

Инструменты для автотестов Android

Дмитрий Мовчан

Дмитрий Мовчан

- Android разработчик в Revolut
- Android Academy MSK организатор
- Спикер Mobius, Appsfest и другие...

Revolut



Android Academy MSK



<https://t.me/AndroidAcademyMsk>
<https://t.me/AndroidAcademyMskNews>

Twitter Мобильный разработчик

<https://twitter.com/mobileunderhood>



Так, опять автотесты???

Все хотят выпускать фичи как можно быстрее

Что для этого нужно?

Один из пунктов это как можно быстрее убедиться в том, что продукт требуемого качества

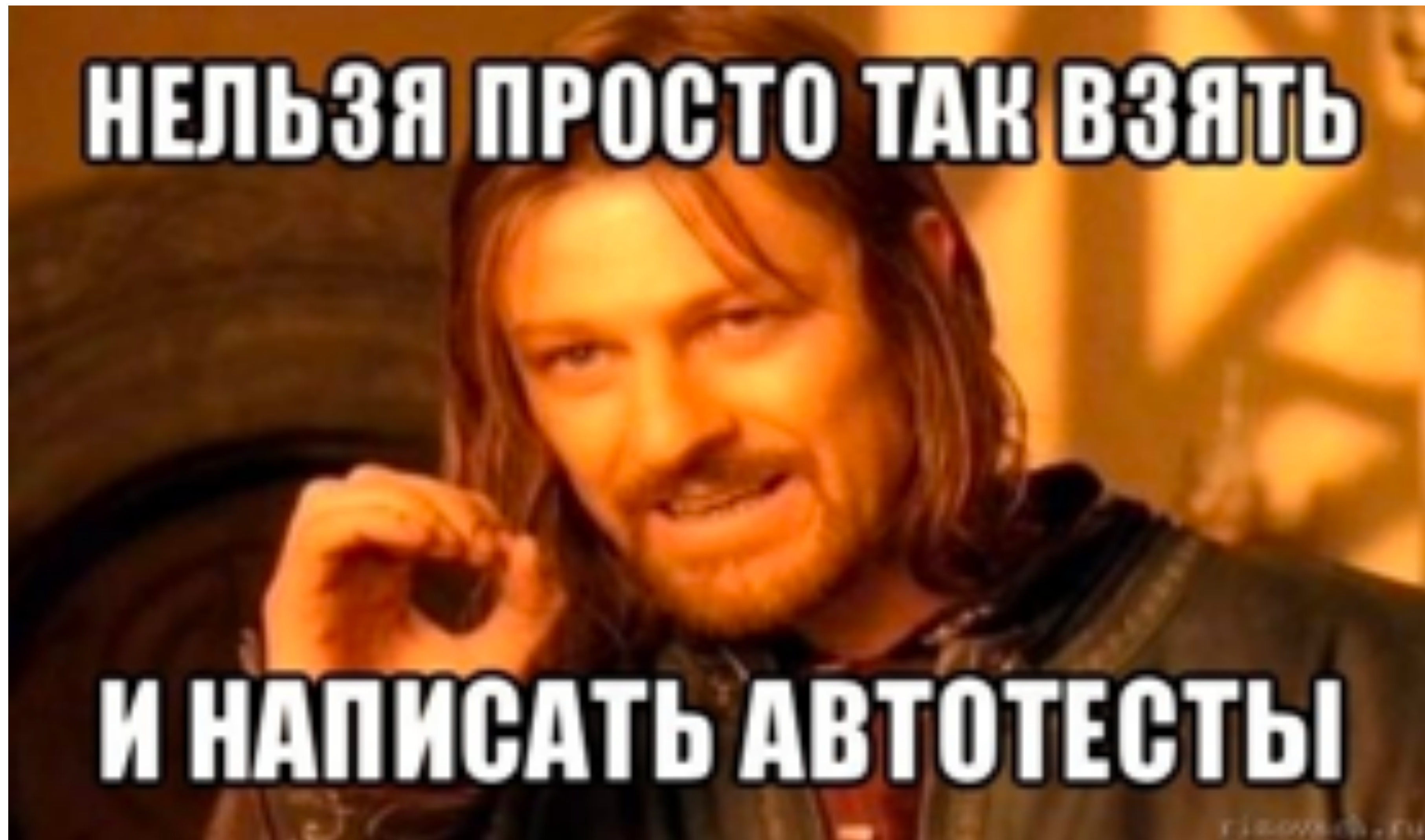
Как это сделать?

Как это сделать?

Проводить автоматическое тестирование, которое максимально приближено к поведению пользователя => **UI тесты**



Начать писать автотесты



Так в чем же проблема?

Проблемы:

- в индустрии нет четкого флоу, как писать автотесты
- есть много открытых и неясных вопросов
- каждый сам за себя



Вот зачем этот доклад

- сэкономит вам много времени
- сэкономит вам кучу нервов
- даст вам базу, с которой вы сможете просто начать писать автотесты

**В какой момент стоит
задуматься об автотестах**



In 05.2018

Релизы были крайне редко

Регресс длился 21 ч/д

Комьюнити начинало говорить о пользе автотестов

- Подкасты
- Конференции
- Статьи
- Инструменты

Что переосмыслить?

- Тесты должны быть не только black box
- Тесты должны быть вместе с проектом
- Разработчики должны драйвить написание автотестов

**Что нам потребуется в
первую очередь?**

Инструменты

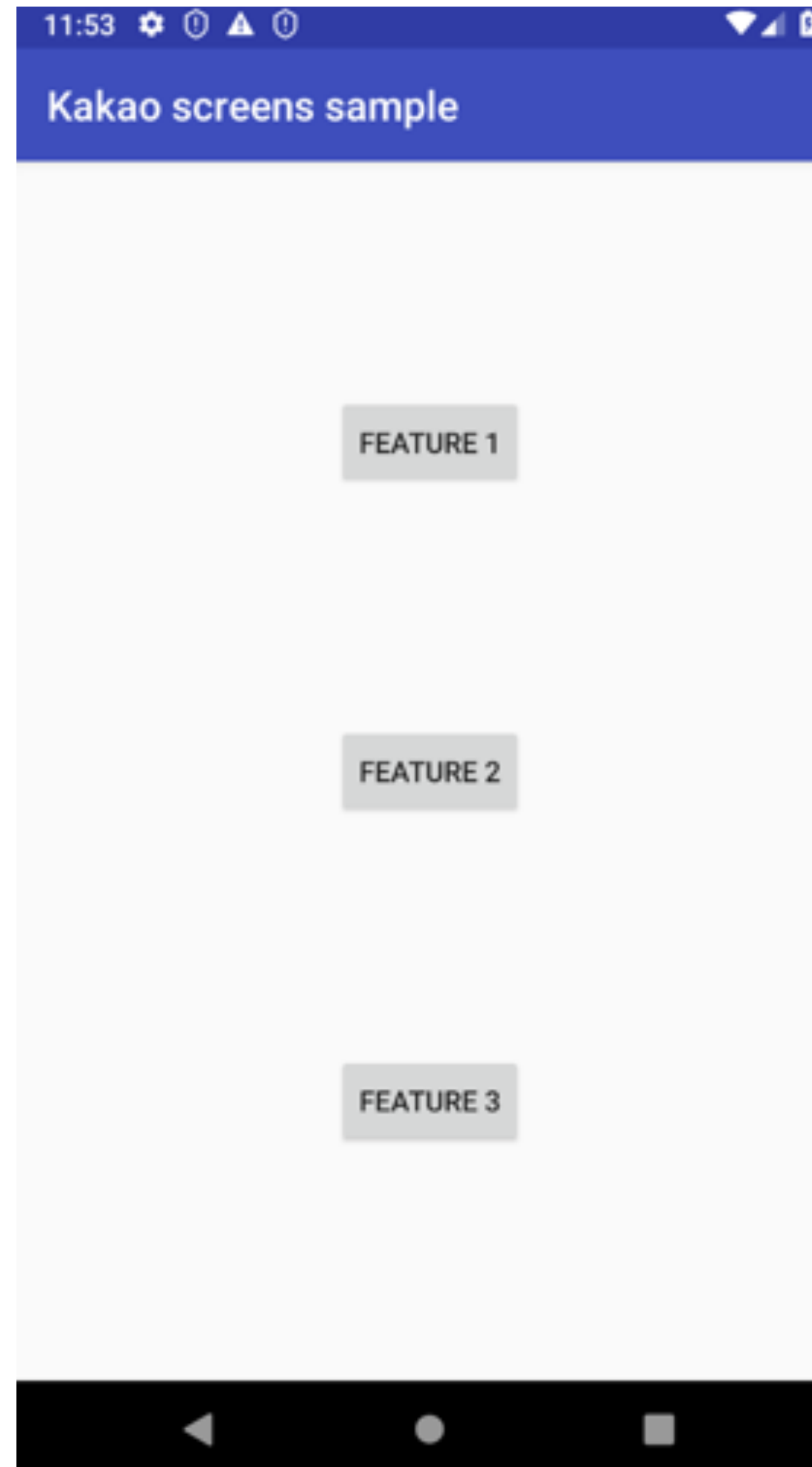
Выбор инструментов

Что может быть проще? Ответ же очевиден - Espresso.

Espresso

```
@Test
fun testFirstFeature() {
    onView(withId(R.id.toFirstFeature))
        .check(ViewAssertions.matches(
            ViewMatchers.withEffectiveVisibility(
                ViewMatchers.Visibility.VISIBLE)))
    onView(withId(R.id.toFirstFeature)).perform(click())
}
```


Kakao



Page Object

```
class MainActivityScreen: Screen<MainActivityScreen>() {  
    val toFirstFeatureButton = KButton { withId(R.id.toFirstFeature)}  
    val toSecondFeatureButton = KButton { withId(R.id.toSecondFeature)}  
    val toThirdFeatureButton = KButton { withId(R.id.toThirdFeature)}  
}
```



```
private val mainScreen = MainActivityScreen()
```

```
@Test
```

```
fun testFirstFeature() {
```

```
    mainScreen {
```

```
        toFirstFeatureButton {
```

```
            isVisible()
```

```
            click()
```

```
        }
```

```
    }
```

```
}
```

Kakao

```
private val mainScreen = MainActivityScreen()
```

```
@Test
fun testFirstFeature() {
    mainScreen {
        toFirstFeatureButton {
            isVisible()
            click()
        }
    }
}
```

Espresso

```
@Test
fun testFirstFeature() {
    onView(withId(R.id.toFirstFeature))
        .check(ViewAssertions.matches(
            ViewMatchers.withEffectiveVisibility(
                ViewMatchers.Visibility.VISIBLE)))
    onView(withId(R.id.toFirstFeature)).perform(click())
}
```

Barista

Расширяет функционал espresso, но не решает проблемы громоздкого кода

```
// Clear all app's SharedPreferences
@Rule public ClearPreferencesRule clearPreferencesRule = new ClearPreferencesRule();

// Delete all tables from all the app's SQLite Databases
@Rule public ClearDatabaseRule clearDatabaseRule = new ClearDatabaseRule();

// Delete all files in getFilesDir() and getCacheDir()
@Rule public ClearFilesRule clearFilesRule = new ClearFilesRule();

// Use @AllowFlaky to let flaky tests pass if they pass any time.
@Test
@AllowFlaky(attempts = 5)
public void some_flaky_test() throws Exception {
    // ...
}
```

- **Scrolls on all views:** Barista scrolls on all scrollable views, including `NestedScrollView`. Espresso only handles `ScrollView` and `HorizontalScrollView`

Все? Конечно нет.

Все автотесты нужно запускать и тут не обошлось без проблем...

Запуск автотестов

AndroidJUnitRunner

- Общий раннер, встроенный в систему Android
- Замена предыдущему раннеру InstrumentationTestRunner (deprecated)
- Запускается путем adb команд, например:

Running all tests: adb shell am instrument -w com.android.foo/
android.support.test.runner.AndroidJUnitRunner

Running all tests in a class: adb shell am instrument -w -e class com.android.foo.FooTest
com.android.foo/android.support.test.runner.AndroidJUnitRunner

Running a single test: adb shell am instrument -w -e class com.android.foo.FooTest#testFoo
com.android.foo/android.support.test.runner.AndroidJUnitRunner

Running all tests in multiple classes: adb shell am instrument -w -e class
com.android.foo.FooTest,com.android.foo.TooTest com.android.foo/
android.support.test.runner.AndroidJUnitRunner

<https://developer.android.com/reference/android/support/test/runner/AndroidJUnitRunner>
<https://developer.android.com/reference/android/test/InstrumentationTestRunner.html>

Почему использовать чистый раннер сложно

- Полное отсутствие конфигурации (за исключением только того, что можно указать какие конкретно тесты должны быть запущены)
- Запускает только 1 инстанс приложения на все тесты
При падении хотя бы одного теста – будет краш всех следующих тестов
- Сложная работа с распараллеливанием тестов на несколько устройств

Вывод:

- Нужно искать какую-то обертку над этим раннером, в которой все эти проблемы решены

Какие есть решения

В порядке технической сложности и реализованных фичей:

- Orchestrator
- connectedAndroidTest – нативная задача gradle
- Spoon/composer
- Marathon

Orchestrator

По сути не является отдельным решением, помогает только «расширить» функционал стандартного раннера, путем создания отдельных инстансов на каждый тест + возможность подчищать состояние перед каждым тестом.

+:

- Создание инстансов на каждый тест (если 1 тест упадет, тестирование не остановится)
- Возможность полностью очищать состояние приложения перед каждым тестом (pm clear)

-:

- Требуется установка дополнительных тестовых сервисов на каждое устройство
- Нет отчета

	AndroidJUnitRunner	Orchestrator
Каждый тест - отдельный инстанс	-	+
Возможность очистки состояния	-	+
Требует установленных Google play services	-	+
Распараллеливание	+ -	+ -

connectedAndroidTest

`./gradlew connectedAndroidTest`

Поддерживает orchestrator с помощью дополнительных параметров

Что делает:

- Собирает проект и тестовую apk
- Запускает тесты
- Генерирует отчет

Что плохо:

- Практически никакой настройки нет (по сути можно только передавать параметры)
- Нельзя отделить сборку проекта от запуска тестов -> следовательно тесты будут гнаться на том же компьютере, который собирает проект. Возможности запустить отдельно тесты без сборки проекта – нельзя.

connectedAndroidTest – как выглядит отчет

Package net.rafaeltoledo.coverage

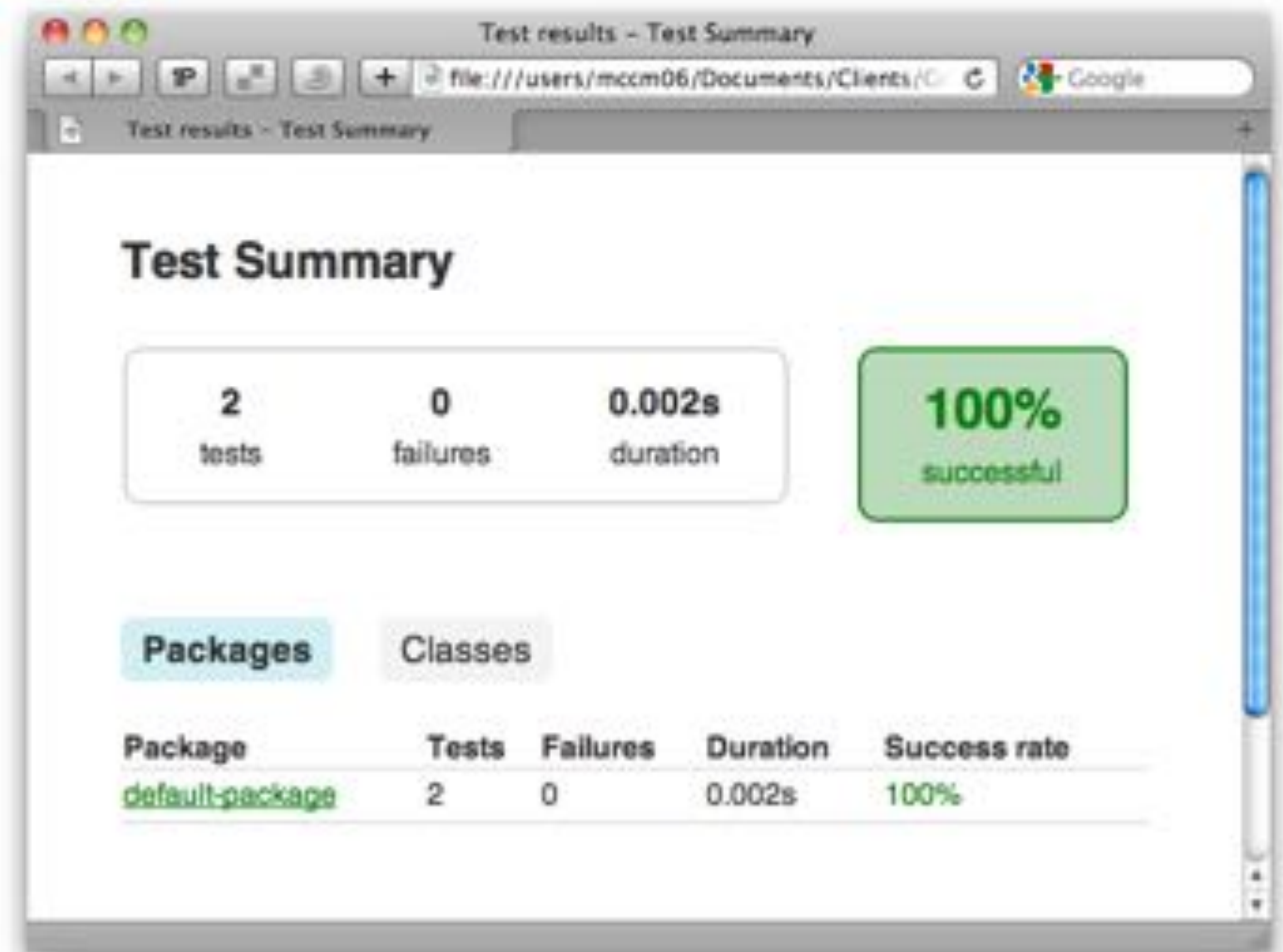
all > net.rafaeltoledo.coverage

1 tests 0 failures 0.554s duration

100%
successful

Classes

Class	Tests	Failures	Duration	Success rate
MainActivityTest	1	0	0.554s	100%



	AndroidJUnitRunner	Orchestrator	ConnectedAndroidTest
Каждый тест - отдельный инстанс	-	+	/
Возможность очистки состояния	-	+	/
Требует установленных Google play services	-	+	/
Распараллеливание	+ -	+ -	/
Отчет	-	-	+

Spoon/composer

Spoon - полноценное решение для запуска тестов. В силу обстоятельств Spoon был заброшен, в то время как создатели Composer сделали свое решение практически используя те же наработки что и создатели Spoon.

+:

- Поддержка распараллеливания тестов.
- Для запуска тестов нужно указать путь к собранному арк и арк с тестами.
- Поддержка Orchestrator

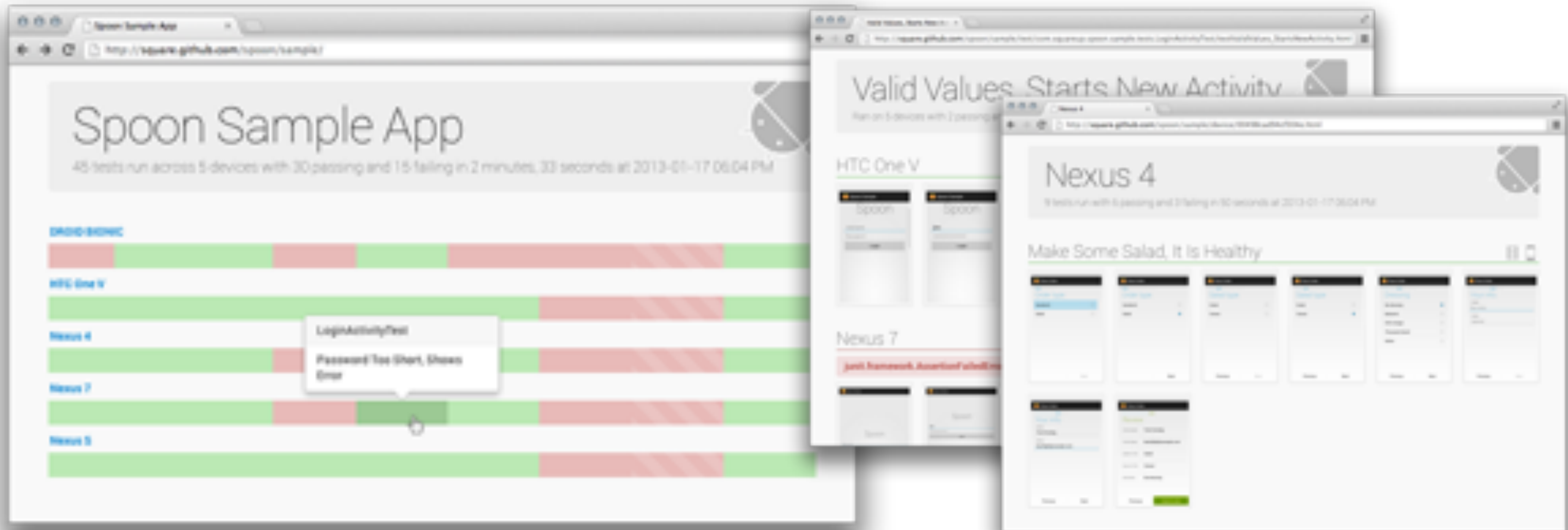
-:

- Не умеют работать с flaky тестами

<http://square.github.io/spoon/>

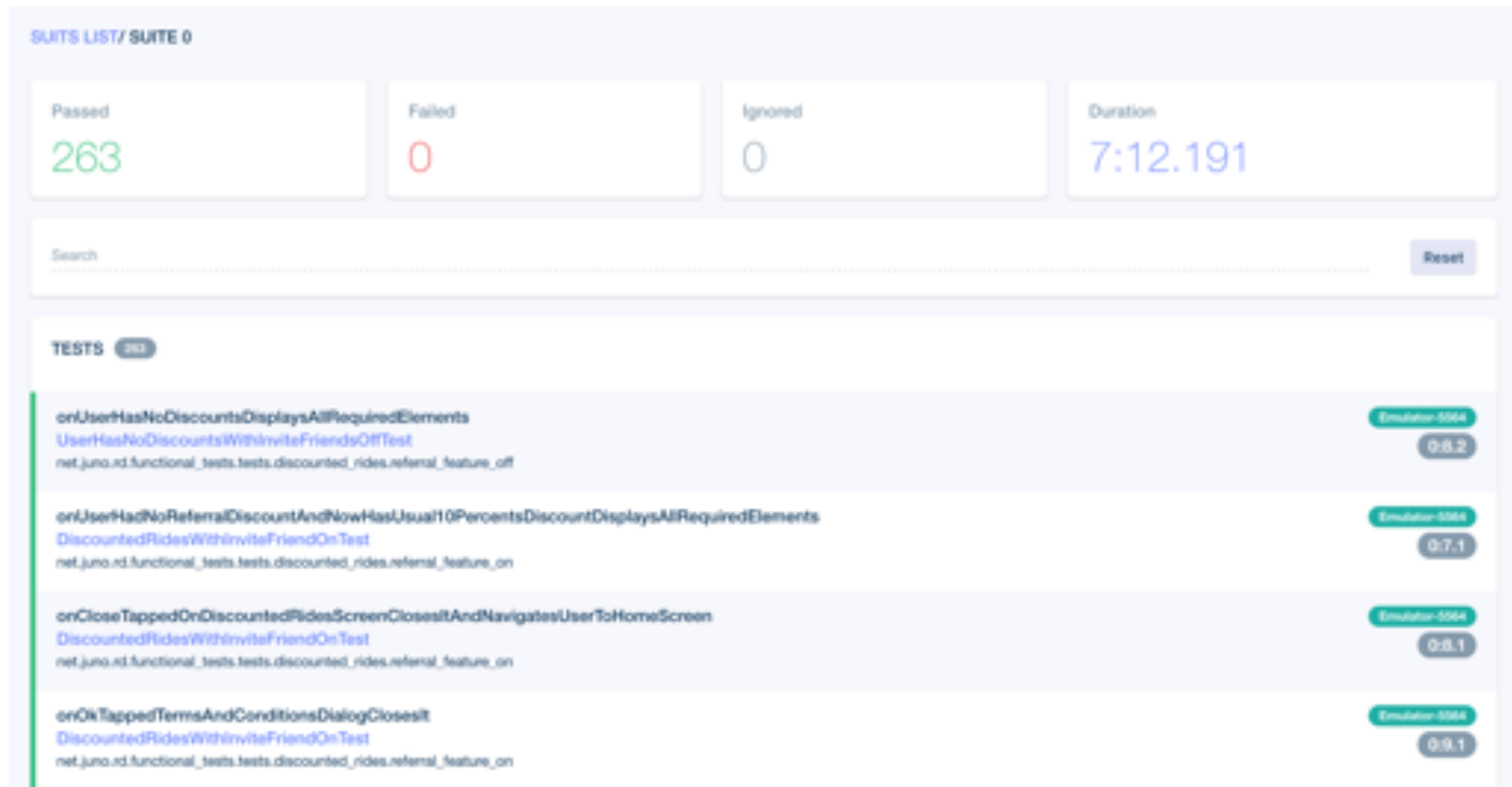
<https://github.com/gojuno/composer>

Spoon отчет



<http://square.github.io/spoon/>

Composer отчет



	AndroidJUnitRunner	Orchestrator	ConnectedAndroidTest	Spoon/Composer
Каждый тест - отдельный инстанс	-	+	/	+
Возможность очистки состояния	-	+	/	+
Требует установленных Google play services	-	+	/	+ -
Распараллеливание	+ -	+ -	/	+
Отчет	-	-	+	+

Marathon

Еще более продвинутое решение в запуске тестов, из основных плюсов:

+:

- Кроссплатформа (iOS + Android)
- Отдельный файл конфигурации
- Свой встроенный аналог orchestrator (больше не нужно устанавливать на устройства тестовые сервисы)
- Обработка flaku тестов из коробки
- Умное распараллеливание (выбираем как тесты должны запускаться – например, каждый тест 5 раз или на всех подключенных устройствах)
- Разбиение по группам версии ОС, архитектуры процессора и т.д.

-:

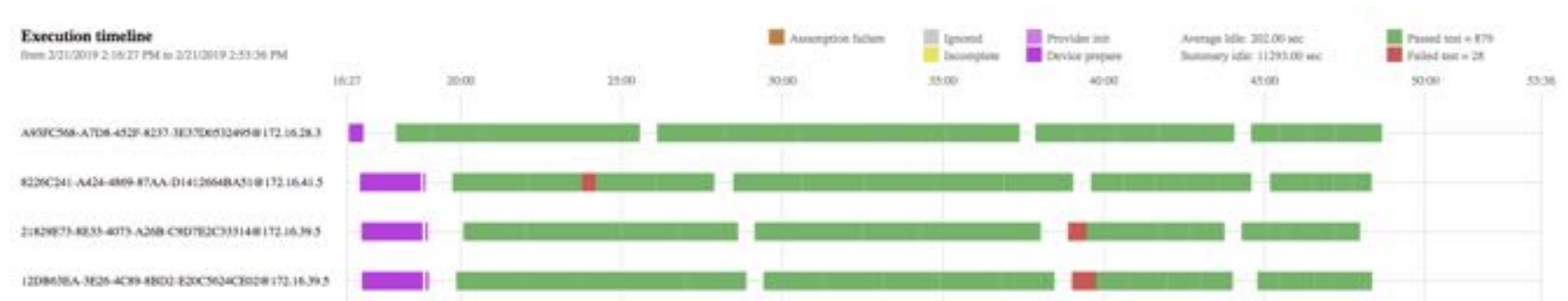
- Чуть более высокий порог входа
- Версия 0.4.1, до релиза пока далеко (однако во время использования критичных багов замечено не было)

<https://github.com/Malinskiy/marathon>

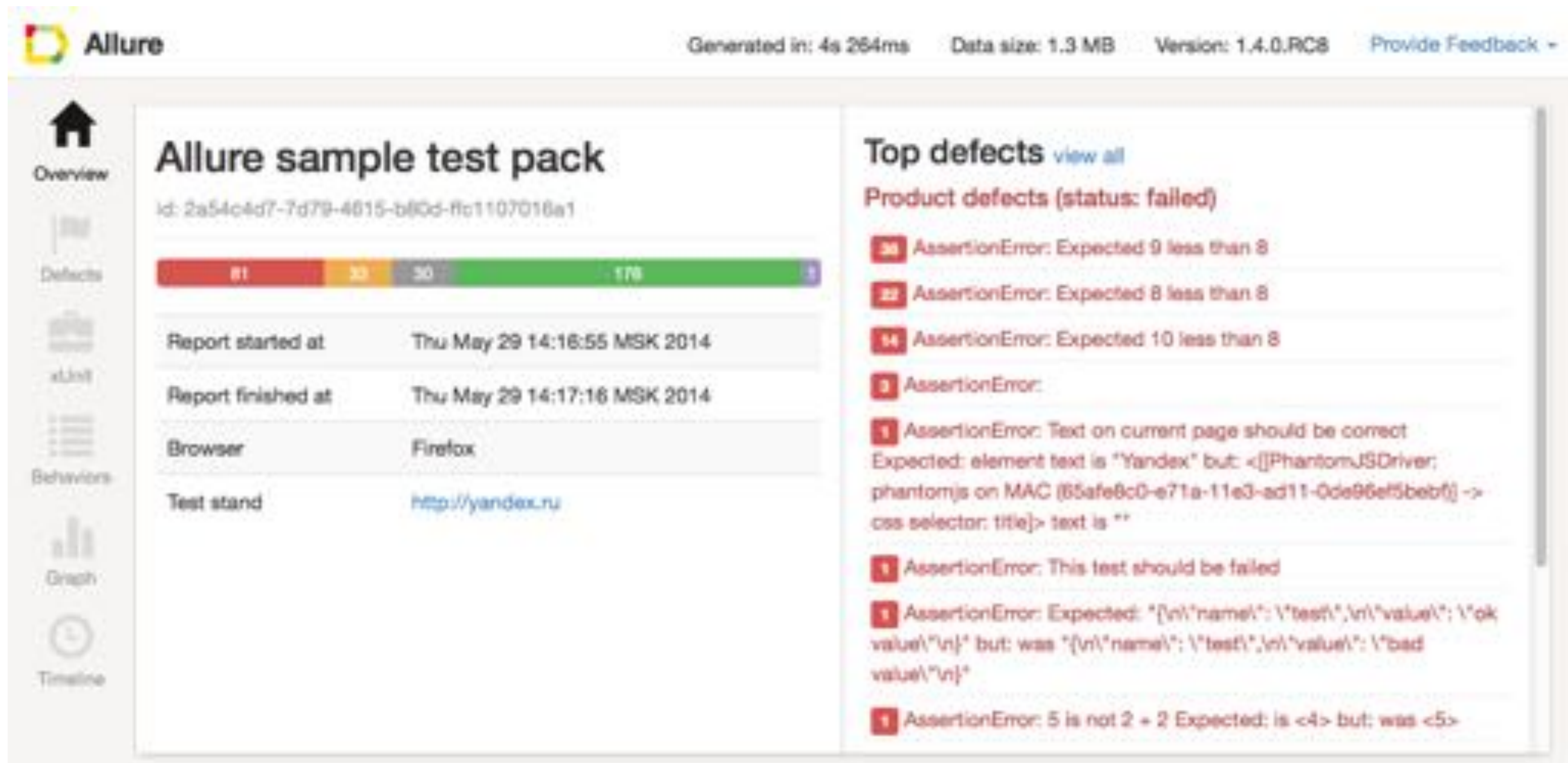
Marathon отчет

За основу отчета взят отчет от Composer, однако чуть изменен вид самого теста.
Теперь туда вставляется видео (или GIF в зависимости от версии OS) в случаях, если тест провалился.

Также есть отчет о распараллеливании:



Marathon + Allure = ❤️



	AndroidJUnitRunner	Orchestrator	ConnectedAndroidTest	Spoon/Composer	Marathon
Каждый тест - отдельный инстанс	-	+	/	+	+
Возможность очистки состояния	-	+	/	+	+
Требует установленных Google play services	-	+	/	+-	-
Распараллеливание	+-	+-	/	+	++
Отчет	-	-	+	+	+

Marathon где узнать больше?

Официальная документация:

<https://malinskiy.github.io/marathon/>

Статьи о работе Marathon:

<https://proandroiddev.com/marathon-chapter-1-97f295054cc4>

<https://proandroiddev.com/marathon-chapter-2-1cde95cfdb87>

Что-то еще?

- Firebase Test Lab
- Fork

Что было дальше?



Когда появились первые результаты, столкнулись с тем, что:

Espresso флекает

Логирование действий автотестов

Скриншоты при ошибках/асертах

А также ряд других серьезных вещей

Топ проблем

И их возможное решение

Пример

1) 3 фичи

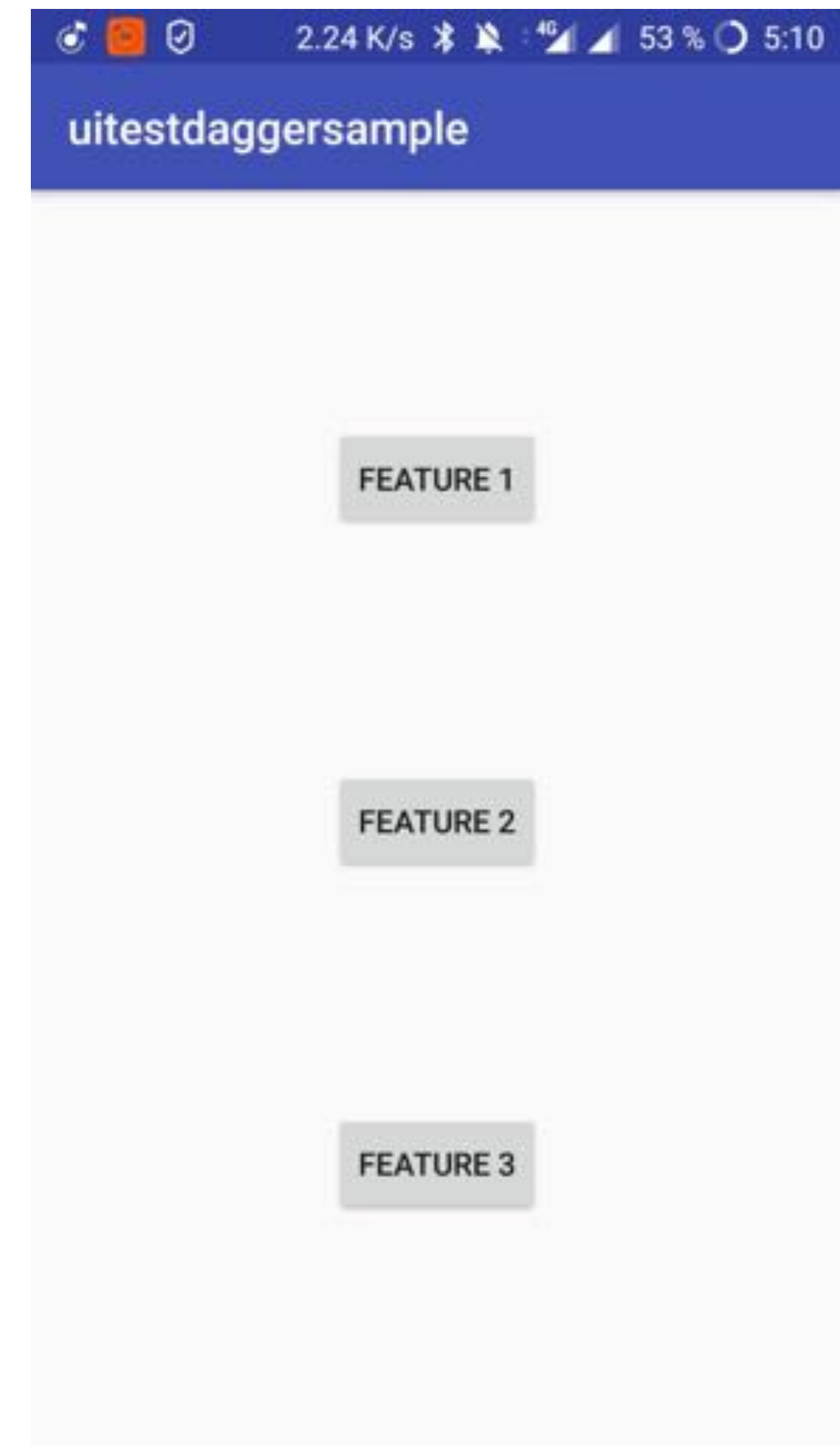
- 2 завязаны на 1 компонент
- 1 с прохождением FRW и инициализацией

2) Тесты на Какао

- Использован PageObject

3) Решены основные проблемы

- Сброс состояния
- Подмена классов
- Асинхронная операция



Как побороть flaky тест

```
MainScreen {  
    scanButton {  
        isVisible()  
        click()  
    }  
}
```

Как побороть flaky тест

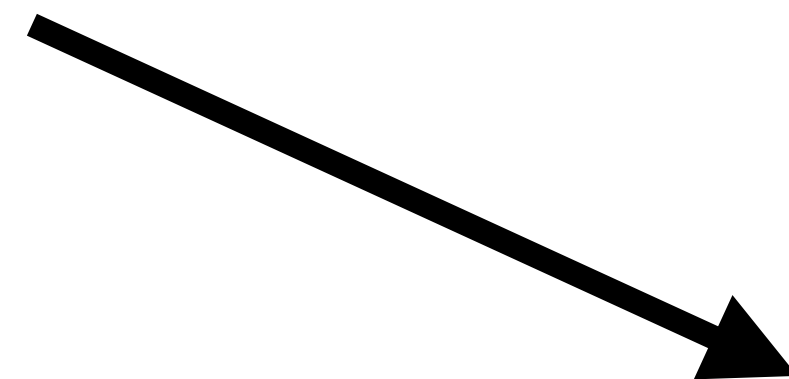
```
fun KButton.clickWithWait() {  
    var bool = true  
    while (bool) {  
        try {  
            click()  
            bool = false  
        } catch (e: Exception) {  
            idle(200)  
        }  
    }  
}
```

Как побороть flaky тест

```
fun <T> attempt(action: () -> T, maxAttempt: Int) {  
    var attempt = 0  
    while (attempt < maxAttempt) {  
        try {  
            action.invoke()  
            break  
        } catch (e: Throwable) {  
            if (attempt == maxAttempt - 1) throw e  
            attempt++  
            Thread.sleep(200)  
        }  
    }  
}
```


Как побороть flaky тест

```
MainScreen {  
    scanButton {  
        isVisible()  
        click()  
    }  
}
```



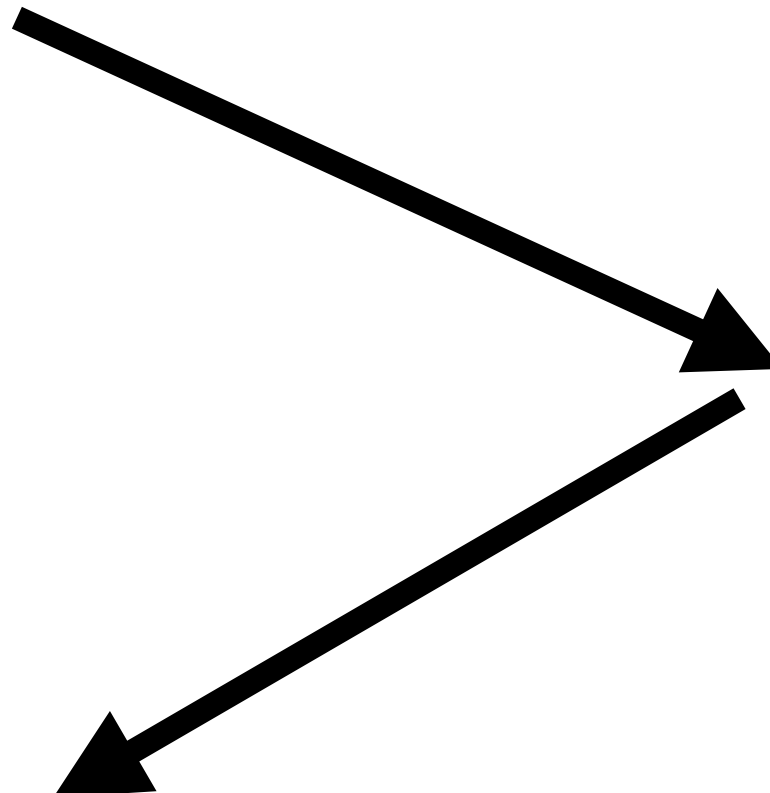
```
fun BaseActions.safeClick() {  
    attempt { click() }  
}
```

```
fun BaseAssertions.safeIsVisible() {  
    attempt { isVisible() }  
}
```

Как побороть flaky тест

```
MainScreen {  
    scanButton {  
        isVisible()  
        click()  
    }  
}
```

```
MainScreen {  
    scanButton {  
        safeIsVisible()  
        safeClick()  
    }  
}
```



```
fun BaseActions.safeClick() {  
    attempt { click() }  
}
```

```
fun BaseAssertions.safeIsVisible() {  
    attempt { isVisible() }  
}
```

Логирование действий / скриншот

```
MainScreen {  
    scanButton {  
        safeIsVisible()  
        safeClick()  
    }  
}
```

```
fun BaseActions.safeClick() {  
    attempt { click() }  
}  
  
fun BaseAssertions.safeIsVisible() {  
    attempt { isVisible() }  
}
```

Логирование действий/скриншот

```
MainScreen {  
    scanButton {  
        safeIsVisible()  
        safeClick()  
    }  
}
```

```
fun BaseActions.safeClick() {  
    attempt { click() }  
    logger.i("Some info")  
}  
  
fun BaseAssertions.safeIsVisible() {  
    try {  
        attempt { isVisible() }  
        logger.i("Some info")  
    } finally {  
        screenshots.makeIfPossible("Some tag")  
    }  
}
```

Очистка состояния приложения

3 теста, просто запускают каждую активити фичи

- 1) Один запуск Application класса
- 2) Общее состояние между тестами
- 3) Время прогона ~5 секунд

Pm clear

3 теста, просто запускают каждую активити фичи

1) Запуск Application класса на каждый тест

2) Никакого общего состояния между тестами

3) Время прогона ~8 секунд

Подмена зависимостей

```
@Component(modules = {AuthModule.class, /* ... */})
interface MyApplicationComponent { /* ... */ }

@Module
class AuthModule {
    @Provides AuthManager authManager(AuthManagerImpl impl) {
        return impl;
    }
}

class FakeAuthModule extends AuthModule {
    @Override
    AuthManager authManager(AuthManagerImpl impl) {
        return new FakeAuthManager();
    }
}

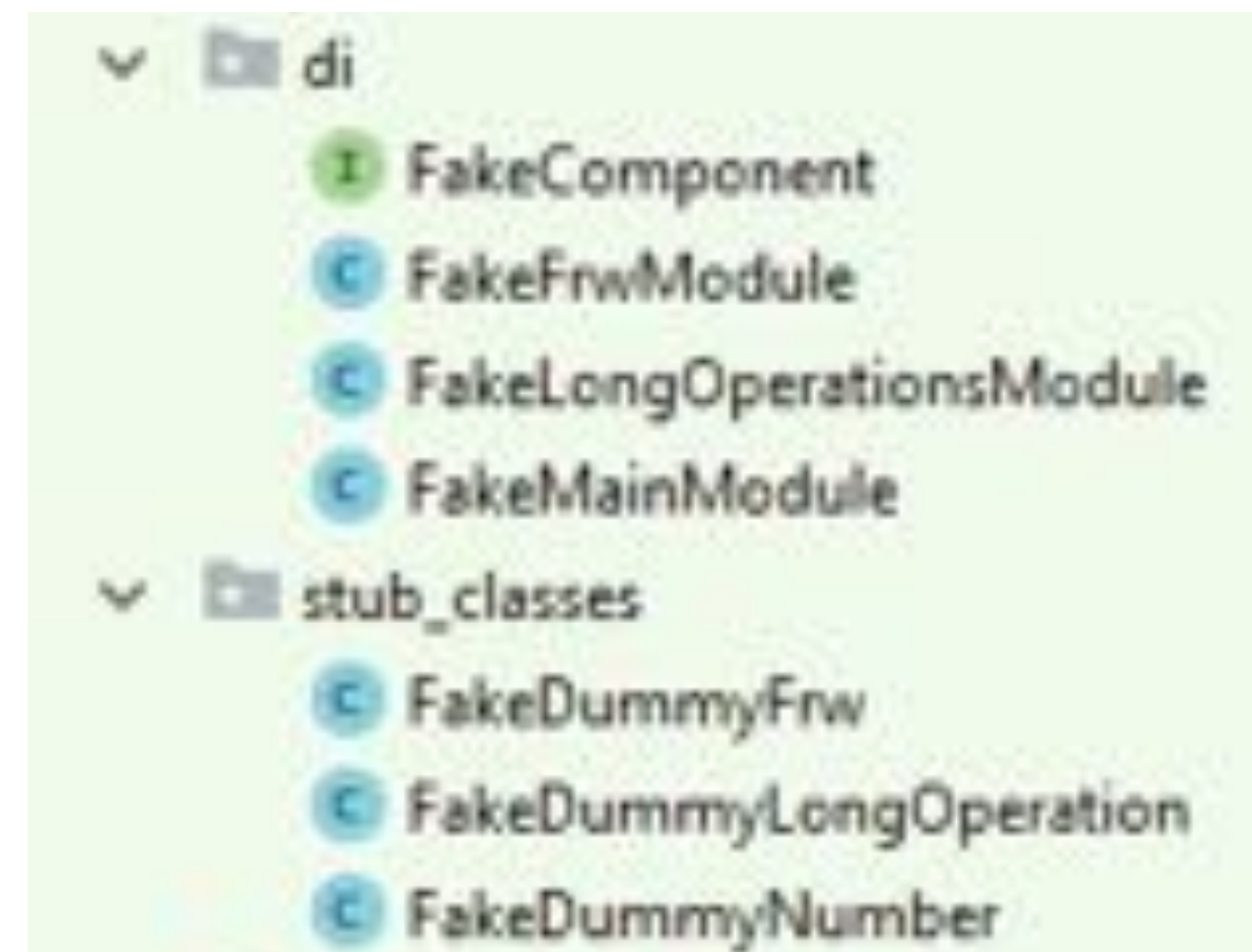
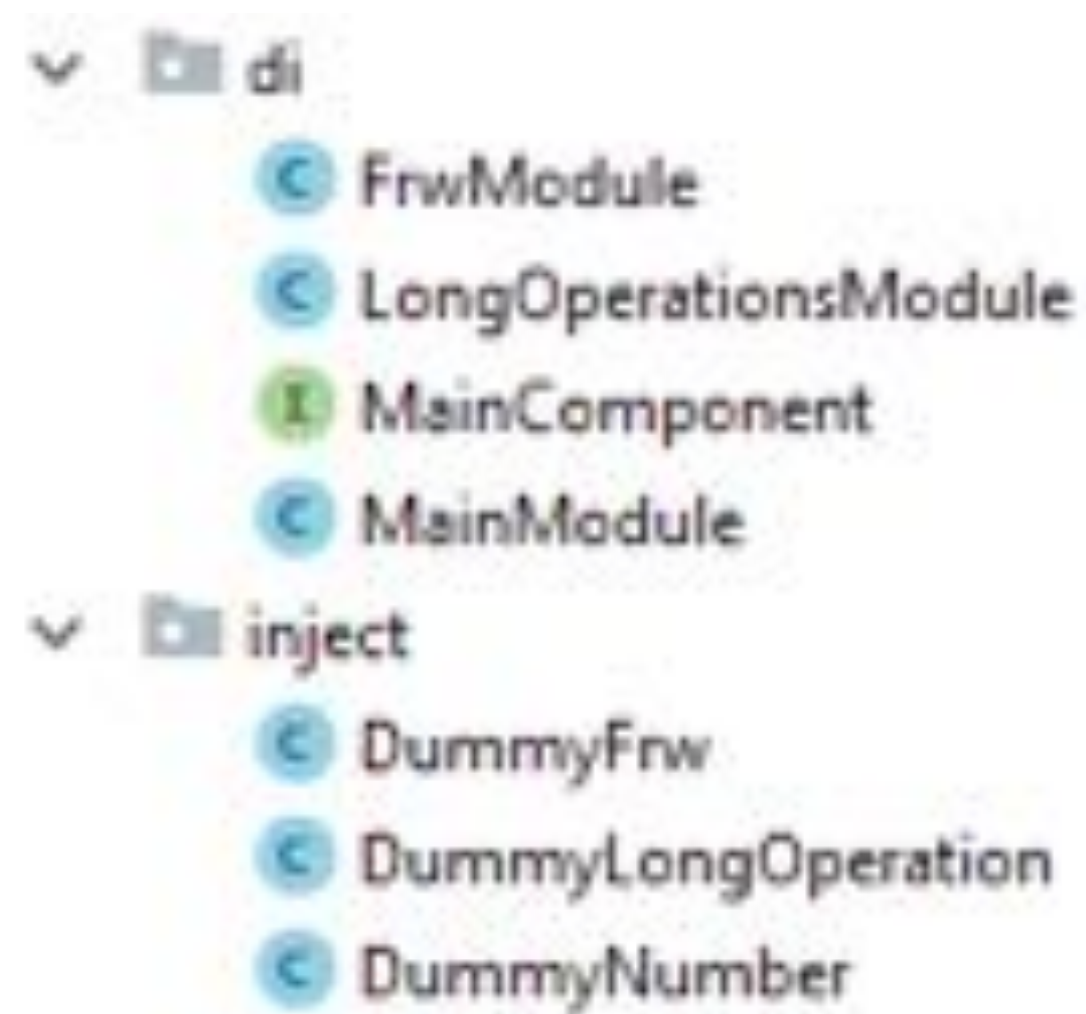
MyApplicationComponent testingComponent = DaggerMyApplicationComponent.builder()
    .authModule(new FakeAuthModule())
    .build();
```

Подмена зависимостей

```
@Component(modules = {
    OAuthModule.class, // real auth
    FooServiceModule.class, // real backend
    OtherApplicationModule.class,
    /* ... */ })
interface ProductionComponent {
    Server server();
}

@Component(modules = {
    FakeAuthModule.class, // fake auth
    FakeFooServiceModule.class, // fake backend
    OtherApplicationModule.class,
    /* ... */ })
interface TestComponent extends ProductionComponent {
    FakeAuthManager fakeAuthManager();
    FakeFooService fakeFooService();
}
```


Подмена зависимостей



@Before

```
fun init() {  
    mComponent = DaggerFakeComponent.builder()  
        .fakeMainModule(FakeMainModule(getApp())) .build()  
    getApp().mainComponent = mComponent  
    mComponent.inject(this)  
}
```

```

@Test
fun testThirdFeature() {
    mainScreen {
        toThirdFeatureButton {
            click()
        }
    }
    mThirdFeatureFrwScreen {
        proceedButton {
            click()
        }
    }
    mThirdFeatureScreen {
        featureDisclaimer {
            isVisible()
        }
    }
}

```

```

@Test
fun testThirdFeatureFake() {
    rule.launchActivity(null)
    mainScreen {
        toThirdFeatureButton {
            click()
        }
    }
    mThirdFeatureFrwScreen {
        proceedButton {
            click()
        }
    }
    mThirdFeatureScreen {
        featureDisclaimer {
            isVisible()
        }
    }
}

```

Kaspresso

Kaspresso

```
@Test
fun test() {
    before {
        activityTestRule.launchActivity(null)
    }.after {
    }
    run {
        step("Open Simple Screen") {
            testLogger.i("I am test logger")
            device.screenshots.take("Additional_screenshot")
            MainScreen {
                simpleButton {
                    isVisible()
                    click()
                }
            }
        }
    }
}
```

```
class MyTestCase {  
  
    @Test  
    fun someTest() {  
        MainScreen {  
            scanButton {  
                isVisible()  
                click()  
            }  
        }  
        // ...  
    }  
}
```

Kakao + Kaspreso

```
class MyTestCase : TestCase() {
```

```
    @Test
```

```
    fun someTest() {  
        MainScreen {  
            scanButton {  
                isVisible()  
                click()  
            }  
        }  
    }
```

```
    // ...  
}
```

```
}
```

<https://github.com/KasperskyLab/Kaspresso>

**[https://habr.com/ru/company/kaspersky/
blog/467617/](https://habr.com/ru/company/kaspersky/blog/467617/)**



Kaspresso - развитие

Куда пошло развитие

Accessibility

Activities

Apps

Files

Internet

Permissions

Screenshots

Permissions

```
/**
 * An interface to work with permissions.
 */
interface Permissions {

    /**
     * Passes the permission-requesting permissions dialog and allows permissions.
     */
    fun allowViaDialog()

    /**
     * Passes the permission-requesting permissions dialog and denies permissions.
     */
    fun denyViaDialog()
}
```

Internet

```
/**
 * An interface to work with internet settings.
 */
interface Internet {

    /**
     * Enables wi-fi and mobile data using adb.
     */
    fun enable()

    /**
     * Disables wi-fi and mobile data using adb.
     */
    fun disable()

    /**
     * Toggles only wi-fi. Note: it works only if flight mode is off.
     */
    fun toggleWiFi(enable: Boolean)
}
```

Files

```
/**
 * An interface to work with file permissions.
 */
interface Files {

    /**
     * Performs adb push.
     *
     * @param serverPath a file path relative to the server directory.
     * @param devicePath a path to copy.
     */
    fun push(serverPath: String, devicePath: String)
}
```

Adb + Espresso = ?

Зачем нам adb во время тестов? ^R

Стандартные команды:

- установка других арк во время теста
- push / pull
- выставление системных настроек

Зачем нам adb во время тестов? ^R

Нестандартные команды **adb emu** (работают только с эмуляторами):

- установка геопозиции
- выставление скорости сети, задержки
- симулировать настоящих звонков
- выставить значения акселерометра
- имитация работы с отпечатком пальца
- и т.д.

<https://developer.android.com/studio/run/emulator-console>

<https://github.com/KasperskyLab/AdbServer>



Заключение

Заключение

Автотесты:

- Огромный пласт работы
- Большие трудозатраты
- Результат того стоит

Заключение

Где почерпнуть еще информации?

- 1) Митап по UI-тестированию в Авито 11.08.2018 <https://www.youtube.com/watch?v=wZnilAhuLTE>
- 2) PageObject + UiAutomator <https://habr.com/post/416397/>
- 3) Какао статья на хабре <https://habr.com/post/339664/>
- 4) Android Dev Podcast #60 <https://androiddev.apptractor.ru/android-dev-podcast-60/>
- 5) Первые попытки решить большинство проблем автотестов <https://github.com/v1sar/UiTestApp>
- 6) Выступление меня и Жени на Mobius 2019 SPb https://www.youtube.com/watch?v=q_8UUhVDV7c
- 7) Анонс релиза Kaspresso и AdbServer <https://www.youtube.com/watch?v=cTykctRSmuA>

Мовчан Дмитрий

Android Developer @ Revolut

dmitry.movchan@revolut.com

github.com/v1sar

t.me/@v1sar