Automatisiertes Testing von Android Applikationen

30 Mai 2014 | **Software Engineering** | **Alexander Pacha**
Lesezeit: 5 Minutes

Motivation

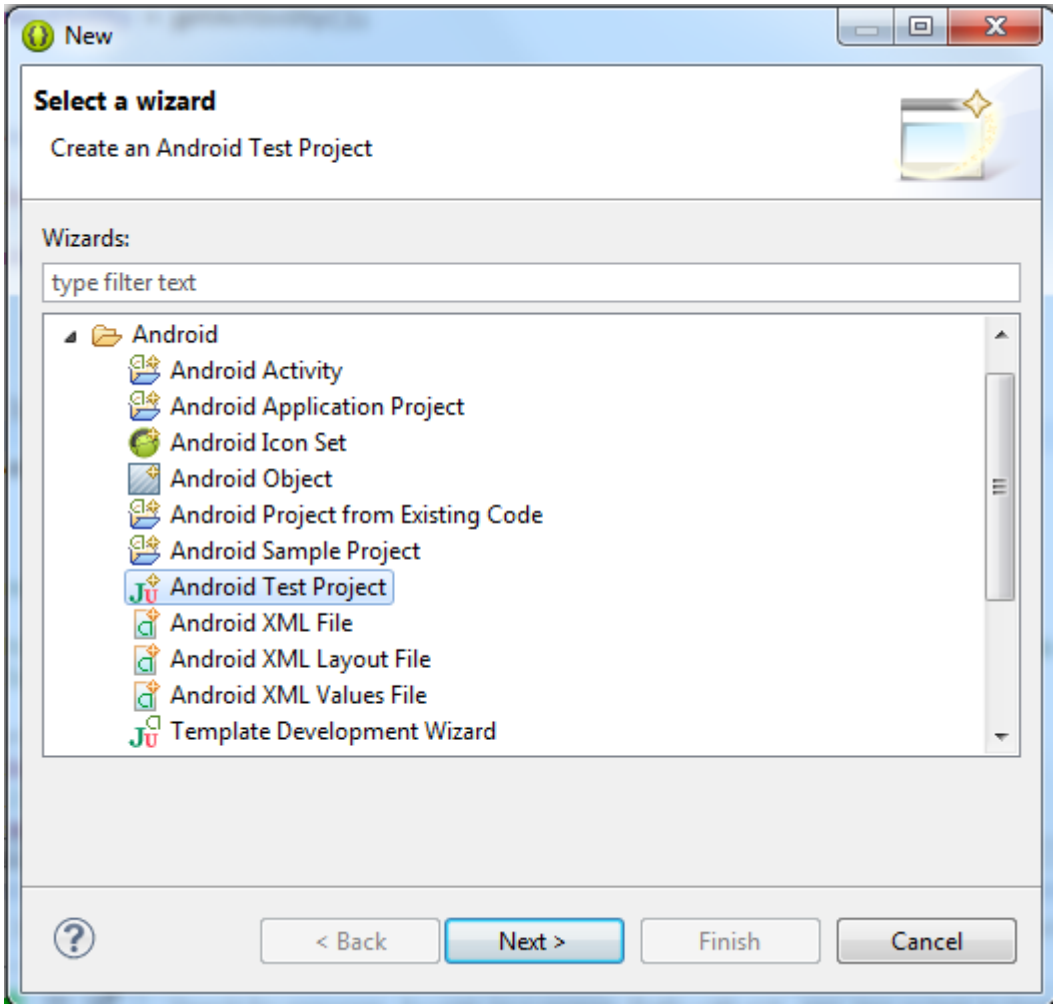
Testgetriebene Softwareentwicklung und automatisiertes Testen von Anwendungen helfen nicht nur bei der Entwicklung, sondern sichern auch die Qualität der Software. Doch wie mache ich das, wenn ich eine Android-Anwendung umsetzen möchte?

Die App auf ein Gerät spielen und manuell debuggen ist keine Option, denn Unit-Testing, Mocking und automatisiertes Testen gehören heute in den Werkzeugkoffer von jedem seriösen Softwareentwickler. Wie man einfache Java-Klassen mit JUnit testet ist hinreichend bekannt. Als Android-Entwickler bekommt man zusätzlich von Android bereits einige Tools zur Verfügung gestellt. Dennoch gibt es dabei ein paar Dinge zu beachten:

- Wer ein Android Testprojekt erstellt, muss noch mit JUnit3 arbeiten. Hübsche Annotationen wie @Test werden daher nicht unterstützt. Stattdessen muss man mit Namenskonventionen arbeiten.
- Das von Android mitgelieferte Mocking-Framework kann dazu genutzt werden, um Android-spezifische Klassen zu Mocken (z.B. Activity, Context, …), sowie um die Applikation zu instrumentieren.
- Mocking mit Mockito ist möglich, benötigt aber zwei zusätzliche Bibliotheken.
- Auf das User-Interface kann mittels Invocations zugegriffen werden, oder man nutzt z.B. das Robotium Framework um elegantes UI-Testing umzusetzen.

Erstellen eines Android Testprojekts

Das Erstellen eines Android Testprojekts ist relativ einfach und mit ein paar Klicks in Eclipse erledigt.



Eclipse Wizard zur Erstellung eines Android Testprojekts.

Mit folgendem einfachen Testfall kann zum Beispiel auf ein Element aus dem User-Interface durch Invocation zugegriffen, und der dort dargestellte Text überprüft werden:

```
public void testMainUI() {
    mActivity.runOnUiThread(new Runnable() {
        public void run() {
            mTextView.requestFocus();
            assertEquals("Hello world!",
mTextView.getText().toString());
        }
    }
    );
}
```

Instrumentierung mit dem Android Testing-Framework

Das Android Testing-Framework kümmert sich je nach Basisklasse (z.B. ActivityInstrumentationTestCase2) nicht nur um die Erzeugung von Activites, sondern man kann auch gezielt in den Lifecycle von Activites eingreifen. Dies funktioniert zum Beispiel so:

```
StartupActivity startupActivity = getActivity();
Instrumentation instrumentation = getInstrumentation();
// Simulate a start of the activity.
// Actually it's a restart here, because getActivity starts
// the Activity automatically if it wasn’t started before.
instrumentation.callActivityOnResume(startupActivity);
```

Besonders spannend ist jetzt folgender Use-Case: Meine StartupActivity wird kurz gestartet und je nachdem ob der User bereits angemeldet ist, wird entweder die LoginActivity gestartet, oder die MainActivity.

Folgendes (vereinfachte) Interface wird zur Authentifizierung verwendet.

```
public interface ICredentialChecker {
    public boolean isUserLoggedIn();
}
```

Die Klasse StartupActivity könnte nun in etwa so aussehen:

```
public class StartupActivity extends Activity {
    // Default initialisation – is missing here
    private ICredentialsChecker checker = getDefaultChecker();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    protected void onResume() {
        super.onResume();

        // Is the user logged in? If so, start MainActivity,
        // otherwise start LoginActivity
        if (checker.isUserLoggedIn()) {
            // Start MainActivity
            Intent intent = new Intent(this, MainActivity.class);
            startActivity(intent);
            finish();
        } else {
            // Start LoginActivity
            Intent intent = new Intent(this, LoginActivity.class);
            startActivity(intent);
            finish();
        }
    }

    /**
     * Dependency-inject the credentials checker
     */
    public void setCredentialsChecker(ICredentialsChecker checker) {
        this.checker = checker;
    }
}
```

Mit der Instrumentierung kann man den Programmablauf nun folgendermaßen testen:

```
// Register that we are interested in the MainActivity
// (after StartupActivity finished)
Instrumentation.ActivityMonitor monitor =
    instrumentation.addMonitor(MainActivity.class.getName(), null,
false);

// Simulate the start of the Startup Activity
instrumentation.callActivityOnResume(startupActivity);

// Wait for the MainActivity to start...
Activity mainActivity =
    getInstrumentation().waitForMonitor(monitor);

// Make sure the activity was started and exists
assertNotNull(mainActivity);
// Get any view-element of the new activity ...
View button = mainActivity.findViewById(R.id.slideshow_button);
//... and assert that it is not null
assertNotNull(button);
```

Das sieht doch schon recht brauchbar aus. Leider müssen wir noch sicherstellen, dass der User auch tatsächlich angemeldet ist. Dies löst man üblicherweise mittels Dependency Injection, indem man zum Beispiel folgenden Stub erzeugt und im Test einschleust.

```
public class CredentialsCheckerStub : ICredentialsChecker {
    public boolean isUserLoggedIn() {
        return true;
    }
}
```

Mocking mit Mockito

Da man nun aber nicht jedes Mal eine eigene Klasse erstellen möchte, oder Klassen mocken möchte, die kein Interface haben, gibt es Mocking-Frameworks, die das Verhalten von Klassen dynamisch simulieren können. Mockito zählt zu den bekanntesten Mocking-Frameworks für Java. Um Mockito für Android verwenden zu können, müssen noch zusätzlich die zwei Bibliotheken *dexmaker-1.0.jar* und *dexmaker-mockito-1.0.jar* eingebunden werden. Dies hat mit der Dalvik VM zu tun um elegant Proxy-Objekte erstellen zu können.

Einmal eingebunden könnte man den Test um folgenden Mock erweitern und erspart sich dadurch die Erstellung der Klasse CredentialsCheckerStub:

```
ICredentialsChecker checkerMock  =
    Mockito.mock(ICredentialsChecker.class);
Mockito.when(checkerMock.isUserLoggedIn()).thenReturn(true);
// Dependency Inject the mock into the activity
startupActivity.setCredentialsChecker(checkerMock);
// Simulate the start of the Startup Activity
instrumentation.callActivityOnResume(startupActivity);
// Do the rest of the Unit-Test ...
```

User-Interface Testing mit Robotium

Zu guter Letzt wünscht man sich neben einem dummen Affen, der zufällig auf das UI drückt auch noch die Möglichkeit um Benutzerinteraktionen (meist Berührungen des Displays) zu simulieren. Dafür bietet das Framework Robotium eine sehr elegante Lösung an:

```
public class PlayTest extends
ActivityInstrumentationTestCase2<MainActivity> {
    private MainActivity mActivity;
    private Solo solo;

    public void testEnableDebugging() throws Exception {
        // Open the menu
        solo.sendKey(Solo.MENU);
        // Click on settings
        solo.clickOnText("Einstellungen");
        // Activate debugging
        solo.clickOnText("Debuggen aktivieren");
        // Go back to main activity
        solo.goBack();
        // Assert that we are in main activity and debugging is active
        Assert.assertTrue(solo.searchText("Debugging ist aktiv"));
    }
}
```

Mit wenigen Codezeilen kann man Tastendrücke simulieren, Text suchen und vollständige User-Szenarios in Tests abbilden.

Wem das Schreiben dieser Tests noch immer zu aufwändig ist, der kann den Robotium Recorder ausprobieren, der nach einer kurzen Trial um etwa 300$ zu haben ist. Bei uns hat der Recorder leider nicht wie gewünscht funktioniert, aber wir haben es auch nicht besonders lange versucht, ihn zum Laufen zu bringen.

Ähnliche Tests kann man auch mit Selendroid oder Google Espresso erstellen.

Fazit

Android Testing ist mit den oben genannten Frameworks inzwischen wirklich brauchbar geworden. Was damit allerdings noch nicht geprüft wurde ist, ob das User-Interface für verschiedene Formfaktoren, Sprachen und Geräte sinnvoll aussieht und tatsächlich auch funktioniert. Hierfür gibt es Anbieter, die die Applikation auf einer Vielzahl von Geräten testen (online im Emulator und physisch auf Geräten) zum Beispiel TestObject oder Testdroid.

Wer noch eine Stufe weitergehen möchte und Behaviour-Driver-Development mit Feature-Spezifikationen machen möchte, der kann dies zum Beispiel mit Calabash tun. Mehr Infos zu Calabash und dem selbstgebauten Testing Rig bei Zühlke gibt es in dem Beitrag von Michael Sattler.