Automatisches Testen von Android Apps

Katrin Rose

HUCK IT, Roßdorf

02.02.2018

Inhalt

1. Motivation
2. Grundlagen
3. Androidtesting
   - Unittests
   - Instrumented Tests

Motivation

Qualität

Automatische Tests sind nötig um qualitative Software zu liefern. Software ist heute zu komplex um sich auf ein paar manuelle Tests zu beschränken. Unerwünschte Seiteneffekte werden sonst nicht gefunden.

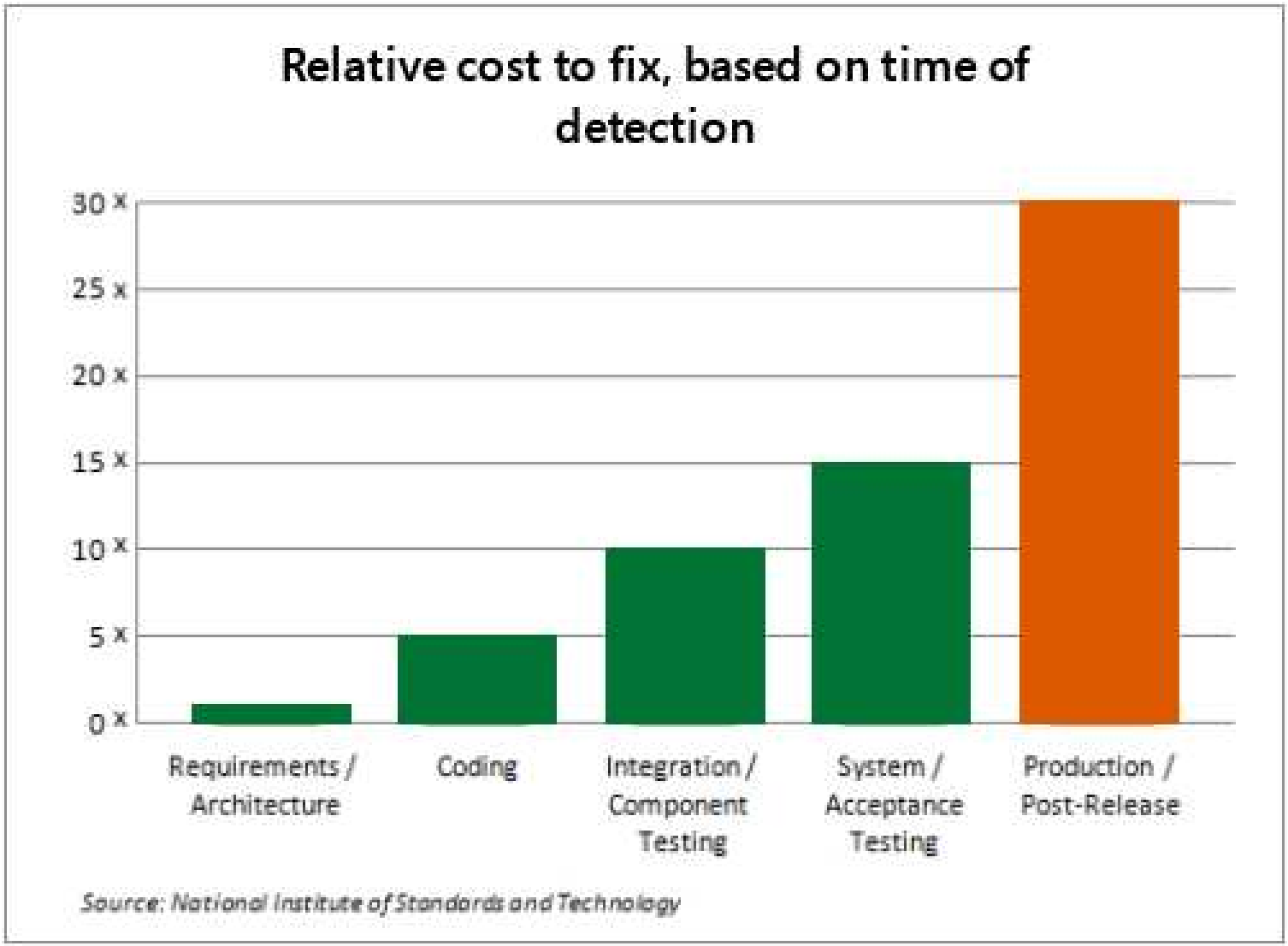
Sicherheit

Wir stecken mehr Aufwand in die Wartung der Software als in die Neuentwicklung. Tests geben Sicherheit bei Refactoring, Bugfixing und Erweiterung bestehender Funktionalität.

Wirtschaftlichkeit

Je später ein Fehler gefunden wird, desto teurer ist er → Grafik nächste Folie.

Motivation: Kostenentwicklung eines Softwarefehlers



Grundlagen

Unittests

Testen eine "Unit" des Codes, d.h. ein bestimmtes Verhalten/Feature. Tests sind simpel, kurz und schnell. Fehler werden sehr schnell gefunden.

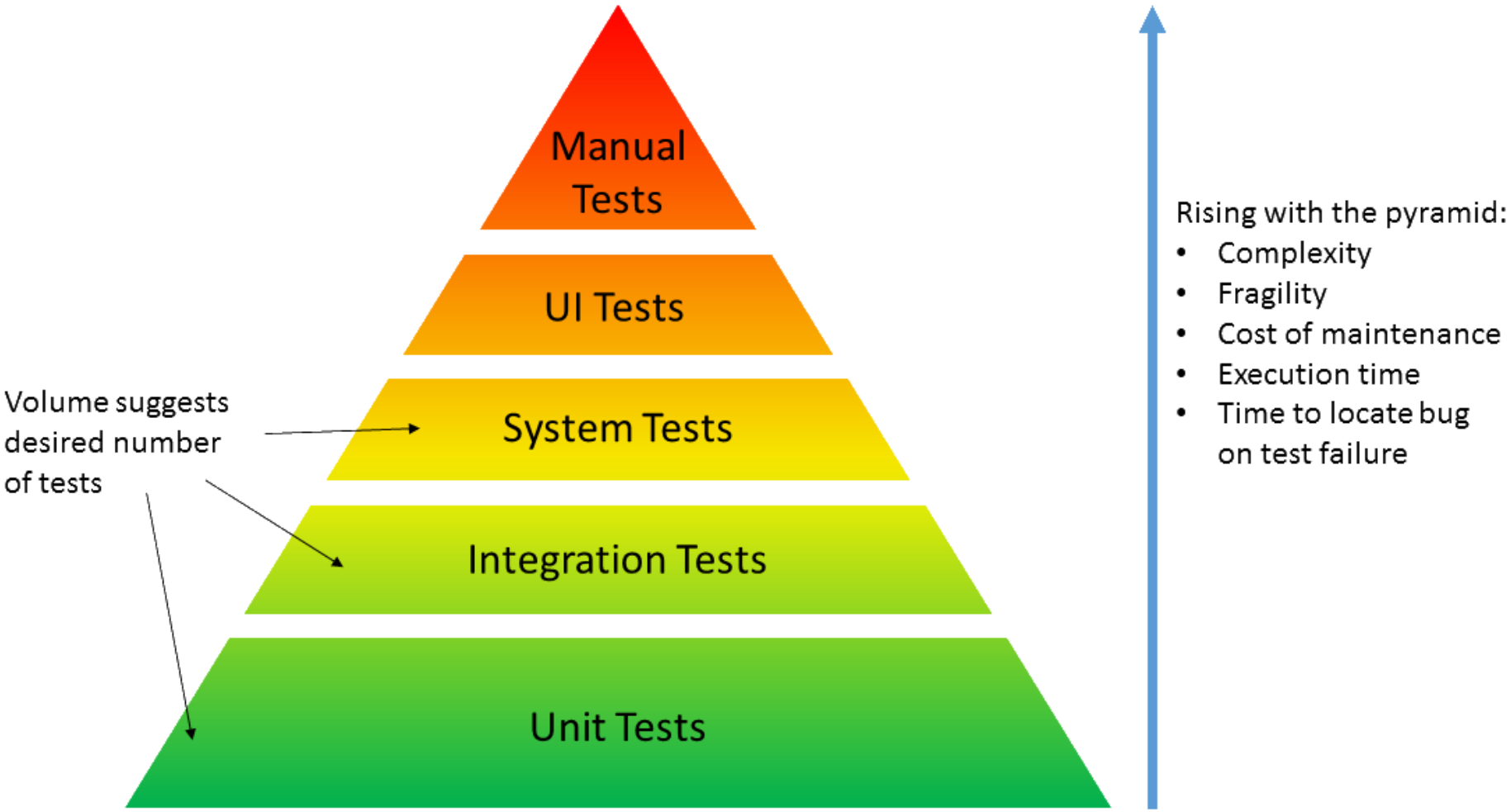
Integrations/Systemtests

Testen das Verhalten mehrerer Komponenten. Tests sind komplexer und haben eine längere Laufzeit. Fehlersuche etwas aufwändiger.

Oberflächentests (UI Tests)

Testen die Interaktionen auf der Benutzeroberfläche (Buttons, Textfelder, etc.). Oberflächentests sind die teuersten (Entwicklungs- und Laufzeit) und komplexesten automatischen Tests. Fehlersuche am aufwendigsten.

Grundlagen: Testpyramide



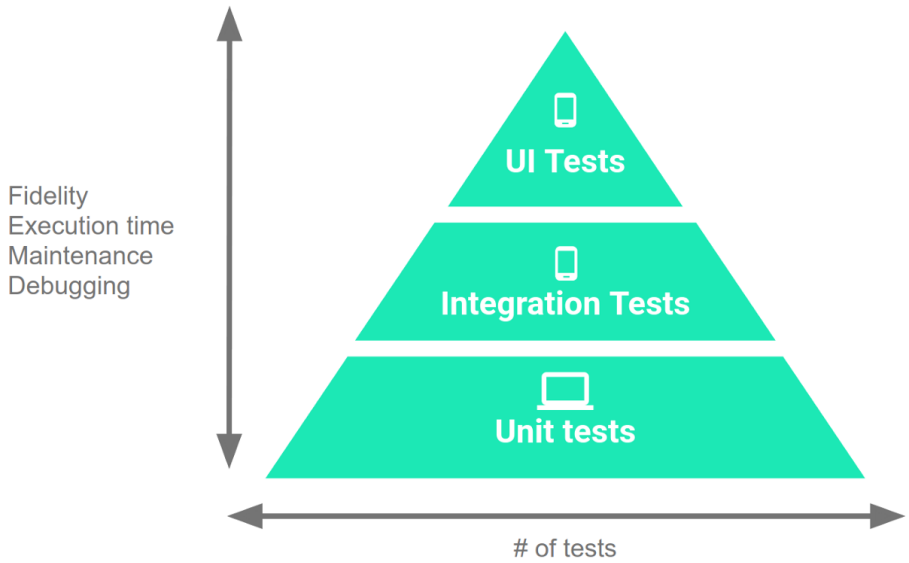
Androidtesting

In Android Apps unterscheiden wir zwischen

- Java-basiertem Verhalten
- Android-basiertem Verhalten

Daher gibt es für jedes Verhalten eigene Tests

- Unittests auf lokaler JVM (JUnit4)
- Integrations/UI-Tests auf (virtuellem) Android Gerät



→ large tests
(complete UI workflow)

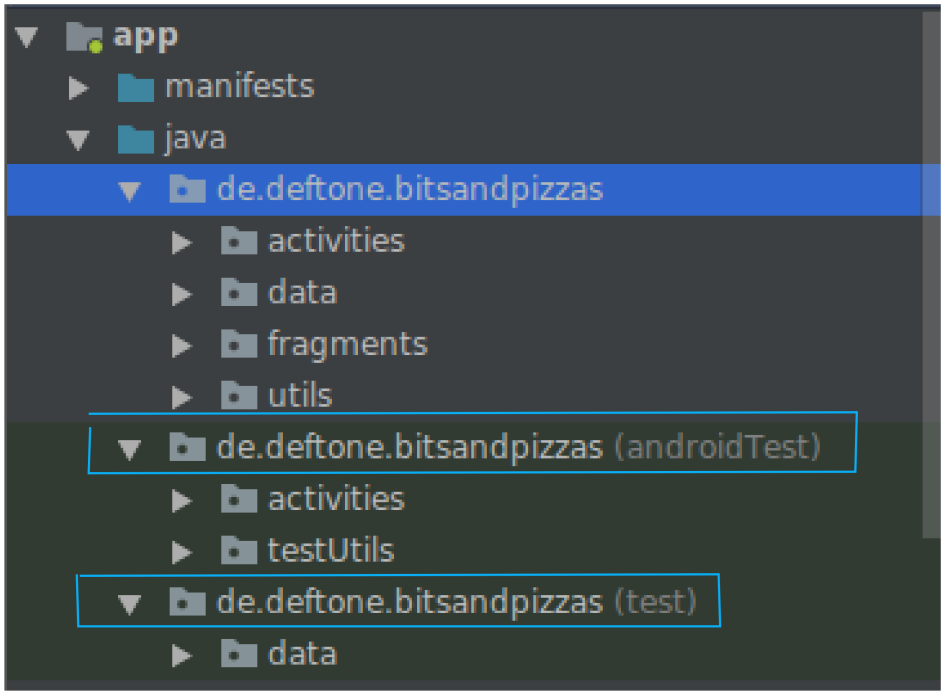
→ medium tests
(integrate several components)

→ small tests

Androidtesting: Android Studio

App, Unit und UI-Tests werden mit dem Android Studio verwaltet (geschrieben, gedebugged und ausgeführt)

- Appname: Paketstruktur mit source code
- Appname (androidTest): Paketstruktur mit UI Tests (Instrumented tests)
- Appname (test): Paketstruktur mit Unittests



Androidtesting: Unittests

- reine "Javatests", laufen auf lokaler JVM, kein Android Device nötig

Androidtesting: Unittests

- reine "Javatests", laufen auf lokaler JVM, kein Android Device nötig
- nutzen JUnit4 $\rightarrow$ Assert benutzen zum Prüfen der Testkriterien
  - assertEqual(object_expected, test_object);
  - assertNull(test_object);
  - assertTrue(test_object); u.v.m.

Androidtesting: Unittests

- reine "Javatests", laufen auf lokaler JVM, kein Android Device nötig
- nutzen JUnit4 → Assert benutzen zum Prüfen der Testkriterien
  - assertEqual(object_expected, test_object);
  - assertNull(test_object);
  - assertTrue(test_object); u.v.m.
- Testreihenfolge zufällig
  - Tests dürfen nicht von einander abhängen
  - Tests dürfen sich nicht beeinflussen
  - → vor jedem Test gleicher Ausgangszustand
  - → dafür Methoden mit @Before und @After Annotation nutzen (laufen vor bzw. nach jedem Test)

Androidtesting: Unittests

- reine "Javatests", laufen auf lokaler JVM, kein Android Device nötig
- nutzen JUnit4 → Assert benutzen zum Prüfen der Testkriterien
  - assertEqual(object_expected, test_object);
  - assertNull(test_object);
  - assertTrue(test_object); u.v.m.
- Testreihenfolge zufällig
  - Tests dürfen nicht von einander abhängen
  - Tests dürfen sich nicht beeinflussen
  - → vor jedem Test gleicher Ausgangszustand
  - → dafür Methoden mit @Before und @After Annotation nutzen (laufen vor bzw. nach jedem Test)
- @Test Annotation vor jeden Test

Androidtesting: Unittests

- reine "Javatests", laufen auf lokaler JVM, kein Android Device nötig
- nutzen JUnit4 → Assert benutzen zum Prüfen der Testkriterien
  - assertEqual(object_expected, test_object);
  - assertNull(test_object);
  - assertTrue(test_object); u.v.m.
- Testreihenfolge zufällig
  - Tests dürfen nicht von einander abhängen
  - Tests dürfen sich nicht beeinflussen
  - → vor jedem Test gleicher Ausgangszustand
  - → dafür Methoden mit @Before und @After Annotation nutzen (laufen vor bzw. nach jedem Test)
- @Test Annotation vor jeden Test
- statische Methode mit @BeforeClass (@AfterClass) läuft einmal vor (nach) allen Tests in der Testklasse

Androidtesting: Unittests

- reine "Javatests", laufen auf lokaler JVM, kein Android Device nötig
- nutzen JUnit4 → Assert benutzen zum Prüfen der Testkriterien
  - assertEqual(object_expected, test_object);
  - assertNull(test_object);
  - assertTrue(test_object); u.v.m.
- Testreihenfolge zufällig
  - Tests dürfen nicht von einander abhängen
  - Tests dürfen sich nicht beeinflussen
  - → vor jedem Test gleicher Ausgangszustand
  - → dafür Methoden mit @Before und @After Annotation nutzen (laufen vor bzw. nach jedem Test)
- @Test Annotation vor jeden Test
- statische Methode mit @BeforeClass (@AfterClass) läuft einmal vor (nach) allen Tests in der Testklasse
- Testabdeckung ermittelbar: Run 'testKlasse' with Coverage

Androidtesting: Auszug aus einer Unittestklasse

```java
public class ExerciseDetailAddPointsTest {

    private SharedPreferences sharedPreferencesDates;
    private SharedPreferences.Editor sharedPreferencesDatesEditor;
    private ExerciseDetailAddPoints sut;

    @Before
    public void setUp() throws Exception {
        this.sharedPreferencesDates = mock(SharedPreferences.class);
        this.sut = new ExerciseDetailAddPoints(sharedPreferencesDates, sharedPreferencesPoints);

        when(sharedPreferencesDates.edit()).thenReturn(sharedPreferencesDatesEditor);
    }

    /** pure java unit tests for getDateKey (AAA)**/
    @Test
    public void noDatesInSharedPrefs() {
        //arrage
        Set<String> sharedPrefsTestData = new HashSet<>();
        long currentTime = 1514761200000l; //1.1.2018

        //act
        String key = sut.getDateKey(currentTime, sharedPrefsTestData);

        //assert
        assertEquals(String.valueOf(currentTime), key);
    }

    @Test
```

Androidtesting: UI oder Instrumented Tests

- laufen auf realem oder virtuellem Android Gerät

Androidtesting: UI oder Instrumented Tests

- laufen auf realem oder virtuellem Android Gerät
- haben die volle Android Umgebung zur Verfügung

Androidtesting: UI oder Instrumented Tests

- laufen auf realem oder virtuellem Android Gerät
- haben die volle Android Umgebung zur Verfügung
- nutzen ebenfalls JUnit (@Test, @Before, @BeforeClass, ...)

Androidtesting: UI oder Instrumented Tests

- laufen auf realem oder virtuellem Android Gerät
- haben die volle Android Umgebung zur Verfügung
- nutzen ebenfalls JUnit (@Test, @Before, @BeforeClass, ...)
- Annotation @RunWith(AndroidJUnit4.class) vor Testklasse, damit der Android JUnit test runner benutzt werden kann

Androidtesting: UI oder Instrumented Tests

- laufen auf realem oder virtuellem Android Gerät
- haben die volle Android Umgebung zur Verfügung
- nutzen ebenfalls JUnit (@Test, @Before, @BeforeClass, ...)
- Annotation @RunWith(AndroidJUnit4.class) vor Testklasse, damit der Android JUnit test runner benutzt werden kann
- Testregel mit Annotation @Rule in jeder Testklasse
  - startet (beendet) die Targetactivity vor (nach) jedem Test
  - innerhalb der Testrulemethode können @Before und @After Methoden benutzt werden

Androidtesting: UI oder Instrumented Tests

- laufen auf realem oder virtuellem Android Gerät
- haben die volle Android Umgebung zur Verfügung
- nutzen ebenfalls JUnit (@Test, @Before, @BeforeClass, ...)
- Annotation @RunWith(AndroidJUnit4.class) vor Testklasse, damit der Android JUnit test runner benutzt werden kann
- Testregel mit Annotation @Rule in jeder Testklasse
  - startet (beendet) die Targetactivity vor (nach) jedem Test
  - innerhalb der Testrulemethode können @Before und @After Methoden benutzt werden
- nutzen Espresso um mit den Views der Activity zu interagieren
  - view finden, z.B. mit onView(withId())
  - auf view interagieren, z.B. mit perform(click())
  - view prüfen, z.B. mit check(matches(()))

Androidtesting: UI oder Instrumented Tests

- laufen auf realem oder virtuellem Android Gerät
- haben die volle Android Umgebung zur Verfügung
- nutzen ebenfalls JUnit (@Test, @Before, @BeforeClass, ...)
- Annotation @RunWith(AndroidJUnit4.class) vor Testklasse, damit der Android JUnit test runner benutzt werden kann
- Testregel mit Annotation @Rule in jeder Testklasse
  - startet (beendet) die Targetactivity vor (nach) jedem Test
  - innerhalb der Testrulemethode können @Before und @After Methoden benutzt werden
- nutzen Espresso um mit den Views der Activity zu interagieren
  - view finden, z.B. mit onView(withId())
  - auf view interagieren, z.B. mit perform(click())
  - view prüfen, z.B. mit check(matches(()))
  - Hamcrest Matcher, RecyclerViewActions, eigene Matcher u.v.m. nötig um alles testen zu können
  - Wichtig: wenn eine view nicht sichtbar ist, so kann NICHT mit ihr interagiert werden $\rightarrow$ Test schlägt fehl

Androidtesting: Auszug aus einer UI-Testklasse

```java
@RunWith(AndroidJUnit4.class)
public class ExerciseDetailActivityTest {

    @Rule
    public ActivityTestRule<ExerciseDetailActivity> mActivityRule =
            new ActivityTestRule<>(ExerciseDetailActivity.class, initialTouchMode: true, launchActivity: false);

    @Test
    public void clickButtonCheckPoints() throws InterruptedException {
        //arrange: start intent and reset shared preferences
        int position = 0;
        startIntent(position, TYPE_LEG_EXERCISES);
        SharedPreferences sharedPreferencesDates = mActivityRule.getActivity().getSharedPreferences(PREFS_DATES,
        SharedPreferences.Editor sharedPreferencesDatesEditor = sharedPreferencesDates.edit();
        sharedPreferencesDatesEditor.clear().apply();

        //act: scroll down to button and click
        onView(withId(R.id.detail_scrollview)).perform(ViewActions.swipeUp());
        onView(withId(R.id.weight_button)).perform(click());

        //assert: check toast is visible with the correct points
        int[] points = LEG_EXERCISES[position].getWeight();
        String toastText = mActivityRule.getActivity().getString(R.string.points_today_1) + points[0]
                + mActivityRule.getActivity().getString(R.string.points_today_2);
        onView(withText(toastText))
                .inRoot(withDecorView(not(is(mActivityRule.getActivity().getWindow().getDecorView()))))
                .check(matches(isDisplayed()));
    }
```

Fazit

- automatische Tests sind ein Qualitätsmerkmal: sie liefern Sicherheit, dass die Software so funktioniert, wie sie spezifiziert wurde
- kosten Zeit beim Entwickeln - aber sparen Zeit beim (fehlerärmeren) Releasen

Fazit

- automatische Tests sind ein Qualitätsmerkmal: sie liefern Sicherheit, dass die Software so funktioniert, wie sie spezifiziert wurde
- kosten Zeit beim Entwickeln - aber sparen Zeit beim (fehlerärmeren) Releasen
- UI-Tests für Android sind kein Hexenwerk sondern (relativ einfach) umzusetzen

Fazit

- automatische Tests sind ein Qualitätsmerkmal: sie liefern Sicherheit, dass die Software so funktioniert, wie sie spezifiziert wurde
- kosten Zeit beim Entwickeln - aber sparen Zeit beim (fehlerärmeren) Releasen
- UI-Tests für Android sind kein Hexenwerk sondern (relativ einfach) umzusetzen

Aber Vorsicht:

- auch automatische Tests werden keine fehlerfreie Software garantieren, 100% Testabdeckung sollte nicht das Ziel sein (unwirtschaftlich)
- Prioritäten setzen: die wichtigsten und häufigsten Usecases sollten abgedeckt sein

Fazit

- automatische Tests sind ein Qualitätsmerkmal: sie liefern Sicherheit, dass die Software so funktioniert, wie sie spezifiziert wurde
- kosten Zeit beim Entwickeln - aber sparen Zeit beim (fehlerärmeren) Releasen
- UI-Tests für Android sind kein Hexenwerk sondern (relativ einfach) umzusetzen

Aber Vorsicht:

- auch automatische Tests werden keine fehlerfreie Software garantieren, 100% Testabdeckung sollte nicht das Ziel sein (unwirtschaftlich)
- Prioritäten setzen: die wichtigsten und häufigsten Usecases sollten abgedeckt sein
- wichtig ist außerdem, dass man sich am besten beim Entwickeln Gedanken über Akzeptanzkriterien (oder Tests) macht: Test Driven Development als oberste Kür - dann schreibt man nur den Code, der wirklich benötigt ist und hat direkt "black box" Tests

Danke für eure Aufmerksamkeit