Automated Testing for Android

Stefan Piotrowski
Technische Universität
Berlin
Email: janstefanpiotrowski@gmail.com

I. Einleitung

Die Interaktion mit mobilen Softwareanwendungen, wie zum Beispiel Android-Applikationen, geschieht größtenteils über eine graphischen Benutzeroberfäche. Eine wichtige Schnittstelle zum Anwendungsnutzer ist also der Bildschirm, mit einer On-Screen-Tastatur oder teilweise auch mit einer Hardware-Tastatur für die Texteingabe. Weitere Interaktionsarten können über das Mikrofon, die Kamera oder das eingebaute Gyroskop umgesetzt werden. Will man eine mobile Anwendung entwickeln, die großen Erfolg und vor allem große Akzeptanz bei den Nutzer hat, so müssen diese Schnittstellen absolut korrekt funktionieren. Diese Qualitätsanforderung wird über Softwaretests gewährleistet. Durch die Automatisierung dieser Tests verringert sich die Zeitspanne eines Softwareprojekts erheblich. Aus diesem Grund wurden für die verschiedensten Programmiersprachen Mechanismen entwickelt, die diese Automatisierung unterstützen. Auch für Android gibt es mehrere Methoden der Testautomatisierung.
In dieser kurzen Arbeit wird zuerst auf zwei verschiedene *Modelle der Softwareentwicklung* eingegangen. In den weiteren Kapiteln wird in kurzer Form die *Android-Architektur* erklärt, sowie einzelne *Testarten für Androidanwendungen* vorgestellt. Nach einem Exkurs zu den *Testplanentwürfen*, werden noch *JUnit*, sowie zwei *Testframeworks* und die Automatisierungsmethode *Maven* vorgestellt. Zum Schluss gibt es noch eine abschließende *Diskussion* mit einem Ausblick, welche Techniken in der Zukunft angewandt werden könnten.

S. Piotrowski
Mai 20, 2011

II. Modelle der Softwareentwicklung

Das Testen von Software ist ein wichtiger Bestandteil der Qualitätssicherung in der Softwareentwicklung. Viel zu oft wird der Aufwand und der Nutzen einer gründlichen Überprüfung von Software unterschätzt. Bei größeren Softwareprojekten werden mittlerweile eigene Teams aus Testingenieuren eingesetzt, um den Testaufwand durch die Entwickler zu verringern und die Qualität der Tests zu erhöhen.
Am Anfang jedes Projektes steht ein Lastenheft. Aus ihm wird das Pflichtenheft und später eine Spezifikation abgeleitet. Diese Spezifikation ist die Grundlage jedes Softwaretests. Ein Verhalten eines Programmes, das nicht der Spezifikation entspricht kann als Fehler angesehen werden.
So sollte jedes Softwareprojekt beginnen. Für die weitere Vorgehensweise, kann man zwischen zwei verschiedenen Modellen der Softwareentwicklung unterscheiden. Hat man viel Zeit für ein Projekt und ist die Wichtigkeit der Korrektheit besonders hoch so wird oft das *traditionelle Modell* angewandt[1]. Beispiele sind hierfür das V-Modell oder das Wasserfall-Modell. Muss jedoch schnell auf Veränderungen im Markt reagiert werden können und stehen nur wenige Entwickler zur Verfügung, so wählt man eher ein *agiles Entwicklungsmodell* zur Umsetzung seines Softwareprojekts.

A. Traditionelles Modell

Im Laufe der Zeit wurden viele Testkonzepte entwickelt. Das wohl bekannteste ist das V-Modell (Figure 1). Die einzelnen Phasen dieses Modells sind wie ein V angeordnet. Auf der linken Seite stehen die Konzeption und die Implementierung einer Software. Rechts findet man in zeitlicher Abfolge die verschiedenen Testarten, die die Software immer allgemeiner Testen. Angefangen wird mit den *Komponententests*. Hier werden die einzelnen Module einer Software ohne Beachtung des Gesamtsystems getestet. In der nächsten Phase folgen die *Integrationstests*. Dabei wird die Zusammenarbeit unterschiedlicher voneinander abhängiger Module überprüft. Als nächstes folgen die *Systemtests*. In dieser Phase wird das System das letzte Mal von den Entwicklern als Ganzes getestet. Zuletzt folgen die *Abnahmetests*. Diese Tests werden oft manuell durchgeführt und dienen der Ermittlung der Bedienbarkeit und Akzeptanz.
Besondere Merkmale dieses traditionellen Modells ist der hohe Zeitbedarf, die große Iterationsschrittweite, der ebenso hohe Personalbedarf und der enorme Dokumentationsaufwand.

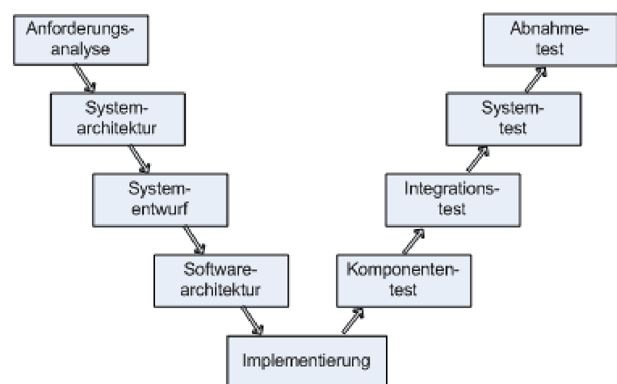

Fig. 1. V-Modell

Parameter	XP	Scrum	Crystal	FDD
Teamgröße	6-10	5-10	Clear: 3-6 Orange: 21-40	Max. 800 mit 3-5 pro Feature
Iterationszeit	1-4 Wochen	4 Wochen	Clear: 8-12 Wochen Orange: 12-16 Wochen	2 Wochen
Testlevel	je Komponente	je Sprint	je Unit	je Unit oder Iteration
Vorteile	Einfachheit	Teaminteraktion	variable Projektgröße	Geschäftsmodelle

TABLE I
VERGLEICH AGILER ENTWICKLUNGSMODELLE [3]

B. Agile Modelle

Im Vergleich zu den traditionellen Modellen sind die verschiedenen agilen Modelle besonders leichtfüßig. Die Iterationszeit, also die Zeit für die Entwicklung und den Test eines neuen Features, beträgt in der Regel nur zwischen einer und vier Wochen. Der Dokumentationsaufwand ist sehr gering und es besteht eine regelmäßige Kommunikation zwischen den Entwicklern, beziehungsweise dem Produktmanager, und dem Auftraggeber. So können kleine Änderungen schnell und unbürokratisch umgesetzt werden. Durch die Aufteilung der Software in Kern und zusätzliche Features, kann die Software früher auf den Markt gebracht werden. Neue Features werden dann schrittweise durch neue Releases hinzugefügt.[2]
Beispiele für agile Softwareentwicklungsmodelle sind unter anderem Scrum, Extreme Programming (XP), Feature-driven Development (FDD), und Crystal. In Tabelle 1 werden Unterschiede in Teamgröße, Iterationszeit, Testansatz und Vorteile dargestellt.

C. Testverfahren

In den einzelnen Phasen eines Testmodells können zwei verschiedene Arten von Testverfahren angewandt werden, statische Tests und dynamische Tests.
Bei statischen Tests wird der Programmcode auf syntaktische und logische Fehler überprüft. Die dynamischen Tests befassen sich dagegen nur mit dem Verhalten eines Testobjekts. Bei den dynamischen Tests unterscheiden wir noch zwischen dem Blackbox-Verfahren und dem Whitebox-Verfahren.
Unter dem Whitebox-Verfahren versteht man alle Tests, die von dem gleichen Programmierer entwickelt wurden, der auch das zu testende Objekt programmiert hat. Der Tester hat also Kenntnisse über die innere Funktionsweise des Testobjekts.
Beim Blackbox-Verfahren ist der innere Aufbau des Testobjekts dem Tester unbekannt. Der Tester kennt nur die Schnittstelle zum Objekt und die erwarteten Ergebnisse. Dieses Verfahren wird vor allem für Oberflächentests angewandt. Da Android-Applikationen zum Großteil aus GUI-Elementen besteht, ist dieses Verfahren bei Androidtests das am meisten genutzte.

D. Testphasen

Alle Tests bestehen aus drei grundlegenden Testphasen.
In der *Testplanung* werden die in der Spezifikation vorgeschriebenen Anforderungen analysiert. Aus dieser Analyse resultieren unterschiedliche Testfälle, die alle korrekten und alle fehlerhaften Eingabekombinationen abdecken.
Die *Testdurchführung* beinhaltet das manuelle oder vorzugsweise automatische Ausführen der Tests.
Die Ergebnisse der Testdurchführung werden in der *Testdokumentation* festgehalten.[4]

Bis auf die Abnahmetests werden die meisten Tests automatisch durchgeführt. Auch in der Softwareentwicklung für das Android-Betriebssystem von Google gibt es eine automatisierte Testdurchführung.
Android ist eine auf Java aufbauende Programiersprache.[5] Daher liegt es nahe auch die von Java her bekannte Test-Framework *JUnit* zu nutzen. Doch bevor wir uns jetzt mit Android-spezifischen Testverfahren auseinandersetzen, folgt zuerst eine eine kurze Erklärung der Android-Architektur.

III. DIE ANDROID-ARCHITEKTUR

Die Android-Plattform setzt sich, wie in Figure 2 dargestellt, aus vier Schichten zusammen. Ganz oben befindet sich die Anwendungsschicht. Unter ihr folgen das Anwendungsframework, die Bibliotheken und die Laufzeitumgebung, sowie der Linux-Kernel.
Die Android-Programme laufen in der obersten Schicht. Services, zum Beispiel der *Activity Manager* oder der *Content Provider* befinden sich in der zweiten Schicht. In den Bibliotheken werden alle statischen Bibliotheken und Oberflächenmanager zusammengefasst. Die Android-Laufzeitumgebung besteht aus Android-Laufzeitbibliotheken und der textitDalvik Virtual Machine (DVM). Die DVM kann mit einer Java-Virtual-Machine verglichen werden. Jede Android-Applikation wird in einer eigenen DVM gestartet. So wird sichergestellt, dass abstürzende Programme nicht das gesamte System beeinflussen können.
Die oben genannten Schichten setzen alle auf dem Linux-Kernel auf, der die Hardware verwaltet und benötigte Ressourcen verteilt. [5]
Android-Softwaretests und vor allem Android-GUI-Tests arbeiten überwiegend mit den drei oberen Schichten. Dies liegt wohl daran, dass mit mobilen Anwendungen nur sehr selten komplexe Geschäftslogiken umgesetzt werden. Diese sind meist auf Servern ausgelagert, da sie eine höhere Prozessorlast erzeugen und somit einen größeren Energiebedarf haben.

Jede Android-Applikation besteht aus vier Komponenten. Die *Activity* stellt alle Oberflächenelemente dar, liest Benutzereingaben aus und bearbeitet sie gegebenenfalls weiter. Eine Activity ist ein obligatorischer Bestandteil aller Android-Anwendungen. Jeder Benutzeroberfläche wird

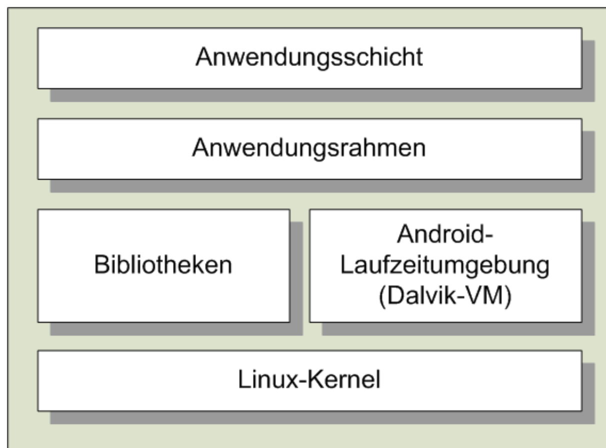

Fig. 2. Architektur der Android-Plattform

durch eine eigene *Activity* erzeugt. So besteht eine Android-Anwendung aus einer Vielzahl miteinander verknüpfter Activities.
Die zweite Komponente ist der *Service*. Auch diese Komponente kann Benutzereingaben auswerten. Nach dem Schließen einer Anwendungsoberfläche kann sie jedoch als Hintergrundprozess weiterarbeiten. Anwendung finden Services zum Beispiel bei Musikabspielprogrammen oder Emailbenachrichtigungsdiensten.
Der *Content Provider* stellt die Schnittstelle zwischen der Applikation und persistenten Daten dar. Sollen zum Beispiel Spielstände gespeichert oder Einstellungen abgerufen werden, kommt diese Komponente zum Einsatz.
Als vierte Komponente bleibt noch der *Broadcast Receiver*. Er ist die Schnittstelle zum restlichen Gerät. Wird zum Beispiel der Akku schwach oder hat das mobile Endgerät eine benötigte Netzwerkverbindung verloren, empfängt der Broadcast Receiver die Systemnachricht und bearbeitet sie entsprechend.

Der Hauptteil aller Android-Applikationen besitzt eine eigene grafische Benutzeroberfläche (GUI) und wird mittels dieser auch bedient. Aus diesem Grund befassen sich die meisten Tests von mobilen Anwendungen auch mit der Überprüfung dieses Programmteils. Weitere wichtige zu testende Anwendungsteile sind die Kommunikation zwischen den einzelnen Android-Komponenten und die Erzeugung von Systemnachrichten.

IV. Testarten im Detail

Im Kapitel *Testkonzepte* wurden verschiedene Phasen von Tests in der Softwareentwicklung vorgestellt. In diesem Kapitel wird nun näher darauf eingegangen und ein Bezug zum Android-Betriebssystem hergestellt.
Als wichtigste Testarten kann man bei der Überprüfung von Android-Software folgende unterscheiden:

- Test der Gesamtanwendung
  Der Test der Gesamtanwendung wird im Allgemeinen auch als Systemtest bezeichnet. Er wird meist in einer späten Entwicklungsphase durchgeführt. Zu diesem Zeitpunkt liegt ein Großteil der Android-Module bereits im getesteten Zustand vor.
  Die Durchführung geschieht in der Regel manuell. Diese Art des Tests wird aber immer mehr automatisiert. Zum Test der Gesamtanwendung gehören auch sogenannte Last- und Stresstests. Darunter versteht man die explizite Beanspruchung und Überbeanspruchung einer Software. Beispiele sind hier zum Beispiel das Anlegen von besonders vielen Speicherständen in einer Android-Anwendung oder die Übertragung von großen Datenmengen über das Netzwerk. Last- und Stresstests werden vorzugsweise automatisch durchgeführt. Die Android-SDK bietet mit *Monkey* ein nützliches Tool an, das die automatische Ausführung von Stresstests auf der Androidoberfläche vereinfacht. Monkey simuliert randomisierte Benutzereingaben, wie zum Beispiel Klicks, Gesten oder Berührungen. Außerdem können mit Monkey Systemnachrichten erzeugt werden. [6]
  In Figure 3 ist ein Testprozess dargestellt, das Monkey nutzt. Das Testfallerstellungstool *Monkey* erzeugt Testfälle für alle Events, die durch Activities hervorgerufen werden können. Dies geschieht in der JUnit-Ebene. Die Testausführung und die Testdokumentierung (Log files) ist äquivalent zu den Testphasen der Vorgangs- und Modultests.

- Test eines Vorgangs
  Beim Test eines Vorgangs werden einzelne Prozesse überprüft. Unter diesen Prozessen versteht man hier zum Beispiel die Anmeldung in einer Android-Anwendung, die Änderung von Benutzereinstellungen innerhalb der Anwendung oder die einzelnen Schritte bei der Routensuche in der Android-Version von GoogleMaps ®. Umgesetzt wird diese Testart häufig durch *Oberflächentests*.
  Für die Umsetzung solcher Oberflächentests programmiert der Testentwickler eine eigene Klasse, die Eingabe auf der Benutzeroberfläche simuliert. Diese Klasse überprüft auch die Reaktion der Anwendung und dokumentiert die Ergebnisse in Log-Dateien, die bei der Testauswertung analysiert werden.[5]
  Eine weiteres Anwendungsgebiet der Oberflächentests sind die bereits oben vorgestellten Tests der Gesamtanwendung. Diese werden in diesem speziellen Fall *smoke tests* genannt. Dabei werden ausgewählte Vorgangstests durchgeführt. Diese sollen aus allen Bereichen der Anwendung stammen und eine grobe Funktionalität der Software gewährleisten. Smoke tests werden in der Softwareentwicklung nach jedem neuen Branch (Änderung der Softwareversion) durchgeführt. Sie stellen sicher, dass eine Anwendung nach größeren Änderungen im Grundkonzept noch funktioniert. Ein vollständiges Testen nach Änderungen des Softwarecodes wäre an dieser Stelle viel zu aufwendig.

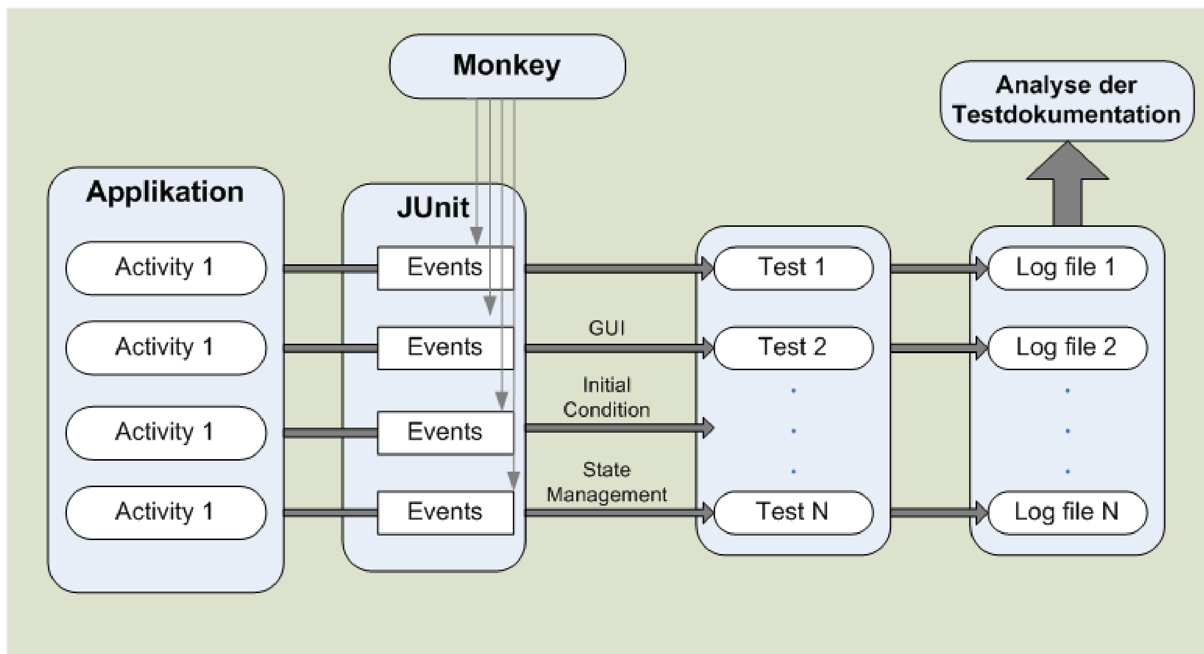

Fig. 3. Testprozess mit Testfallerstellung durch Monkey [7]

Das Verfahren, das bei den Oberflächentests eingesetzt wird, ist das Blackbox-Verfahren.
Wie bei allen Tests werden hier bestimmte Gesichtspunkte beachtet:

- Läuft ein Vorgang bei bestimmten Eingaben korrekt ab?
- Werden Fehleingaben erkannt und auch entsprechend behandelt?

Die Erstellung von Testfällen für Oberflächentests kann manuell oder mit verschiedenen verfügbaren Tools durchgeführt werden. Eines dieser Tools ist *robotium*. Dieses Test-Framework ist standardmäßig nicht in der Android-SDK vorhanden. Die Bibliothek kann jedoch kostenfrei im Internet bezogen werden. Robotium verbessert die Erstellbarkeit, die Lesbarkeit und die vermindert die Ausführungszeit von Oberflächentests. Um die Softwareentwicklung noch weiter zu vereinfachen ist das Tool noch in Maven und Ant integriert. [8]
Um möglichst nahe am System zu testen, nutzen Oberflächentests das selbe Dateisystem und die selben Datenbanken wie die zu testende Anwendung. Damit sich nacheinander ausgeführte Tests nicht gegenseitig beeinflussen können, wird nach jeder Testdurchführung der alte Zustand des Systems wiederhergestellt, der vor dem Test vorlag. Um dieses *Rollback* möglichst zu vermeiden, versucht man Schreiboperationen in Oberflächentests zu vermeiden.[5]

- Modultest
  Modultests entsprechen den im Kapitel Testkonzepte angesprochenen Komponententests. Diese Art der Überprüfung von Software befasst sich mit den einzelnen Klassen der Anwendung. Bei den Modultests kann man dabei zwischen den androidfreien Klassen und den androidabhängigen Klassen unterscheiden. Die androidfreien Klassen, also Klassen, die nicht die Android-SDK nutzen, werden, wie gewöhnlicherweise mit allen Java-Klassen verfahren wird, mit JUnit getestet. JUnit-Tests werden hier nicht weiter behandelt.
  Der für uns interessantere Teil der androidabhängigen Klassen wird mit androidspezifischen Methoden getestet. Die Tests finden isoliert vom System, also den anderen Komponenten statt. Die Testklasse muss daher die Schnittstelle der zu testenden Komponente zum System simulieren. Dabei werden nur die benötigten Ressourcen bereitgestellt. Eine Simulation des Gesamtsystems ist nicht erforderlich. Aus diesem Grund muss für einen Modultest auch keine Android-Laufzeitumgebung (DVM) gestartet werden.
  Wie bei den Oberflächentests werden gültige und ungültige Eingaben überprüft und die erwartete Reaktion der Komponente mit der beobachteten Reaktion verglichen.
  Für fast alle Komponententypen, die Android bereitstellt, sind bereits vorgefertigte Testfall-Basisklassen vorhanden. Hier sind einige von ihnen beispielhaft dargestellt:

  - ActivityUnitTestCase
    Basisklasse für den Test von Activity-Klassen. Da jede Android-Anwendung aus mindestens einer Activity-Klasse bestehen muss, wird bei der Testplanerstellung auch eine oder mehrere Testfälle

mit dieser Basisklasse erstellt.

– ProviderTestCase2
Basisklasse für den Test des Content Providers. Diese Basisklasse simuliert den Zugriff auf den Content Provider und somit auf die Daten des Dateisystems. Die vorhandenen daten des Dateisystems werden dabei nicht verändert.

– ServiceTestCase
Basisklasse für den Test von Services. Hiermit kann man Services in seinem gesamten Lebenszyklus in einer isolierten Umgebung testen.

– AndroidTestCase
Basisklasse für den Test von Zugriffen auf Ressourcen oder den Anwendungskontext. Der Anwendungskontext lässt sich hierbei um weitere Einschränkungen erweitern.

Alle drei Testarten, also Tests des Gesamtsystems, Tests von Vorgängen und die Modultests, sind obligatorisch, um eine möglichst große Testabdeckung einer Android-Anwendung zu erreichen. Ein vollständiger Test einer Anwendung ist aufgrund viel zu großer Kombinationsmöglichkeiten der Testfälle nicht möglich. [5]

V. Automatisierung des Entwicklungsprozesses durch Maven

Maven ist Build-Management-Werkzeug, das alle Phasen eines Softwarelebenszykluses automatisieren kann. Es steht für verschiedene Programmiersprachen zur Verfügung. Auch Android-Softwareprojekte können mit Maven automatisiert verarbeitet werden.[9]
Im der folgenden Auflistung sind einige Phasen, die durch Maven verarbeitet werden können, dargestellt. Dabei müssen nicht alle Prozesse automatisiert werden.

- *Validierung* der Projektstruktur
- *Kompilierung* des Softwarecodes
- *Test* gegen die Spezifikation
- *Verpacken* in ein Softwarepacket (z.B. xyz.jar bei Java)
- *Integrationstest* des Softwarepackets in einem System
- *Verifizierung* der Struktur des Pakets
- *Installation* auf einem oder verschiedenen Systemen

VI. Testplanentwurf

Wie wird entschieden was in einem Softwaretest überprüft werden muss? Das hängt immer ganz von der Anwendung und dem Umfeld in der sie genutzt wird ab. Eine Anwendung, die einem Kardiologen Daten zum Zustand eines Patienten bereitstellt, hat sicherlich viel höhere Anforderungen in der Korrektheit ihrer Funktionsweise als eine Android-Applikation, die lediglich die Wetterdaten zur Verfügung stellt.
Wie oft wird ein Vorgang von einem Anwender ausgeführt und wie gefährlich wäre eine fehlerhafte Reaktion der Anwendung? Selten genutzte und weniger wichtige Funktionalitäten müssten zum Beispiel in *smoke tests* nicht mit aufgeführt werden.
Abgesehen von diesen Gesichtspunkten, muss man sich bei der Testplanerstellung auch die Frage stellen auf welchen Systemen die Applikation überhaupt laufen soll. Das Android-Betriebssystem kann mittlerweile nicht nur noch auf ARM-Prozessoren aufgesetzt werden. Auch die X86-Architektur und viele weitere Systeme werden schon unterstützt. So wird es in der Zukunft Android-Anwendungen in Fernsehern, in Kraftfahrzeugen und auch Netbooks geben, also unterschiedlichste Anwendungsgebiete mit verschiedensten Anforderungen.

VII. JUnit - Das Java-Testframework

Wie bereits in Kapitel IV bei den Testarten beschrieben, kann man bei Android-Applikationen zwischen Android-abhängigen und Android-freien Klassen unterschieden werden. Die Android-abhängigen Klassen werden durch spezielle Android-Testframeworks getestet. Diese werden, durch zwei Beispiele, im nächsten Kapitel vorgestellt. Dieses Kapitel behandelt den Test von Android-freien Klassen mittels JUnit. JUnit ist ein Java-Testframework, welches auch bei reinen Java-Applikationen zum Testen verwendet wird. Mit ihm lassen sich einzelne Module einer Anwendung isoliert von der Gesamtsoftware in ihrer Korrektheit überprüfen. Anwendung findet das Framework auch bei der *Testgetriebenen Softwareentwicklung*. Hier werden zuerst die Modultests erstellt und im Nachhinein der Programmcode entwickelt, der die Tests bestehen muss. Besteht ein Programmteil eine Anforderung eines Tests, so bekommt sie einen *grünen* Status. Fällt es jedoch durch, so ist der Status *rot*. Dies kann sowohl durch durch Fehler im Code oder auch durch falsch Ergebnisse hervorgerufen werden. Jeder Fehler eines JUnittests ruft eine Exception hervor und kann somit auch dokumentiert werden. Das JUnit-Framework ist die Grundlage vieler Android-Testframeworks, welche im folgenden Kapitel beschrieben werden.[10]

VIII. Android-Testframeworks

Im Kapitel IV wurde kurz das Testframework *robotium* angesprochen. Es dient der vereinfachten Erstellung von Oberflächentests. Nun werden zwei weitere Frameworks für die Implementierung von GUI-Tests vorgestellt. Beide nutzen unterschiedliche Ansätze.

- Das erste Framework ist das in der Android-SDK integrierte *Android Instrumentation Framework* (AIF). Es stellt unterstützende Testklassen bereit, mit denen die zu testende Anwendung in einem Testmodus gestartet, bedient und wieder beendet werden kann. Das AIF nutzt das JUnit-Framework als Grundlage. So ist es für JUnit-Entwickler einfach zu

handhaben. Figure 4 zeigt den Aufbau des Android-Testframeworks. Auf der rechten Seite sieht man den *MonkeyRunner*, der optional für Stresstests genutzt werden kann. Das AIF und das später vorgestellte Positron Framework bauen auf dieser Grundlage auf. Die Klasse *ActivityInstrumentationTestCase2* erweitert die JUnit-Klasse TestCase und ermöglicht somit die Verifikation von erwartetem und vorhandenem Verhalten der Anwendungsoberfläche. Das AIF agiert dabei als Anwender, indem es Oberflächeneingaben simuliert. Da, wie bereits angesprochen, jede Activity eine eigene Oberfläche erstellt und jeweils nur eine Oberfläche auf dem Bildschirm dargestellt werden kann, kann man jede Oberfläche auch separat testen.
Das *Android Instrumentation Framework* stellt eine Low-Level-API zur Verfügung. Dies ermöglicht zum Beispiel die Simulation einfacher Tastatureingaben, wie in Listing 1 zu sehen ist. Dies ist sehr nah am realen Benutzerverhalten, erzeugt jedoch einen höheren Aufwand bei der Testerstellung.

Listing 1. Eingabe von "Name" in ein Textfeld inklusive Bestätigung mit dem AIF [11]

```
public void insertName() {

    sendKeys( KeyEvent.KEYCODE_N);
    sendKeys( KeyEvent.KEYCODE_A);
    sendKeys( KeyEvent.KEYCODE_M);
    sendKeys( KeyEvent.KEYCODE_E);

    Runnable saveRun = new Runnable();
        public void run() {
        save.performClick();
        ...
    }
}
```

Um die Testergebnisse zu dokumentieren nutzt AIF *JUnit assertions*. Für JUnit-Entwickler ist dies wieder kein neuer Ansatz der Software-Verfikation.Ein kleiner Nachteil dieser *assertion* ist jedoch, dass nur die negativen Ergebnisse erfasst werden. Für die Testanalyse ist es somit nur bedingt möglich zu erfassen welche Bedingungen bereits getestet wurden. [11]

- Das zweite Framework, das hier vorgestellt werden soll, ist das *Positron Framework*. Es ist eine Erweiterung des *Android Instrumentation Framework*, arbeitet jedoch nach einem ganz anderen Konzept.
  Dieses Konzenpt wird als Client-Server-Modell bezeichnet. Jeder einzelne Testfall ist ein eigener Client, der mit einer Activity, die als Server fungiert, zusammenarbeitet. Erstellt werden die Activities von den Clients selbst. [12]
  Wie bei AIF erweitern die Positron-Testklassen die JUnit-Klasse *TestCase*. Der Aufbau der Klassen ist genauso wie bei den JUnit-Testklassen. Auch die Verifikation des Verhaltens und der Daten mittels *asserts* ist aus JUnit bekannt.
  Eine große Verbesserung und auch ein großer Unterschied zum *Android Instrumentation Framework* ist die Bereitstellung von High-Level-Methoden. Dies ermöglicht zum Beispiel das Beschreiben von Textfelder mit nur einer Zeile Code, wie in Listing 2 dargestellt [11].

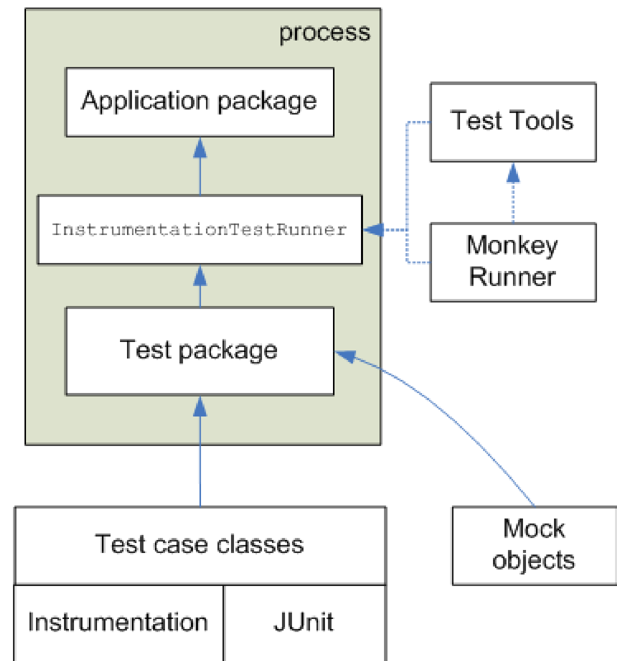

Fig. 4. Aufbau des Android-Testframeworks

Listing 2. Eingabe von "Name" in ein Textfeld inklusive Bestätigung mit Positron [11]

```
public void insertName() {
    ...
    press("Name", DOWN);
    click();
}
```

Unterschiede beider Frameworks sind vor allem die Bereitstellung von Low-Level- bzw. High-Level-APIs dar. Das *Positron Framework* hat bei der Erstellung von Testklassen seine großen Vorteile. der Code ist übersichtlicher und schneller zu erstellen. Mit dem *Android Instrumentation Framework* kann man dagegen viel tiefergehende Tests umsetzen, die einen direkteren Zugang zu den benötigten Ressourcen haben. Außerdem hat das AIF eine klaren Vorteil bei der Ausführung der Testklassen. Das *Positron Framework* muss bei jedem Start eines Tests eine Activity erstellen und den Kontakt herstellen [11]. Das kostet bei der Ausführung vieler Tests, zum Beispiel beim regelmäßigen Start von *smoke tests* oder Regressionstests, viel Zeit, die den Verfikateuren in der Regel nicht gegeben ist.
Die Vorteile beider Frameworks sind der Zugriff auf die Ressourcen über die Activity, die Ausführung der Tests auf der

Zielplattform, sowie die Grundlage von JUnit, was die Einarbeitung von Entwicklern mit JUnit-Kenntnissen erleichtert [11].

IX. Diskussion

Was haben wir gelernt?
Android baut auf der Programmiersprache Java auf und nutzt auch dessen vorgegebenes Testframework als Grundlage für eigene Testframeworks. Weiterhin konzentrieren sich Tests für Androidanwendungen überwiegend auf den Test der Benutzeroberfläche. Durch die große Community und den Open-Source-Ansatz von Android, gibt es schon viele Lösungen zu verschiedenen Testproblemen. Die Entwicklung von automatischen Testframeworks wird aber nicht stehenbleiben.
In der Zukunft wird es immer mehr mobile und auch stationäre Geräte geben, die das Android-Betriebssystem unterstützen. Durch die wachsende Nutzerzahl, erhöht sich auch die Notwendigkeit die Qualität der Software zu verbessern. Die Oberflächentests tragen einen hohen Teil zur Qualitätssicherung bei. Was bisher aber noch nicht betrachtet wurde ist die Benutzbarkeit der mobilen Anwendungen. Was nutzt eine korrekt funktionierende Anwendung, wenn zum Beispiel die Buttons oder die Schrift viel zu klein sind? Auch die Nutzung mittels Sprachbefehlen, wie sie Menschen mit körperlichen Einschränkungen verwenden könnten, wird mit aktuellen Testverfahren nicht abgedeckt. Es ist also noch viel zu tun bei der Weiterentwicklung von automatisierten Tests für Androidanwendungen.

References

[1] R. R. V. Rahimiam, “Designing an agile methodology for mobile software development: A hybrid method engineering approach,” *Research Challenges in Information Science*, pp. 337–342, 2008.

[2] C. T.-e. S. Zhong, C. Liping, “Agile planning and development methods,” *Computer Research and Development*, pp. 488–491, 2011.

[3] M. A. F. F. Kanwal, K. Junaid, “A hybrid software architecture evaluation method for fdd - an agile process model,” *Computational Intelligence and Software Engineering*, pp. 1–5, 2010.

[4] T. L. A. Spillner, *Basiswissen Softwaretest*, dpunkt.verlag, Ed. dpunkt.verlag, 2004.

[5] M. P. A. Becker, *Android 2 - Grundlagen und Programmierung*. dpunkt.verlag, 2010.

[6] Ui/application exerciser monkey. [Online]. Available: http://developer.android.com/guide/developing/tools/monkey.html

[7] I. N. C. Hu, “Automating gui testing for android applications,” *AST*, vol. 5, pp. 10–17, 2011.

[8] User scenario testing for android. [Online]. Available: http://code.google.com/p/robotium/

[9] Developers. Apache maven project. [Online]. Available: http://www.apache.maven.org

[10] ——. Junit.org resources for test driven development. [Online]. Available: http://www.junit.org

[11] P. M. M. Kropp, “Automated gui testing on the android platform,” *IFIP ICTSS*, vol. 11, pp. 64–72, 2010.

[12] Autoandroid. Positron framework. [Online]. Available: http://code.google.com/p/autoandroid/wiki/Positron