UNIVERSITÄT
LEIPZIG

Softwaretechnikpraktikum

Testbericht

Gruppe:	nw19a
Mitglieder:	Thomas Pause, Sabine Lorius, Arik Korte, Martin George, Josephine Lange, Esther Prause, Anh Kiet Nguyen, Bärbel Hanle
Verantwortlich:	Thomas Pause
Betreuer:	Dr. Nicolas Wieseke
Tutor:	Martin Frühauf
Abgabedatum:	06.01.2020
Version:	0.5 (Vorprojekt)

Stand: 6. Januar 2020

Inhaltsverzeichnis

1 Verwendete Testframeworks

Für die Durchführung der Komponententests verwenden wir JUnit5, für die Integrationstests die zugehörige Erweiterung Mockito.
Systemtests, insbesondere die korrekte Funktionsüberprüfung des User-Interfaces und Ende-zu-Ende-Tests werden mit Espresso realisiert. Hierfür wird teilweise der Espresso Test Recorder verwendet.
Des Weiteren haben wir die *gradle.build*-Dateien auf App- und Projektebene so angepasst, dass zum Beispiel die Ergebnisse der Tests übersichtlicher erkennbar sind, indem auch bei bestandenen Tests die jeweiligen Bezeichner ausgegeben werden.
Die sehr nützliche Funktion von Kotlin, Testnamen als Strings mit Leerzeichen definieren zu können, sorgt für bessere Lesbarkeit.

2 GitLab CI

Wie im Qualitätssicherungskonzept vereinbart, werden bei jedem Push die drei Stages der GitLab Continuous Integration durchlaufen. Diese wurden in der Datei *.gitlab-ci.yml* definiert.
Es wird ein Build, das Linting und die Tests durchgeführt.
Dabei gibt es unterschiedliche Jobs für die verschiedenen Branches, welche mit Hilfe des *Gradle-Wrappers* durchgeführt werden.

3 Unit-Tests

Mit den Komponententests prüfen wir wichtige Standard- und Grenzfälle ab. Hierfür wurden in bislang 2 Testklassen verschiedene Tests entworfen. Die Parameter der zahlreichen Testfälle sind in den jeweiligen Testklassen ersichtlich. Im folgenden findet sich eine Auflistung der Testmethoden und ihr jeweiliges Ergebnis.

→ Konsolen-Output beim Aufruf von *./gradlew test*:

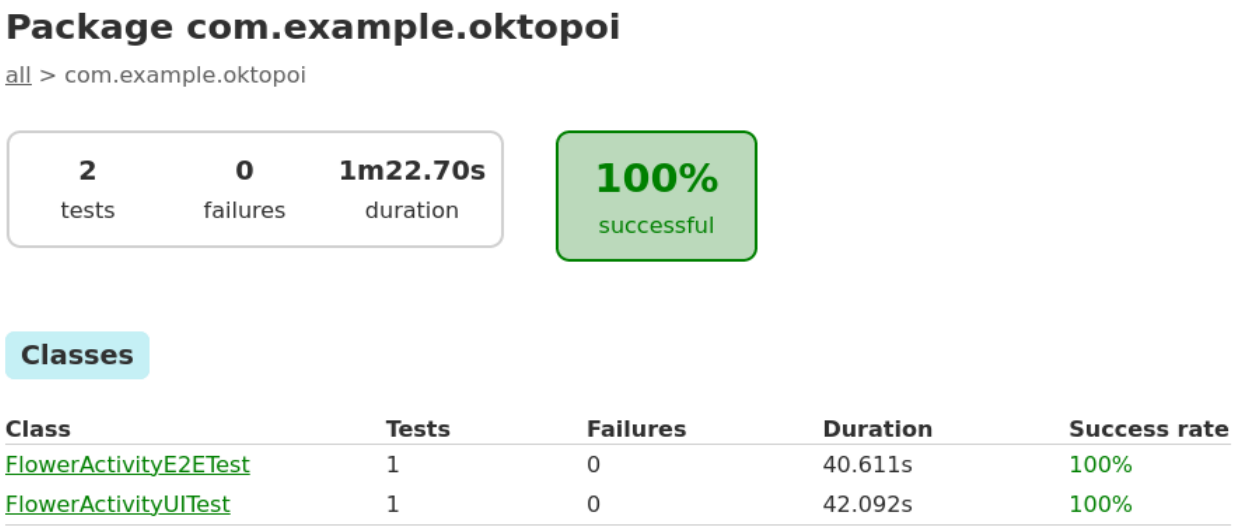
```
> Task :app:testReleaseUnitTest

com.example.oktopoi.FlowerActivityTest

  Test validateInput regex matching PASSED

com.example.oktopoi.FlowerCalcTest

  Test yHand to assert y position of hand at time t PASSED
  Test calcFlower() to assert list of flower coordinates PASSED
  Test xHand to assert x position of hand at time t PASSED
  Test yFlower to assert y position of poi at time t PASSED
  Test greatestCommonDivisor to assert greatest common divisor of freq1 and freq 2 PASSED
  Test calcHand() to assert list of hand coordinates PASSED
  Test xFlower to assert x position of poi at time t PASSED

SUCCESS: Executed 8 tests in 770ms
```

4 Integrationstests

Zum aktuellen Entwicklungsstand sind noch keine Integrationstests nötig. Dennoch haben wir uns bereits mit dem Konzept des Mockings mithilfe der JUnit5-Erweiterung *Mockito* befasst und an einem Dummy-Projekt geübt.

5 UI- und Systemtests

Diese Tests wurden mit dem Framework *Espresso* durchgeführt. Sie benötigen eine virtuelle oder physische Android-Instanz (also einen aktiven Emulator oder ein Endgerät) und werden daher aus Performance-Gründen nicht über die GitLab-CI automatisch ausgelöst.
Bislang wurden ein Ende-zu-Ende-Test und ein UI-Test entworfen und erfolgreich durchgeführt. Bei dem ersten handelt es sich um einen beliebigen Durchlauf durch die Funktionalitäten der Anwendung während der zweite die Benutzeroberfläche systematisch auf ihre definitionsgemäße Funktion überprüft.
Im Folgenden sind die Ergebnisse der Systemtests zu sehen, als Ausschnitt aus der im Browser dargestellten Datei *index.html* im Ordner *reports/androidTests/connected/*.



Package com.example.oktopoi

all > com.example.oktopoi

2	0	1m22.70s
tests	failures	duration

100% successful

Classes

Class	Tests	Failures	Duration	Success rate
FlowerActivityE2ETest	1	0	40.611s	100%
FlowerActivityUITest	1	0	42.092s	100%

6 Manuelle Tests

Zusätzlich zu den automatisierten Systemtests führten wir manuelle Tests durch, um die korrekte Funktionalität der Canvas zu überprüfen.
Als Referenz diente hierbei die Seite *https://www.desmos.com/calculator/bjqsd2veim*, auf der wir für 6 Settings die Ergebnisse gegenübergestellt haben (siehe Anhang).
Folgende Konfigurationen wurden getestet und werden im Reviewgespräch am 08.01.2020 genauer erläutert und diskutiert:

- Testfall 1: Radius: 0.4 freq1: -3 freq2: 7 offset: 0.0
- Testfall 2: Radius: 0.5 freq1: 1 freq2: 1 offset: 0.5
- Testfall 3: Radius: 0.5 freq1: 3 freq2: 1 offset: 0.5
- Testfall 4: Radius: 1.25 freq1: 5 freq2: -1 offset: 0.5

- Testfall 5: Radius: 1.5 freq1: 20 freq2: -13 offset: 0.0
- Testfall 6: Radius: 0.9 freq1: -1 freq2: 1 offset: 0.1

Anhand der Ergebnisse lässt sich die Funktionalität der Canvaszeichnung von Flowers klar erkennen.

7 Performance-Testing

Um die oben erwähnten Testverfahren zu ergänzen haben wir die verschiedenen SDKs des Online-Dashboards *Firebase* integriert. Dabei handelt es sich um ein Google-Tool, welches uns Informationen zu folgenden Fragestellungen gibt:
→ **Performance:** Die Leistung auf unterschiedlichen Geräten wird übersichtlich dargestellt. Hierbei können zum einen vordefinierte und zum anderen eigene *Traces* genutzt werden, um Teilbereiche zu analysieren.
→ **Absturz-Tracking:** Die *Crashlytics-SDK* erfasst Abstürze der App, analysiert diese und kategorisiert sie nach Ausmaß des Fehlers, Verbreitung und weiteren Parametern.
→ **Systemtests auf verschiedenen Geräten:** auf bis zu 5 physischen und 10 virtuellen Geräten können verschiedene Testszenarien (auch selbst definierte Espresso-Tests) durchgeführt und ausgewertet werden. So erreichen wir eine breitere Testcoverage auf Systemtest-Ebene.

Damit bietet Firebase uns ein mächtiges Werkzeug mit einem übersichtlichen Dashboard für verschiedene globale Systemauswertungen und -analysen.

A Anhang

A.1 Testfälle der manuellen Tests