

Автотесты

Прощай рутинная работа

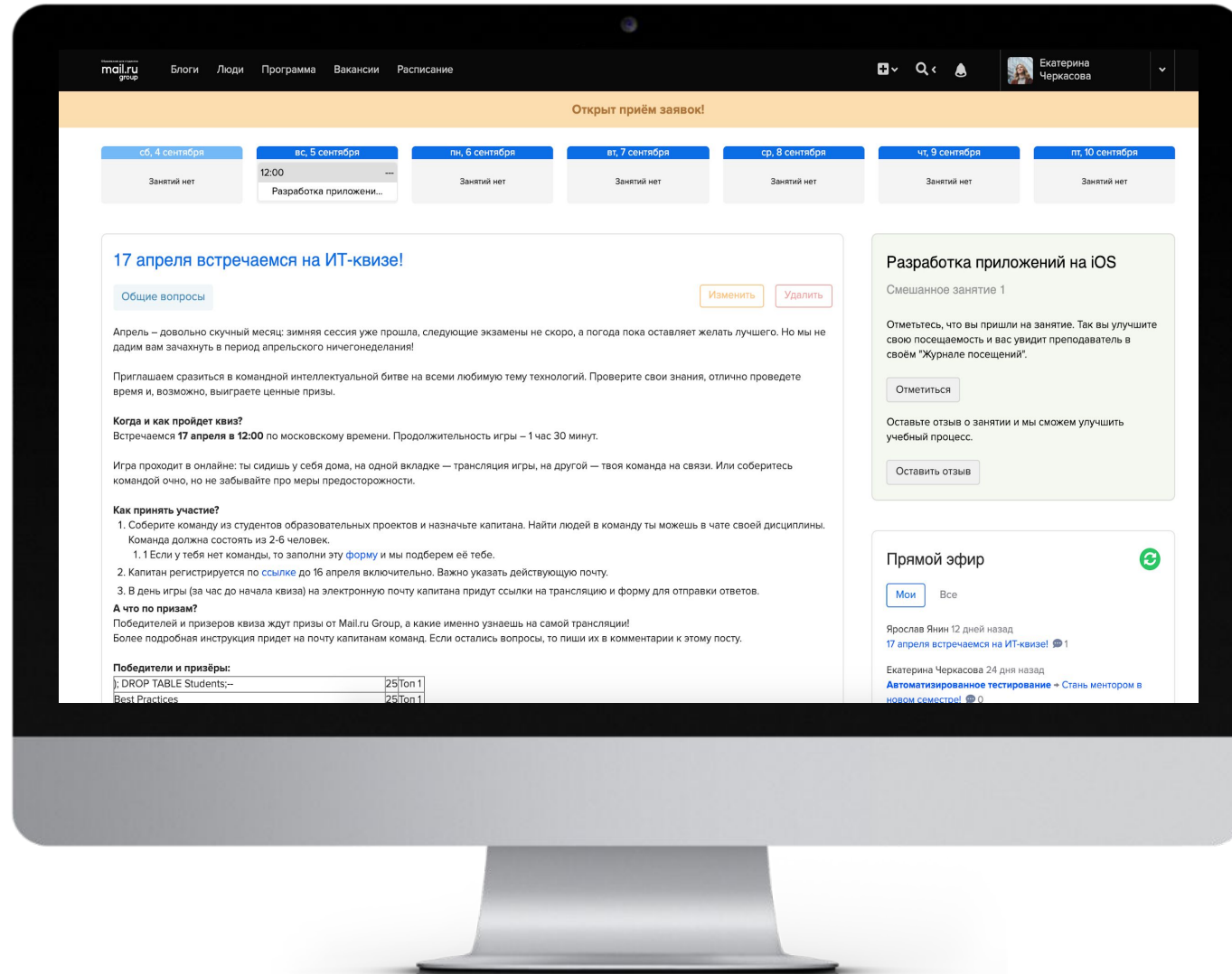




Содержание занятия

1. Тестирование
2. Автотестирование
3. Инструменты

Не забудьте отметиться!



Тестирование



Тестирование - это
сложно или нет?

Нужны тест кейсы

Это проверка работоспособности функциональности.

Необходимы, чтобы можно было проверить приложение. Познакомится с ней не зная код. Рассматривая приложение как “черный ящик”.

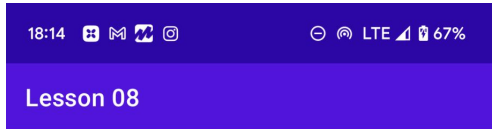
Написать тест кейс – значит создать текстовое описание процесса тестирования функциональности.

Примерная структура:

1. Номер
2. Название
3. Предусловие
4. Порядок действий
5. Ожидаемый результат

Давайте попробуем:)

Давайте создадим пример тест кейса



1. 0001
2. Первый запуск
3. Установить приложение на устройство
4. Запустить приложение
5. Откроется экран, на нем должны быть кнопки FIELDS и LIST



1. 0002
2. Нажатие на FIELDS
3. Приложение открыто на первом экране
4. Нажать на кнопку FIELDS
5. Открытие экрана с полем ввода и кнопкой ACTION



Таблица принятия решений

Инструмент, который помогает составлять возможные тест кейсы. Также – помогает не упустить тест кейсы.

“Один столбец – один тест кейс”.

После составления таблицы - ее можно будет попробовать оптимизировать. А потом записать тест кейсы.

Для удобства, таблицу иногда инвертируют.

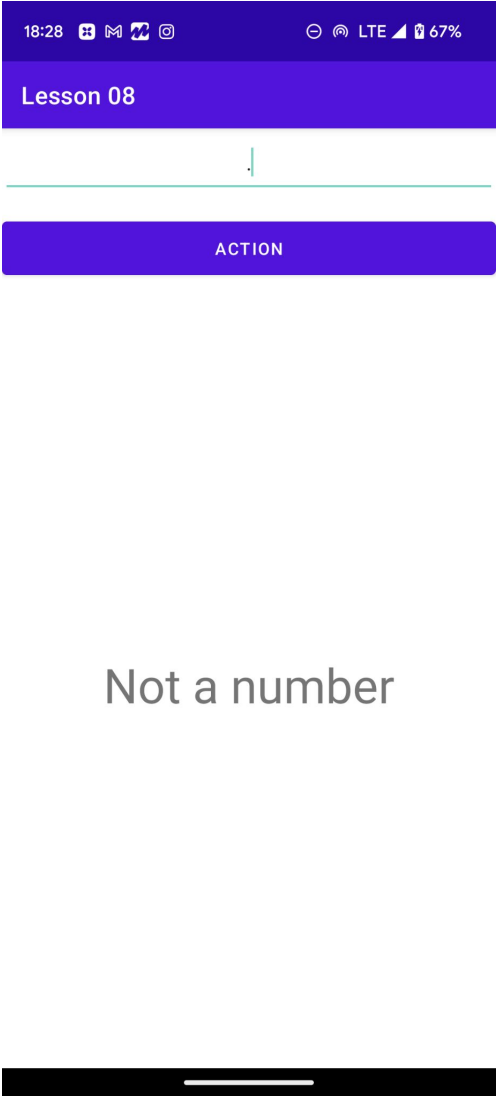
	Правило 1	Правило 2	...	Правило N
Условия				
Условие 1				
Условие 1				
...				
Условие N				
Действия				
Действие 1				
Действие 2				
...				
Действие N				

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Условия																
Потратил	100	100	100	100	500	500	500	500	1000	1000	1000	1000	1000	1000	1000	1000
Выкуп	5%	30%	50%	80%	5%	30%	50%	80%	5%	30%	50%	80%	5%	30%	50%	80%
Плюшка																
Скидка	0%	5%	5%	7%	5%	6%	7%	8%	8%	8%	10%	15%	11%	13%	20%	20%
Кол-во вещей	2	4	4	6	8	8	9	9	6	10	15	15	15	15	15	20

Давайте теперь накидаем
небольшую таблицу

Пример таблицы

	Правило 1	Правило 2	Правило 3	Правило 4
Предусловие	0	-5	aa	
Действие	ACTION	ACTION	ACTION	ACTION
Результат	1	Negative value	Not a number	BLANK



Нужна усидчивость

Какое количество может быть тест кейсов?

ОЧЕНЬ БОЛЬШОЕ даже на небольшой блок экрана.

И с развитием приложения, количество тест кейсов будет только расти.

Человек не может бесконечно оптимизировать свою подготовку, для уменьшения время прохождения одного тест кейса.

И часто придется гонять одни и те же тест кейсы.

Виды тестирования (далеко не полный список)

После составления всех тест кейсов. Может возникнуть вопрос - а их надо постоянно проходить? Ведь это очень много временных ресурсов.

Тест кейсы можно выделять в определенные группы, и использовать их в зависимости от ситуации.

Я сталкивался с ситуацией, что в разных компаниях одинаковые виды тестов называли по разному. И только одна группа тестов именовалось одинаково везде, и это был самый страшный вид тестирования - **Regression Test**:)

Обычно это означает полную перепроверку всех тест кейсов, без учета изменений со стороны разработки. Такой вид тестирования заказывается после внесения опасных и обширных изменений в приложение.

Для экономии времени можно попробовать уменьшить количество тест кейсов:

- **Smoke Test** - направлено на проверку функциональности “вширь”. Обычно - это поверхностная проверка максимального количества фич.
- **Sanity Test** - направлено на проверку функциональности “вглубь”. Выделяется важная (основная) функциональность для проверки по максимум.
- **Re-Test** - проверка исправления дефекта. Для перепроверки используется тест кейс, который ранее воспроизводил дефект.

Автотестирование

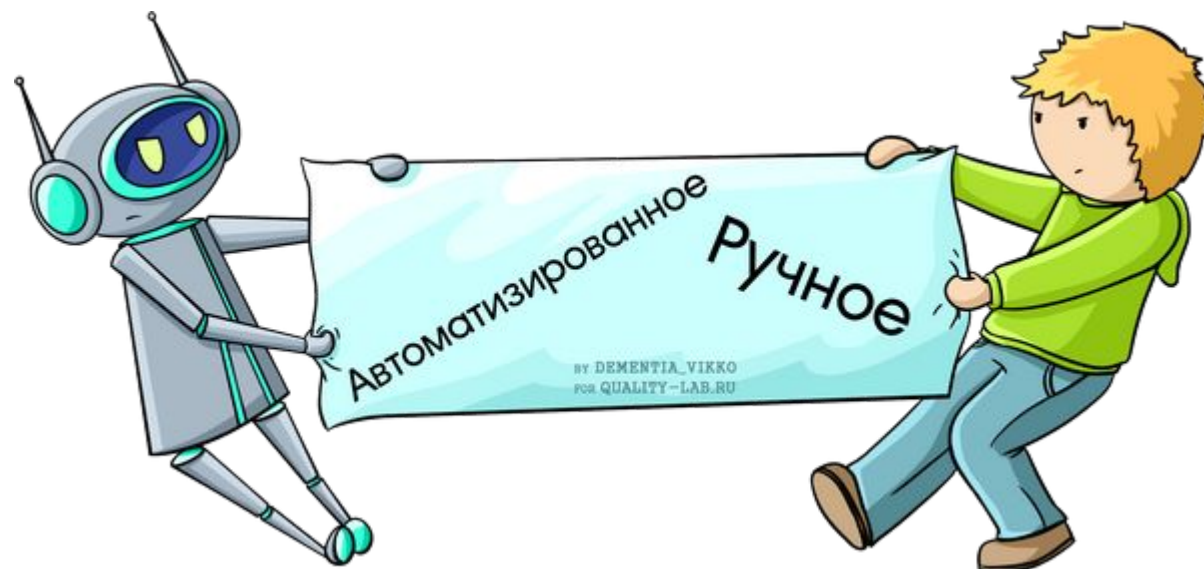


Автотестирование – это

... скрипт, имитирующий взаимодействие пользователя с приложением.

Можно выделить два подхода:

- Модульное тестирование – тестирование на уровне кода
- Тестирование пользовательских интерфейсов – имитация пользовательского взаимодействия



Модульное тестирование

Выделяем модуль, класс, для тестирования логики.

Подаем в модуль данные, смотрим на то, как модуль изменяет объекты и проверяем что объекты изменились согласно ожиданиям.

```
@Before
fun setUp() {
    calculator = FactorialCalculator()
}

@Test
fun check_not_digit_1() {
    calculator
        .calculate("--1")
        .check("Not a number")
}
```


Автотестирование UI

Скрипт взаимодействует с интерфейсом, выполняет пользовательские действия.

Потом скрипт проверяет состояние интерфейса или состояние данных.

Предварительно, приложение может заполняться тестовыми данными и “мокнутыми” объектами. В результате мы можем потом состояние сравнить с ожидаемым.

```
@Test
fun check_navigation_to_fields() {
    onView(withText("Fields")).perform(click())
    onView(withText("Action")).check(matches(isDisplayed()))

    onView(withHint("Field")).perform(typeText("123"))
    onView(withText("Action")).perform(click())
    onView(withId(R.id.text)).check(matches(withText("123")))
}
```

Типы тестирования в Android

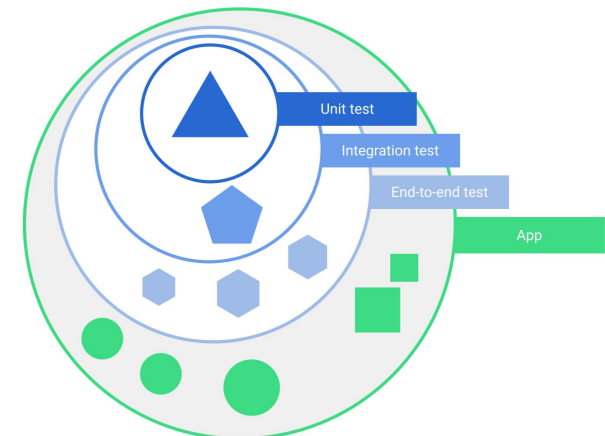
По предмету тестирования


- **Functional** - работает это так как должно?
- **Performance** - работает это быстро и эффективно?
- **Accessibility** - могут ли использовать люди с ограниченными возможностями?
- **Compatibility** - работа вне зависимости от устройства и версии системы
- ... можно подобрать и другие

По объему(скоупу) тестирования

Разделяются на основе степени изоляции или размера.

- **Unit/Small tests** - проверка небольшой части приложения (класс или метод)
- **End-to-End/Big tests** - проверка большей части приложения. Например, пользовательский флоу или экран.
- **Middle tests** - проверка работы и взаимодействия двух или более модулей.





И еще одно важное
разделение:)

По среде запуска

Local test (Локальные, на JVM)

Запускают локально, просто на JVM. Такого рода тесты гоняются быстрее, но не имеют доступ к Android без дополнительной подготовки.

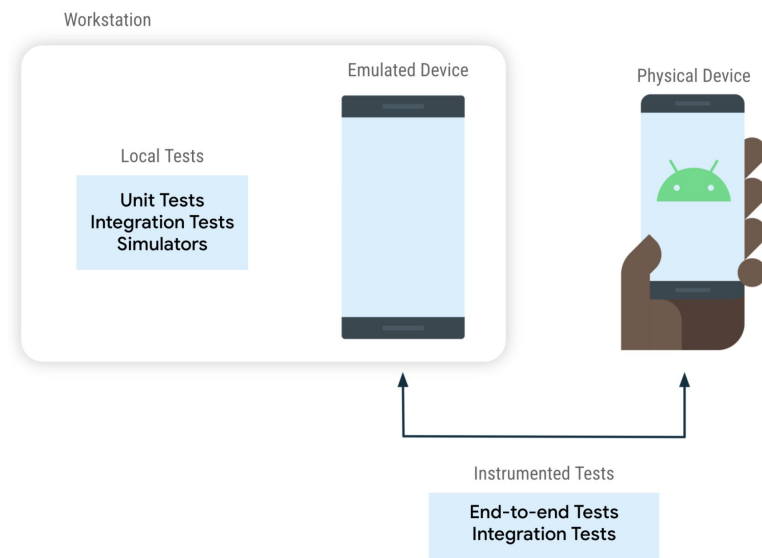
Например, если ваш модуль касается Android зависимостей, то такой тест будет всегда падать. Но можно будет добавить имитацию Android (обычно при помощи дефолтов или Robolectric).

Обычно такие тесты лучше делать для модулей, которые не зависят от Android.

Instrumented tests (В настоящей среде)

Запускаются на Android (эмуляторе или устройстве). Исполняются дольше, зато исполняются в настоящей среде и можно учитывать поведение самой системы.

Если проверяете что именно Android специфичное - работу базовых компонентов или отрисовку. То потребуется использовать эту среду для запуска.



Уход от тестирования “черного ящика”

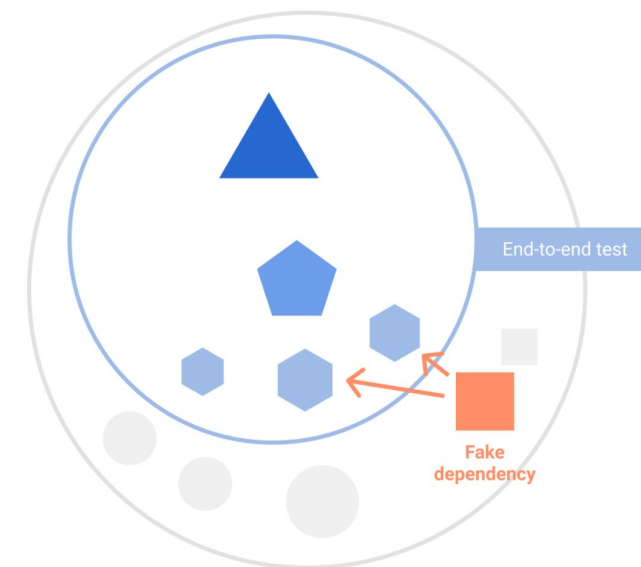
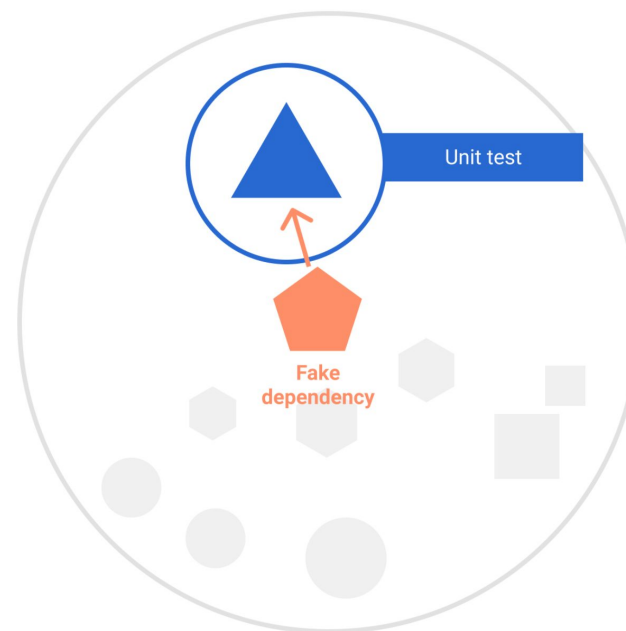
Можно встретить в литературе такое упоминание как “тестовые двойники”.

Для тестирования отдельных модулей или группы модулей позволяет ускорить проверки тест кейсов.

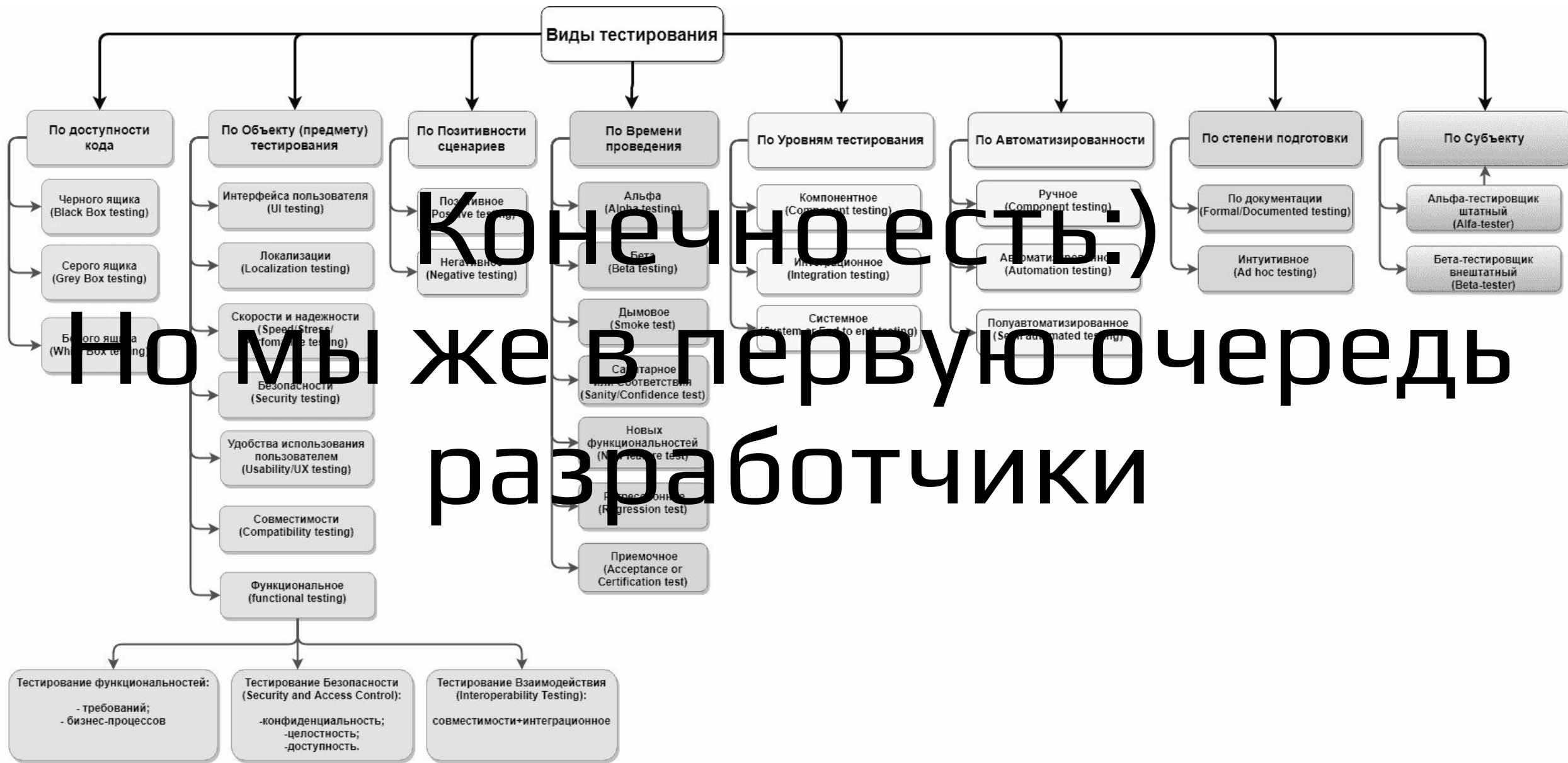
Один из их представителей разбирали в прошлом семестре - это фейковые и заглушечные зависимости или данные.

Еще часто используемый двойник - это мок. Это объекты, которые помогают имитировать поведение и изучать поведение. Есть его рукописный аналог - шпион (spy).

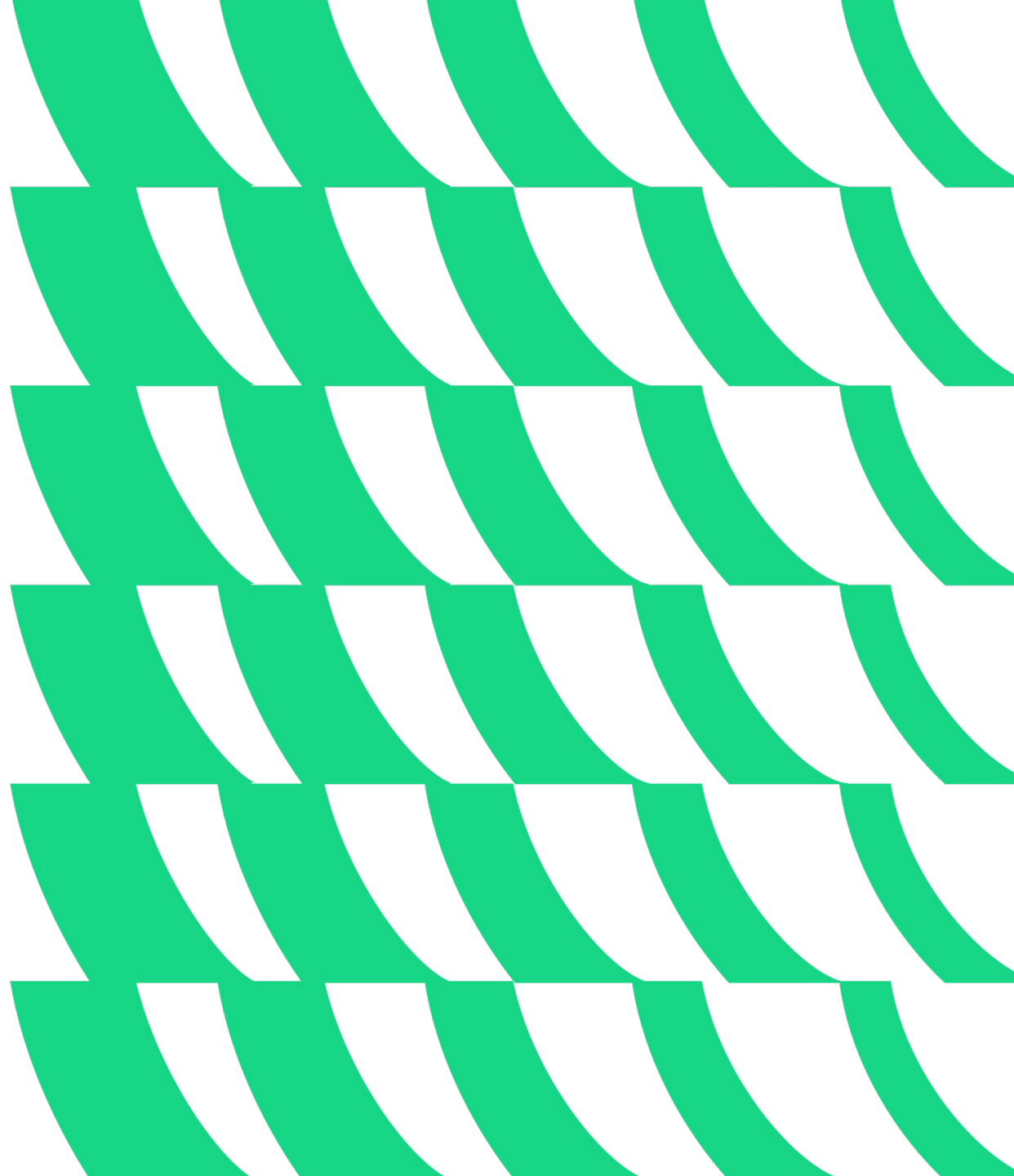
Чтобы использовать двойников требуется следовать инверсии зависимостей и использовать фреймворки для вставки зависимостей.



Есть ли что-то еще?
Ваши версии.



Инструменты



Какой инструмент
пригодится любому
тестировщику?

Снифферы

Нюхач:)

Это инструменты для анализа трафика. Особо актуальный в клиент-серверных приложениях

Популярными представителями можно назвать - Charles или Proxyman (он подходит для Windows и Mac) и Fiddler Classic (подходит в основном для Windows).

Есть более продвинутые инструменты, как Wireshark, например. Но их сложнее использовать для целей тестирования.

Главное чем должен обладать инструмент помимо sniffing - это помогать делать фейковые ответы, или менять ответ в процессе работы.



Fiddler Classic



Фреймворк для автоматизации тестирования

Самый популярный фреймворк в Android автотестах – это **Espresso**.

Его главные плюсы – тест пишется на “нативном” для разработчика языке и может видеть ресурсы приложения.

Его ближайший аналог - **UI Automator**. Если в вашем приложении добавляется множество сторонних библиотек, которые под капотом плодят AsyncTask, то в этом случае можно будет глянуть на него.

Еще есть популярный, кроссплатформенный фреймворк – это **Appium**. На нем тесты пишутся на пайтон, и он имеет слабое представление о ресурсах приложения.

Из своей практики могу сказать что сначала все покупаются на Appium, но потом все равно переходят на нативные фреймворки.



Android
uiautomator

Фасадный фреймворк

Kaspresso

Он устроен так, что вы можете использовать у себя под капотом Espresso или UI Automator.

Также, команда позаботилась над тем, чтобы включить в него все лучшие практики тестирования (чтобы не писать их самостоятельно). Так же есть механика для поддержания работы с flaky тестами (это тесты, у которых нет 100% кейса выполнения, в основном из-за поведения системы).



Test Runner

Простыми словами — тот кто гоняет ваши тесты.

Он зачастую зависит от выбранного фреймворка, указывается в gradle файле.

Для обычных Espresso тестов достаточен AndroidJUnitRunner.

Другие фреймворки, чтобы делать подкапотную работу, потребуют указать свой раннер.

```
defaultConfig {  
    ...  
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
}
```

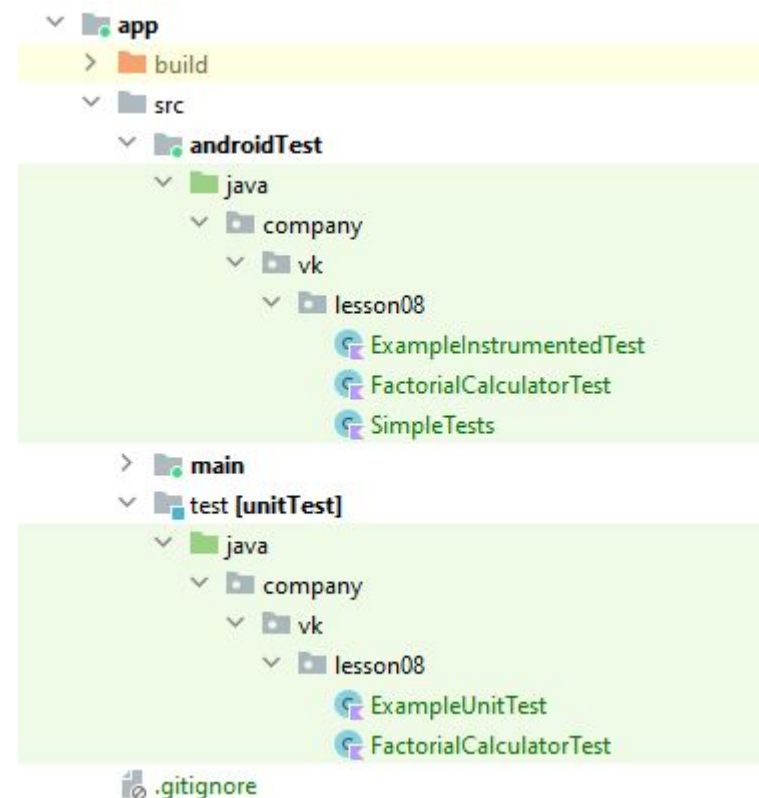
```
defaultConfig {  
    ...  
  
    testInstrumentationRunner  
    "com.kaspersky.kaspresso.runner.KaspressoRunner"  
}
```

Расположение тестов

Есть отдельные директории для тестового кода

- **test** – это локальные тесты
- **androidTest** – instrumented тесты


В этих директориях также можно создавать фейковые объекты и ресурсы для работы.



Зависимости для тестирования

Поскольку эти зависимости не нужны в конечном приложении, поэтому за счет механики работы gradle, можно указать правило подтягивания этих зависимостей только для тестового кода.

```
dependencies {  
    // Core library  
    androidTestImplementation 'androidx.test:core:1.5.0'  
  
    // AndroidJUnitRunner and JUnit Rules  
    androidTestImplementation 'androidx.test:runner:1.5.2'  
    androidTestImplementation 'androidx.test:rules:1.5.0'  
  
    // Assertions  
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'  
    androidTestImplementation 'androidx.test.ext:truth:1.5.0'  
    androidTestImplementation 'com.google.truth:truth:1.1.3'  
  
    // Espresso dependencies  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'  
    androidTestImplementation 'androidx.test.ext:junit-ktx:1.1.5'  
  
    // for jvm tests  
    testImplementation 'junit:junit:4.13.2'  
}
```



**Настало время
автоматизировать
тест кейс**

А если есть асинхронные операции?

Самый простой хак – это опираться на View, который говорит о том что данные грузятся. Когда он переходит в нужное состояние, то можно посмотреть на данные компонента.

Второй способ, который может потребоваться, – это использовать Idling Resource механику. К этому объекту будет обращаться раннер, для того, чтобы понять - надо ли исполнять следующий шаг теста

Для этого нам потребуется взять имеющуюся имплементацию(которых немного) или написать свою. Потом зарегистрировать ее.

Как пример - когда выполняется асинхронный запрос, объект будет стопить выполнение теста при помощи метода `isIdleNow()`.

```
public void waitForViewToDisappear(int viewId, long maxWaitingTimeMs) {  
    long endTime = System.currentTimeMillis() + maxWaitingTimeMs;  
    while (System.currentTimeMillis() <= endTime) {  
        try {  
            onView(allOf(withId(viewId), isDisplayed()))  
                .matches(not(doesNotExist()));  
        } catch (NoMatchingViewException ex) {  
            return; // view has disappeared  
        }  
    }  
    throw new RuntimeException("timeout exceeded");  
}
```

VS

```
IdlingRegistry.getInstance().register(mIdlingResource)
```

RecyclerView тоже вносит коррективы

RecyclerView располагает своих детей с одинаковыми id – это может усложнить выбор конкретного ребенка.

RecyclerView рисует не всех детей.

Для работы с RecyclerView есть отдельный класс - **RecyclerViewActions** с методами

- scrollTo() - промотать до нужной View
- scrollToHolder() - промотать до нужного ViewHolder
- scrollToPosition() - промотать до позиции
- actionOnHolderItem() - выполнить действие на ViewHolder
- actionOnItem() - выполнить действие на View
- actionOnItemAtPosition() - выполнить действие на позиции

```
@Test
fun check_click_at_position() {
    onView(withId(R.id.recycler))
        .perform(RecyclerViewActions.scrollToPosition<ViewHolder>(7))
    onView(withId(R.id.recycler))
        .perform(RecyclerViewActions.actionOnItemAtPosition<ViewHolder>(7, click()))
}
```

Тесты с моками

Есть два популярных фреймворка для моков, примерно с одинаковым api работы:

- mockk
- mockito

Из-за особенностей kotlin, для Android API ниже 33 может потребоваться использовать еще плагин чтобы сделать данную механику рабочей:

- DexOpener

Эти фреймворки позволяют как давать кастомные ответы на вызов методов, так и потом смотреть количество вызовов функций у мокнутых объектов.

```
// TestRunner с использованием DexOpener
class OpenedDexJUnitRunner: AndroidJUnitRunner() {
    override fun newApplication(cl: ClassLoader, className: String, context: Context)
        : Application {
        // MockK supports for mocking final classes on Android 9+.
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.P) {
            DexOpener.install(this)
        }

        return super.newApplication(cl, className, context)
    }
}

// Подключение в gradle
android {
    defaultConfig {
        ...
        testInstrumentationRunner "company.vk.lesson08.OpenedDexJUnitRunner"
    }
    ...
}

@Test
fun check_data_at_position() {
    ...
    val mockedCharacterAccessor = ServiceLocator.inject(ICharacterAccessor::class.java)
    coVerify(exactly = 1) { mockedCharacterAccessor.list(0, 20) }
}

protected fun createFakeFactory(): ServiceLocator.IFactory {
    ...
    val spiedCharacterAccessor = spyk(tmp)
    val fakeFactory = FakeFactory().apply {
        add(ICharacterAccessor::class.java, spiedCharacterAccessor)
    }
    ...
}
```

Тесты с моками запросов

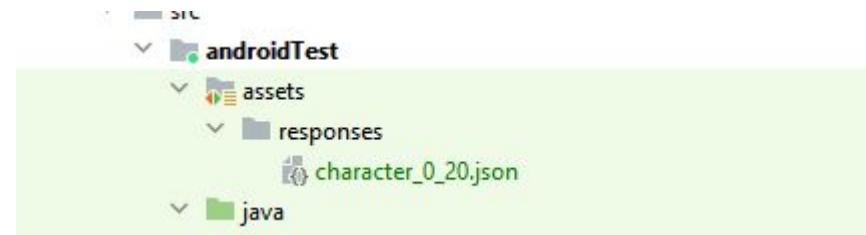
В ситуации, когда хочется максимально сохранить целостность приложения и провести разные UI тесты, то можно мокать ответы на запросы.

Для этого помогает фреймворк:

- MockWebServer

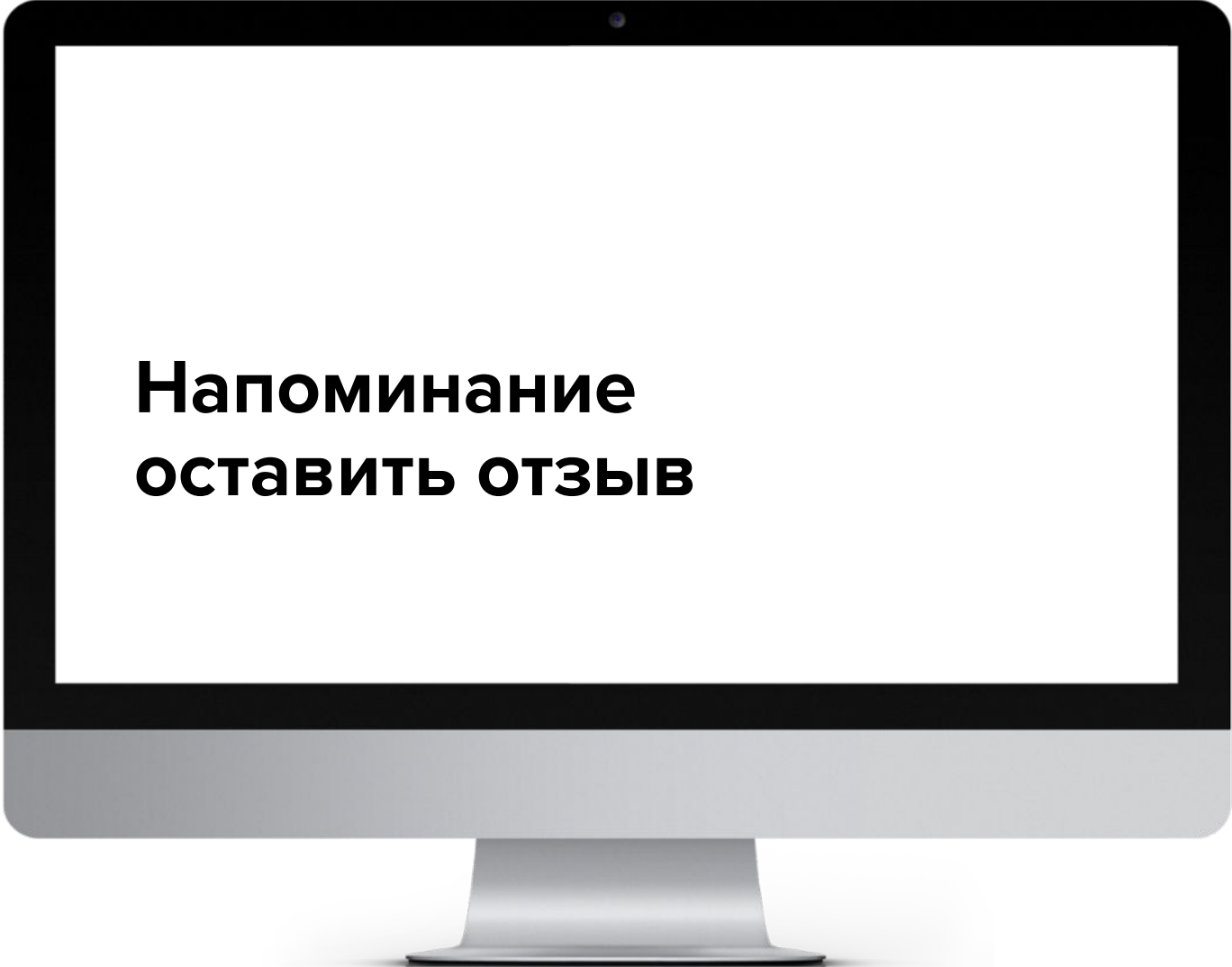
Во время использования этого фреймворка можно максимально кастомизировать ответы, тем самым проверить все состояния (успешные и ошибки).

Если идти этим путем, то придется заранее, при помощи сниффера, сделать большое количество моков ответов. Обычно, эти моки хранят в директории **assets**.



```
@AfterClass
@JvmStatic
fun globalShutdown() {
    mockedWebServer.shutdown()
}
```

```
@Test
fun check_data_at_position() {
    val response = // ответ от API, которые требуется подставить
    mockedWebServer.enqueue(MockResponse().setBody(response))
    ...
}
```



**Напоминание
оставить отзыв**

Спасибо за
внимание!

