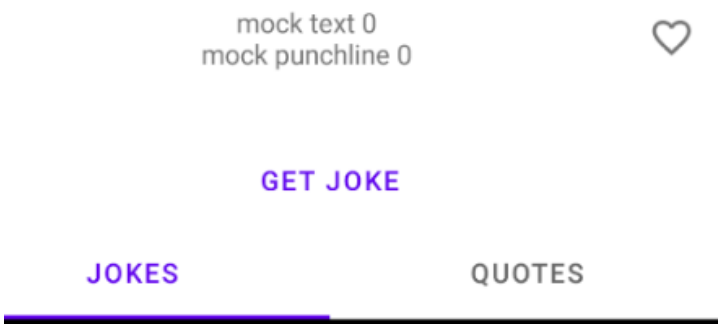
```
interface SimpleCall<T> : Call<T> {
    override fun clone() = throw IllegalStateException("not used")
    override fun enqueue(callback: Callback<T>) = Unit
    override fun isExecuted() = false
    override fun cancel() = Unit
    override fun isCanceled() = false
    override fun request() = throw IllegalStateException("not used")
}
```

Теперь мы можем использовать вместо базовой реализации мок реализацию и проверить что все работает по прежнему. Давайт в модуле шуток заменим инстанс

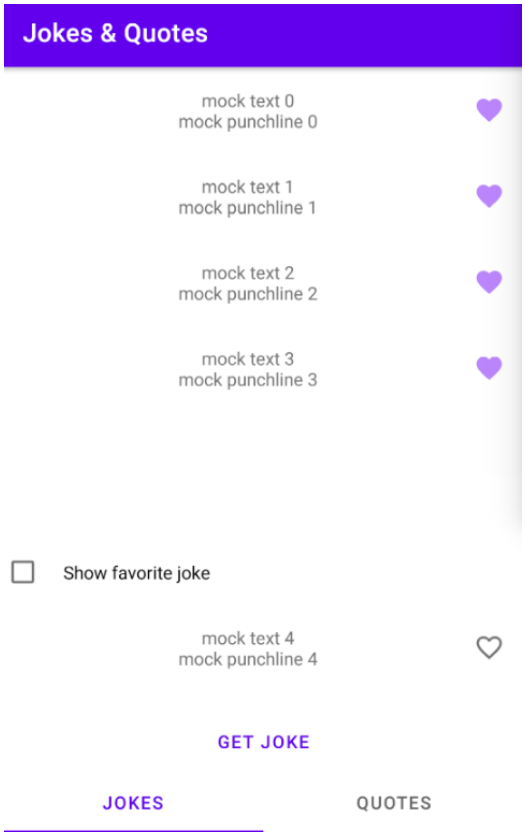
```
    private fun getCloudDataSource() =
        MockJokeCloudDataSource()
//        JokeCloudDataSource(instancesProvider.makeService(BaseJokeService::class.java))
}
```

И запустим проект проверим что все работает.

Да, как видите теперь вместо настоящих шуток от сервера приходят мои мокированные данные. Классно. То же самое можно проделать и с цитатами.

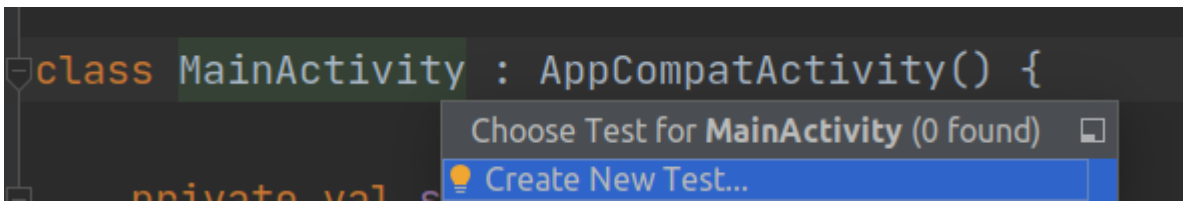


Давайте проверим что добавление в ибранные работает тоже

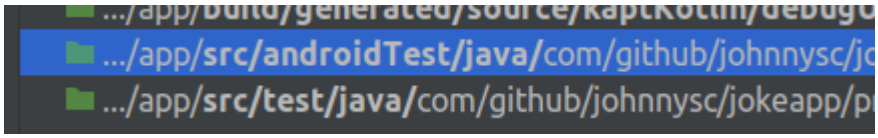


Теперь давайте уже наконец напишем первый юай тест, в котором проверим что при нажатии на GET JOKE приходит шутка в это место. При повторном нажатии заменяется на новую шутку. Так как мы сами написали мок класс то и мы знаем какой будет текст. Ведь мы сможем проверить содержание шутки и по содержанию различать что заменился на другую.

Для того чтобы написать юай тест нужно просто точно так же как и с юнит тестом поставить курсор на имя файла и нажать Ctrl+Shift+T. Только вот мы пишем юай тест на экран. А на какой класс ставить курсор? Давайте на мейн активити



Единственное различие юай теста от обычного юнит теста в том, что мы выбираем пакет androidTest, а не просто test.



Теперь нам нужно написать сам юай тест. Так как это не просто юнит тест нам нужно обозначить его

```
@RunWith(AndroidJUnit4ClassRunner::class)
@LargeTest
class RandomJokesTest {

    @Test
    fun test() {
```

Нам нужен андроидовкий класс раннер и также я обозначил тест как большой, на всякий случай, если вдруг тест действительно будет долгим. В остальном юай тест так же пишется как и юнит тест. Но за одним исключением. У нас не будет ниодного assertEquals потому что мы не работаем с обьектами и не сравниваем их друг с другом. А мы в юай тесте работаем с вью. Именно поэтому в юай тесте суть проверок максимально проста

1. Найти вью : по айди, по тексту, по другому признаку

2. Совершить действие (опционально) (например кликнуть или написать текст если это поле ввода или свайпнуть)

3. Проверить что есть последствие для этой вью (отобразилось, исчезло, поменяло состояние на чек)

Итак, перед тем как написать сам юай тест давайте напишем тесткейс. Что мы делаем и в какой порядке и что ожидаем.

Нажать на кнопку GET JOKE

отображается шутка с тектом mock text 0 mock punchline 0

Нажать на кнопку GET JOKE

отображается шутка с тектом mock text 1 mock punchline 1

Для первого юай теста мне кажется этого достаточно.

И давайте наконец напишем код. Нам нужно найти кнопку с текстом GET JOKE и у нас на выбор 2 решения: мы или находим вью с таким текстом или же находим вью с айди (если мы знаем айди) и после нажимаем на нее (предварительно проверив текст на нем).

Т.е. если мы найдем вью по тексту то сможем сразу нажать на нее. А если найдем по айди то еще нам нужно будет проверить текст на нем. Поэтому давайте упростим себе жизнь и сделаем коротко, ясно и понятно

Для начала нужно добавить правило по которому будет работать наш юай тест. Мы запускаем мейн активити и потому пишем паблик переменную через ActivityScenarioRule.

```
class RandomJokesTest {

    @get:Rule
    val activityTestRule = ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun test() {
        onView(withText( text: "GET JOKE")).perform(click())
        onView(withText( text: "mock text 0\nmock punchline 0")).check(matches(isDisplayed()))
    }
```

И посмотрим на сам юай тест – точнее первую часть. Находим вью кнопки и нажимаем. После находим текст шутки и проверяем что он видимый. Если запустить этот тест то он проходит. Но предварительно вам нужно удалить все что было сохранено до. Потому что первый раз у вас пройдет юай тест только если в избранных нет такой же шутки. Т.е. метод проверки что вью с таким текстом видна сработает только если на экране нет такой же вью с таким текстом. Пусть и с другим айди. Ну ладно, давайте допишем юай тест

Нажать на кнопку GET JOKE

отображается шутка с тектом mock text 1 mock punchline 1

```
fun test() {
    onView(withText( text: "GET JOKE")).perform(click())
    onView(withText( text: "mock text 0\nmock punchline 0")).check(matches(isDisplayed()))
    onView(withText( text: "GET JOKE")).perform(click())
    onView(withText( text: "mock text 1\nmock punchline 1")).check(matches(isDisplayed()))
```

Вот и все. Можем запустить юай тест и он пройдет. Т.е. итог работы : зеленый.

Таким вот образом мы проверили что кнопка получения шутки работает. Все легко и просто. Можете написать такой же юай тест и для цитат, предварительно нажав на кнопку цитат на дне экрана.

Но нас интересует еще и вторая фича : не только получить рандомные шутки, но еще и добавить в избранное. И здесь начинается самое интересное.

Напишем тест кейс

Нажать на кнопку “GET JOKE”

отображается шутка с тектом mock text 0 mock punchline 0

Нажать на сердечко

В списке избранных отображается эта шутка

И здесь встает вопрос: а как проверить? Во-первых можно на старте проверить что отображается текст

No favorites! Add one by heart icon

И после добавления оно исчезнет. Но все равно мы не проверили что в списке добавился такой же текствью. Проблема именно в том, что на экране в этот момент будет 2 вью с одинаковым текстом на нем и потому у нас не пройдет юай тест. Значит нужно сделать через айди. Но у нас есть айди внутри FavoriteDataView, а список является ресайклером. Как нам проверить что именно в списке есть элемент с таким текстом, ведь айди другой. Да, мы можем по айди элемента проверить этот тесткейс потому что в списке будет лишь 1 элемент. И в принципе давайте так и попробуем сделать. Создаем новый тестовый класс в том же пакете и будем писать его в отрыве от существующего теста

```
@RunWith(AndroidJUnit4ClassRunner::class)
@LargeTest
class SaveJokeToFavorites {

    @get:Rule
    val activityTestRule = ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun test() {
        onView(withText( text: "No favorites! Add one by heart icon")).check(matches(isDisplayed()))
        onView(withText( text: "GET JOKE")).perform(click())
        onView(withText( text: "mock text 0\nmock punchline 0")).check(matches(isDisplayed()))

        onView(withId(R.id.changeButton)).perform(click())
        onView(withId(R.id.commonDataTextView)).check(matches(withText( text: "mock text 0\nmock punchline 0")))
    }
```

Сначала идет проверка что отображается собщение что нет избранных, после нажимаем на получение шутки и проверяем что она отобразилась (где бы то ни было). После чего нажимаем на сердечко по айди и проверяем что у текствью с айди которое в ресайклере текст соответствует указанному. Если нажать на тест, то он проходит. Все ок. Но что будет если я второй раз нажму на RUN ? Тест не пройдет. Потому что после первого раза в реалм (бд) сохранился моковый ответ и на старте сообщения что нет избранных не будет и тест упадет на первой же проверке. Как быть? Но давайте оставим на секунду эту проблему и посмотрим на другую проблему : у нас после того как мы прогнали юай тест в бд лежат фейк данные. Как нам теперь вернуться к настоящим данным которые мы получали от сервера?

И здесь мы скажем спасибо что писали интерфейсы и модули, в которых выбирали нужные инстансы для клаудДатаСорс. На данном этапе у нас хардкод. Никакого условия нет. И давайте исправим это. Предположим нам в модуль придет булеан флаг : использовать сервер или ненастоящие данные. И исходя из этого будем выбирать клауддатасорс.

```
private fun getCloudDataSource() = if (useMocks)
    MockJokeCloudDataSource()
else
    JokeCloudDataSource(instancesProvider.makeService(BaseJokeService::class.java))
```

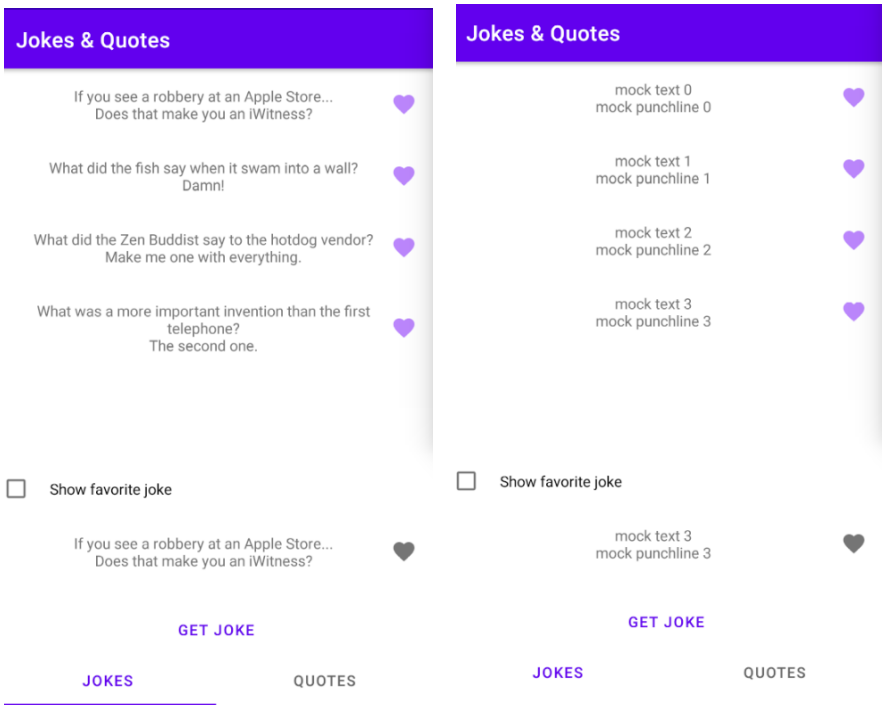
Но это решает лишь проблему источника данных для сервера. Если мы прогнали юай тест, то после него в бд будет лежать фейк. Значит нам нужно написать такой же иф елс и для кешдатасорса? Нет, у нас есть одна точка входа для реалма. Давайте ее сразу и редактировать.

И здесь возникает вопрос: создавать разные инстансы реалма? Или как это сделать? Суть в том, что реалму можно указать какой файл использовать. И здесь мы просто укажем другое имя для моков и для сервера. Посмотрите как это просто сделать

```
class BaseRealmProvider(context: Context, useMocks: Boolean) : RealmProvider {
    init {
        Realm.init(context)
        val fileName = if (useMocks) MOCKS else NAME
        val config = RealmConfiguration.Builder().name(fileName).build()
        Realm.setDefaultConfiguration(config)
    }

    override fun provide(): Realm = Realm.getDefaultInstance()

    companion object {
        const val NAME = "jokesAndQuotesRealm"
        const val MOCKS = "jokesAndQuotesRealmMocks"
```

На старте я инициализирую реалм и создаю конфигурацию. Исходя из пришедшего флага беру или целевое имя файла или мок. Далее ставлю эту конфигурацию по дефолту. Так что после этого данные будут храниться в нужном файле исходя из пришедшего параметра.

Давайте проверим. Для начала запустим проект с флагом useMocks False, сохраним шутки от сервера. После запустим проект с флагом useMocks true и мы должны увидеть пустоту (если до этого удалили приложение). Опять сохраним мок шутки в бд и еще раз запустим с флагом для целевого решения. Все мои шутки от сервера должны быть сохранены.



Как видите каждый тип данных хранится в отдельном файле и не мешает друг другу

С этой задачей разобрались. А как теперь решить задачу, что перед запуском юай теста нужно очистить базу данных? Во-первых для любого юай теста есть аннотации @Before

@After и если пометить методы этими аннотациями, то они будут соответственно запускаться до начала юай теста и после завершения. Проблема в том, что сначала срабатывает onCreate класса Application и лишь потом уже перед тем как начинается юай тест (т.е. когда уже все данные загружены) мы можем что-то сделать. т.е. даже активити создается перед тем как начинается сам юай тест. И это слишком поздно. Значит нам нужно другое решение. Добавим в градл следующую линию.

androidTestImplementation 'androidx.test.ext:junit-ktx:1.1.3'
После чего нам нужно сделать в юай тесте перед прогоном самого теста ленивую инициализацию активити. Т.е. чтобы мы успели сделать что-либо до старта активити. В этом нам поможет активити тест сценарио.
Кастомный класс я запушу на гитхаб, но вы можете его сами найти потратив в гугле пару минут. Теперь мой юай тест выглядит вот так

```
@get:Rule
val activityTestRule = lazyActivityScenarioRule<MainActivity>(launchActivity = false)

@Before
fun before() {
    val realmProvider = BaseRealmProvider(ApplicationProvider.getApplicationContext(), useMocks: true)

    realmProvider.provide().use { it: Realm
        it.executeTransaction { it: Realm
            it.deleteAll()
        }
    }
    activityTestRule.launch(
        Intent(
            ApplicationProvider.getApplicationContext(),
            MainActivity::class.java
        )
    )
}
```

Сначала я беру реалм на моковом файле и очищаю все что там есть. После чего стартую активити как если бы я сам мог нажать на рабочем столе после очищения реалма.

Запустим тест и он проходит! Запускаем повторно и так же все ок.

И давайте напоследок я расскажу о том, как проверять элементы ресайлкера.

Давайте напишем еще один юай тест

1. Получить шутку

2. Сохранить в избранное

3. Повторить пункты 1 и 2

И здесь мы уже не сможем обратиться к текствью по айди который лежит в ресайклере, потому что у обеих тесктвью будет 1 айди и юай тест не пройдет. Значит нам нужен класс, который умеет находить элементы внутри ресайклвью. Для этого напишем кастомный класс матчера ресайкл.

```
class RecyclerViewMatcher(private val recyclerViewId: Int) {
    fun atPosition(position: Int, targetViewId: Int = -1) =
atPositionOnView(position, targetViewId)
    private fun atPositionOnView(position: Int, targetViewId: Int) =
        object : TypeSafeMatcher<View>() {
            var resources: Resources? = null
            var childView: View? = null
            override fun describeTo(description: Description) {
                var idDescription = recyclerViewId.toString()
                if (this.resources != null) {
                    idDescription = try {
                        this.resources!!.getResourceName(recyclerViewId)
                    } catch (e: Resources.NotFoundException) {
                        String.format("%s (resource name not found)", recyclerViewId)
                    }
                }
                description.appendText("RecyclerView with id: $idDescription at
position: $position")
            }
            override fun matchesSafely(view: View): Boolean {
                this.resources = view.resources
                if (childView == null) {
                    val recyclerView = view.rootView.findViewById(recyclerViewId) as
RecyclerView
                    if (recyclerView.id == recyclerViewId) {
                        val viewHolder =
recyclerView.findViewHolderForAdapterPosition(position)
                        if (viewHolder != null) {
                            childView = viewHolder.itemView
                        }
                    } else {
                        return false
                    }
                }
                return if (targetViewId == -1) {
                    view === childView
                } else {
                    val targetView = childView!!.findViewById<View>(targetViewId)
                    view === targetView
                }
            }
        }
}
```

Если немного изучить этот класс, то станет понятен механизм проверки в принципе. Но давайте просто использовать пока что этот класс для нашего юай теста.
И мы сталкиваемся с еще одной проблемой: на экране 2 вью с одинаковыми айди. Иконка сердечка в том месте где сохраняем в избранное и там же где в ресайклере удаляем из избранных имеют одинаковый айди. Просто давайте изменим айди для элемента ресайкла и все будет OK.
Итак, тест для добавления 2 шуток в избранное будет выглядеть вот так

```
@Test
fun test() {
    onView(withText( text: "No favorites! Add one by heart icon")).check(matches(isDisplayed()))
    onView(withText( text: "GET JOKE")).perform(ViewActions.click())
    onView(withText( text: "mock text 0\nmock punchline 0")).check(matches(isDisplayed()))

    onView(withId(R.id.changeButton)).perform(ViewActions.click())
    onView(RecyclerViewMatcher(R.id.recyclerView).atPosition( position: 0, R.id.commonDataTextView))
        .check(matches(withText( text: "mock text 0\nmock punchline 0")))

    onView(withText( text: "GET JOKE")).perform(ViewActions.click())
    onView(withText( text: "mock text 1\nmock punchline 1")).check(matches(isDisplayed()))
    onView(withId(R.id.changeButton)).perform(ViewActions.click())
    onView(RecyclerViewMatcher(R.id.recyclerView).atPosition( position: 1, R.id.commonDataTextView))
        .check(matches(withText( text: "mock text 1\nmock punchline 1")))
}
```

Как видите все довольно просто. Согласен, читать такой код немного сложно, и мы можем упростить его. Каким образом? С помощью котлин экстеншнов и пейдж обьектов.
Что такое котлин экстеншны все знают, а вот пейдж обьекты. Просто напишем класс в который поместим нужные константы. Ведь завтра если мы решим поменять текст отсутствия избранных, то нам фиксить несколько юай тестов. Непорядок. Потому я бы хотел выделить класс и поместить туда константу. Но все это я сделаю вне лекции и вы можете посмотреть на этот код на гитхаб.
А пока давайте напишем еще один юай тест, который добавит 1 избранное и удалит

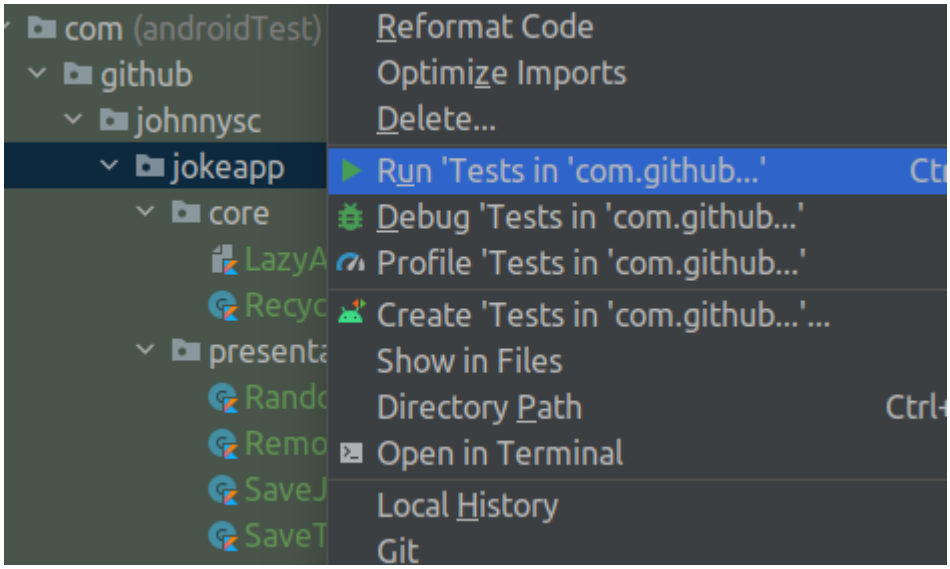
```
@Test
fun test() {
    onView(RecyclerViewMatcher(R.id.recyclerView).atPosition( position: 0, R.id.commonDataTextView))
        .check(matches(withText( text: "No favorites! Add one by heart icon")))

    onView(withText( text: "GET JOKE")).perform(click())
    onView(withText( text: "mock text 0\nmock punchline 0"))
        .check(matches(isDisplayed()))

    onView(withId(R.id.changeButton)).perform(click())
    onView(RecyclerViewMatcher(R.id.recyclerView).atPosition( position: 0, R.id.commonDataTextView))
        .check(matches(withText( text: "mock text 0\nmock punchline 0")))

    onView(RecyclerViewMatcher(R.id.recyclerView).atPosition( position: 0, R.id.removeButton))
        .perform(click())
    onView(withText( text: "yes")).perform(click())
    onView(RecyclerViewMatcher(R.id.recyclerView).atPosition( position: 0, R.id.commonDataTextView))
        .check(matches(withText( text: "No favorites! Add one by heart icon")))

}
```

Если помните то текст о том что нет избранных тоже лежит в ресайклере под тем же айди что и избранный элемент. Потому я могу использовать новый ресайклматчер для этого. И после удаления нужно подтвердить действие в снекбар. Я написал просто так же как и с кнопкой, ведь кнопка в снекбаре тоже вью по сути и у нее есть текст. Главное чтобы не было дублирующихся вью с одним и тем же текстом. Кароче говоря. Вот так выглядит юай тест на удаление из избранных. И на данном этапе можно сказать что весь основной функционал шуток протестирован.
Мы получили шутку, сохранили ее, удалили. То же самое можно написать с цитатами и написать юай тест на сохранение экрана. Кстати точно так же используйте разные имена для шердпрефов и ваш юай тест не будет мешать боевой реализации. Единственное конечно же

неудобно каждый раз менять флаг в апликейшн классе, но вы можете например сделать вот так: если сборка собирается дебажная, то использовать моки, иначе если это релиз, то серверы. Делается это довольно просто

```
class JokesAndQuotesApp : Application() {

    private val viewModelsFactory by lazy {
        ViewModelsFactory(
            MainModule(coreModule),
            JokesModule(coreModule, useMocks),
            QuotesModule(coreModule)
        )
    }

    private lateinit var coreModule: CoreModule
    private val useMocks = BuildConfig.DEBUG

    override fun onCreate() {
        super.onCreate()
        coreModule = CoreModule( context: this, useMocks)
    }

    fun <T : ViewModel> get(modelClass: Class<T>, owner: ViewModelStoreOwner): T =
        ViewModelProvider(owner, viewModelsFactory).get(modelClass)
}
```

Это конечно же не самое лучшее решение. Есть и более подходящие, такие как создание своего флейвора или же константы в градле. Но темой текущей лекции является именно юай тестирование и подмена реализаций.
Как видите не нарушая принципы SOLID и ООП можно добиться того, что весь ваш функционал можно проверить нажав 1 кнопку. Какую?



Run Tests in package и все ваши юай тесты за условные 10 секунд проверят весь функционал. Красота!
Юай тесты проверяют функционал приложения и делают это намого быстрей и лучше чем человек, когда берет в руки девайс, очищает данные и так далее. Пишите юай тесты.