

## **Masterarbeit**

### **Automatisiertes Testen von mobiler Software durch Simulation wechselnder Umgebungsänderungen im Android-Kontext**



von  
Manuel Schmidt

im Studiengang:  
Master of Computer Science

Erstprüfer: Prof. Dr. Peter Becker  
Zweitprüfer: Prof. Dr. Manfred Kaul  
Firmenbetreuer: Dominik Helleberg

Eingereicht am: 13. August 2013

# Danksagung

Als erstes möchte ich mich bei der Firma inovex GmbH für die produktive und überaus angenehme Zusammenarbeit bedanken. Ich habe die Zeit sehr genossen, so dass die Arbeit ein reines Vergnügen war. Besonders herzlich möchte ich mich bei Dominik Helleberg bedanken, der nicht nur meine Arbeit vorbildlich betreut hat, sondern mir auch mit sehr viel Fachwissen zur Verfügung stand.

Auch bei Prof. Peter Becker möchte ich mich für die konstruktive und kompetente Betreuung bedanken. Die Zusammenarbeit war durchweg unkompliziert und zielführend.

Mein besonderer Dank gilt meiner Freundin. Sie hat mein Leben sehr bereichert und mir sehr viel Kraft und Selbstvertrauen gegeben. Sie hat mich auch während meiner Masterarbeit immer liebevoll unterstützt und in schwierigen Situationen immer wieder aufgebaut. Bedanken möchte ich mich bei meiner Freundin auch für den außerordentlichen Einsatz beim Korrekturlesen. Ich habe es ihr wirklich nicht leicht gemacht, jedoch konnte sie durch ihre Kompetenz und ihre gewissenhafte Arbeitsweise den Schreibstil meiner Masterarbeit deutlich verbessern.

Natürlich gilt mein Dank auch meinen beiden fachlichen Korrekturlesern, Herrn Helleberg und Herrn Becker.

*Manuel*

# Zusammenfassung

Bisher konnten Android-Entwickler ihre Software nicht automatisiert testen, wenn die Software während der Laufzeit auf wechselnde Umgebungsänderungen reagieren musste. Einige Beispiele solcher Umgebungsänderungen sind Batteriestand, Lokalisierung, unterschiedliche Netzwerkverbindungen, etc. Mithilfe dieser Arbeit wird in dieser Hinsicht Abhilfe geschaffen, indem ein neuer Lösungsansatz entwickelt wird, der wechselnde Umgebungsänderungen simulieren kann. Dieser neue Ansatz ermöglicht somit das automatisierte Testen von Software, die auf mobile Parameter reagiert. Solche Tests konnten bisher nur manuell durchgeführt werden.

Dafür wird zunächst anhand einer Analyse aufgezeigt, dass die derzeit auf dem Markt verfügbaren Lösungen nicht die erforderliche Funktionalität bieten, um solche automatisierten Tests schreiben zu können. Eine anschließende Identifikation von möglichen Lösungsansätzen listet verschiedene technische Möglichkeiten auf, um wechselnde Umgebungsänderungen zu simulieren. Der beste Ansatz dient als Grundlage für die Entwicklung und Implementierung eines neuen Testframeworks namens Zomby. Zusätzlich werden die technischen Probleme erläutert, die dafür gelöst werden müssen. Eine anschließende Evaluierung zeigt, wie gut die neue Lösung die vorgegebene Problemstellung bewältigt sowie welchen Einschränkungen sie momentan noch unterliegt und wie diese in der Zukunft behoben werden können.

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>vi</b>
<b>Abbildungsverzeichnis</b>	<b>vii</b>
<b>Listingverzeichnis</b>	<b>ix</b>
<b>Tabellenverzeichnis</b>	<b>x</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Gegenstand und Ziel der Master-Thesis . . . . .	2
1.2 Verlauf und Aufbau der Master-Thesis . . . . .	2
1.3 inovex GmbH . . . . .	3
<b>2 Grundlagen</b>	<b>4</b>
2.1 Dynamische Tests . . . . .	4
2.1.1 Black-Box-Test . . . . .	4
2.1.2 White-Box-Test . . . . .	4
2.1.3 Grey-Box-Test . . . . .	5
2.2 Testarten . . . . .	5
2.2.1 Stresstest . . . . .	5
2.2.2 UI-Test . . . . .	5
2.3 JUnit . . . . .	6
2.4 Android SDK . . . . .	6
2.4.1 Emulator . . . . .	7
2.4.2 Virtual Device Manager . . . . .	8
2.4.3 Debug Bridge . . . . .	9
2.5 Android Activity Lifecycle . . . . .	9
2.6 LogCat . . . . .	11
2.7 Telnet . . . . .	11

<b>3</b>	<b>Analyse bisheriger Lösungen</b>	<b>13</b>
3.1	Kriterien der Analyse . . . . .	13
3.1.1	Netzwerkverbindung . . . . .	13
3.1.2	Batterie . . . . .	14
3.1.3	Lokalisierung . . . . .	14
3.1.4	Sensoren . . . . .	15
3.2	Auflistung bestehender Lösungen . . . . .	16
3.2.1	Instrumentation Framework . . . . .	16
3.2.2	Positron . . . . .	19
3.2.3	UI/Application Exerciser Monkey . . . . .	20
3.2.4	monkeyrunner . . . . .	20
3.2.5	Robotium . . . . .	23
3.2.6	Robolectric . . . . .	25
3.2.7	UiAutomator . . . . .	27
3.2.8	SensorSimulator . . . . .	29
3.3	Ergebnis der Analyse . . . . .	31
<b>4</b>	<b>Identifikation möglicher neuer Lösungsansätze</b>	<b>33</b>
4.1	Möglichkeiten der Simulation wechselnder Umgebungsänderungen . . . . .	33
4.2	Lösungsansätze . . . . .	34
4.2.1	Android manipulieren . . . . .	34
4.2.2	Geräte manipulieren . . . . .	35
4.2.3	Emulator manipulieren . . . . .	36
4.3	Ermittlung des besten Lösungsansatzes . . . . .	38
<b>5</b>	<b>Implementierung der neuen Lösung</b>	<b>40</b>
5.1	Problemanalyse . . . . .	40
5.1.1	Synchronisierung . . . . .	41
5.1.2	Manipulation . . . . .	41
5.1.3	Kommunikation . . . . .	43
5.2	Lösungsweg . . . . .	44
5.2.1	Prozesssynchronisierung . . . . .	44
5.2.2	Kommunikation durch Intents . . . . .	46
5.2.3	Manipulationskomponente . . . . .	46
5.3	Entwicklung eines neuen Frameworks . . . . .	49
5.3.1	Architektur . . . . .	51
5.3.2	Implementierung des Webdienstes . . . . .	52

5.3.3	Implementierung der Programmierschnittstelle . . . . .	54
5.3.4	Fehler- und Ausnahmebehandlung . . . . .	60
5.4	Implementierung einer Rekorder-App . . . . .	60
<b>6</b>	<b>Evaluierung des Zomby-Frameworks</b>	<b>65</b>
6.1	Relevante Anwendungsfälle jetzt automatisiert testbar . . . . .	65
6.1.1	Batteriezyklus . . . . .	65
6.1.2	Schwankende Datenverbindung mit plötzlichem Funkloch . . . . .	66
6.1.3	Fußgänger . . . . .	67
6.1.4	Zugfahrt / Autofahrt . . . . .	68
6.1.5	Sensoren . . . . .	69
6.2	Technische Grenzen . . . . .	70
6.2.1	Emulator unterstützt kein echtes WLAN . . . . .	70
6.2.2	Emulator hat keinen Telnetbefehl für den Flugmodus . . . . .	70
6.3	Fehlerbericht - Bugs im Android Emulator . . . . .	72
6.3.1	Batterieanzeige . . . . .	72
6.3.2	GSM-Zustandwechsel . . . . .	73
6.3.3	Falscher Netzwerkstatus . . . . .	73
6.4	Zusammenfassung . . . . .	73
<b>7</b>	<b>Testen mit Zomby</b>	<b>75</b>
7.1	Einrichtung der Entwicklungsumgebung . . . . .	75
7.1.1	Einrichtung des Webserver . . . . .	75
7.1.2	Einbindung der Framework-Bibliothek . . . . .	76
7.2	Robotium mit Zomby . . . . .	76
7.3	UiAutomator mit Zomby . . . . .	79
7.4	Zomby Recorder . . . . .	81
7.5	Die Tücken von Zomby . . . . .	82
7.5.1	Umgang mit Fehlern . . . . .	82
7.5.2	Vorsicht vor Flüchtigkeitsfehlern . . . . .	82
7.5.3	Die Grenzen der Echtzeit . . . . .	82
7.5.4	Drehen an der Echtzeitschraube . . . . .	83
<b>8</b>	<b>Fazit und Ausblick</b>	<b>85</b>
8.1	Fazit . . . . .	85
8.2	Ausblick . . . . .	86
	<b>Literaturverzeichnis</b>	<b>88</b>



# Abbildungsverzeichnis

2.1	<i>JUnit: Darstellung der Testergebnisse</i>	6
2.2	<i>Android Emulator: Virtualisierung des Nexus One</i>	7
2.3	<i>Konfigurationsfenster des AVD Managers</i>	9
2.4	<i>Aktivitätsdiagramm vom Android Activity Lifecycle</i>	10
2.5	<i>LogCat-Ausgabe über Eclipse</i>	11
3.1	<i>Zusammenfassung des Android Test Frameworks</i>	17
3.2	<i>Android Instrumentation Framework Klassendiagramm</i>	18
3.3	<i>Konsolenaufruf des UI/Application Exerciser Monkeys</i>	21
3.4	<i>Grafische Bedienoberfläche des SensorSimulators</i>	30
4.1	<i>Telnetschnittstelle des Android Emulators</i>	36
4.2	<i>Möglicher Ansatz ergänzt bestehende Testframeworks</i>	39
5.1	<i>Erweiterung des Testprozessablaufs durch zusätzliche Manipulationsanweisungen</i>	41
5.2	<i>Schemenhafter Aufbau der Testumgebung</i>	42
5.3	<i>Telnetverbindungsversuch zum Emulator innerhalb der Dalvik VM (LogCat-Ausschnitt)</i>	42
5.4	<i>Aktivitätsdiagramm des erweiterten Testprozesses</i>	43
5.5	<i>Prozessablauf: LogCat auslesen</i>	47
5.6	<i>Zugriff mit Android-Browser auf lokalen Webserver</i>	48
5.7	<i>Netzwerkadressraum des Android Emulators</i>	49
5.8	<i>Aktivitätsdiagramm des Zomby-Frameworks</i>	50
5.9	<i>ZombyException: JUnit-Test schlägt wegen falscher Telneteingabe fehl</i>	60
5.10	<i>Ausschnitt des Startbildschirms der Zomby Recorder App</i>	61
6.1	<i>Fehlerhafter Batteriestatus des Android Emulators</i>	72
7.1	<i>Einbinden des Zomby-Frameworks mit Eclipse</i>	77
7.2	<i>Temperature Sensor der Android Sensor Box</i>	81
.1	<i>Ordnerstruktur der beigelegten DVD</i>	92



# Listingverzeichnis

3.1	Testbeispiel mit <i>Instrumentation</i> . . . . .	16
3.2	Testbeispiel mit <i>Positron</i> . . . . .	19
3.3	Testbeispiel mit <i>monkeyrunner</i> . . . . .	21
3.4	Testbeispiel mit <i>Robotium</i> . . . . .	23
3.5	Testbeispiel mit <i>Robolectric</i> . . . . .	25
3.6	Testbeispiel mit <i>UiAutomator</i> . . . . .	27
3.7	Initialisierung des <i>SensorSimulators</i> . . . . .	29
5.1	Beispiel eines synchronen Manipulationsaufrufs . . . . .	44
5.2	Aufruf des <i>BroadcastReceivers</i> weckt alle schlafenden Testprozesse auf . . . . .	45
5.3	Erweiterung eines <i>Robotium</i> tests mit Manipulationsanweisungen . . . . .	45
5.4	Aufbau der Klasse <i>ZombyServlet</i> . . . . .	52
5.5	Aufbau einer <i>Telnet</i> verbindung . . . . .	53
5.6	Ausführung und Auswertung einer <i>Telnet</i> verbindung . . . . .	53
5.7	Implementierung eines <i>Post-Requests</i> mit <i>Android</i> -Methoden . . . . .	55
5.8	Aufbau einer <i>Low-Level</i> -Klasse . . . . .	56
5.9	Aufbau einer <i>High-Level</i> -Klasse . . . . .	57
5.10	Verwendung des <i>Zomby</i> -Frameworks mithilfe der Klasse <i>Zomby</i> . . . . .	59
5.11	Inhalt einer <i>Zomby Recorder</i> -Datei . . . . .	61
5.12	<i>Zomby</i> : Aufgezeichnete Sensordaten wieder abspielen . . . . .	62
5.13	<i>Zomby-Recorder</i> : Generierte Testmethode mit aufgezeichneten Sensordaten . . . . .	63
6.1	<i>Zomby-Test</i> : Automatisierung der Batteriewerte . . . . .	65
6.2	<i>Zomby-Test</i> : Automatisierung eines Batterieverlaufs . . . . .	66
6.3	<i>Zomby-Test</i> : Automatisierung einer schwankenden Datenverbindung mit plötzlichem Funkloch . . . . .	67
6.4	<i>Zomby-Test</i> : Simulation eines Fußgängers, der eine Straße entlanggeht . . . . .	67
6.5	<i>Zomby Recorder</i> : Lokalisierungsaufzeichnung einer Zugfahrt (Ausschnitt) . . . . .	68
6.6	<i>Zomby-Test</i> : Simulation einer Zugfahrt . . . . .	69
6.7	<i>Zomby-Test</i> : Automatisierung eines Temperaturverlaufs . . . . .	69

---

7.1	Anwendungstestfall mit Robotium und Zomby . . . . .	76
7.2	Zomby-Test: Simulation eines Temperaturverlaufs . . . . .	79
7.3	Ausschnitt aus Kugellabyrinth-Test . . . . .	83

# Tabellenverzeichnis

3.1	Zusammenfassung der bisherigen Lösungen . . . . .	31
4.1	Übersicht über Manipulationsbefehle des Android Emulators . . . . .	37
4.2	Zusammenfassung der Lösungsansätze . . . . .	38
5.1	Übersicht über Manipulationsbefehle des Android Emulators (Ergänzung) . . . .	51
6.1	<i>Liste aller unterstützten Sensortypen der Android-Plattform</i> . . . . .	71
6.2	Vom Zomby-Framework unterstützte Umgebungsvariablen . . . . .	74

# 1 Einleitung

Die mobile Softwareentwicklung hat das Verhalten der modernen Gesellschaft verändert. Die kleinen Helfer, sogenannte „Apps“, sind aus dem Alltag kaum wegzudenken. Wie selbstverständlich werden sie mittlerweile von Jung und Alt genutzt. Obwohl die Entwicklung mobiler Software schon in den neunziger Jahren begann, sollte der revolutionäre Durchbruch mobiler Software erst mit der Einführung des Apple App Stores im Jahre 2008 beginnen, ein Jahr nach Steve Jobs' Vorstellung des iPhones im Juni 2007. Apple hatte den App Store eingeführt, um Drittherstellern die Möglichkeit zu geben, Software für das damalige *iPhone OS*<sup>1</sup> zu schreiben, die sie über diese Plattform verkaufen konnten. Fast zeitgleich führte auch Google mit der ersten Android-Version den Android Market<sup>2</sup> ein. Zu dieser Zeit gab es allerdings keine zum iPhone konkurrierenden Android-Geräte, so dass die Entwickler fast ausschließlich für die iOS-Plattform entwickelten. Die Android-Plattform wuchs erst sehr langsam, später aber um so schneller, was dazu führte, dass sie im Januar 2013 die iOS-Plattform erstmals überholte (Android 800.000 und iOS 775.000 Apps im Store) [Chi13].

Mit dem revolutionären Erfolg der Smartphones und der Tablet-PCs kam gleichzeitig auch der sehr große Bedarf an mobiler Software<sup>3</sup>. Aufgrund von immer leistungstärkeren Geräten wuchsen auch die Anforderungen an mobile Software. Nachdem die Apps am Anfang meist nur eine Funktion unterstützten, gleichen sie heutzutage immer mehr einer Desktop-Applikation, inklusive ihrer Komplexität. Das fordert natürlich seinen Tribut, denn der Entwicklungsaufwand von mobiler Software nähert sich dem von Desktop-Software. Mit der wachsenden Komplexität wird das Testen von mobiler Software zunehmend bedeutender. Das Testen von mobiler Software steckt jedoch noch in den Kinderschuhen. Ein Indiz findet sich in der passenden Literatur. Es gibt kaum Bücher, die sich ausschließlich mit dem Testen mobiler Anwendungen beschäftigen. Es steckt also noch sehr viel Potenzial in diesem, doch recht neuen Forschungsgebiet. Bislang wurde meist versucht die bestehenden Verfahren aus dem Desktop-Bereich in den mobilen Bereich zu adaptieren, wie beispielsweise Unit-Tests. Eine eigene Testkultur muss sich demnach erst noch mit der Zeit entwickeln. Diese Arbeit soll dazu beitragen, dass diese

---

<sup>1</sup>heute: iOS

<sup>2</sup>heute: Google Play

<sup>3</sup>Apple verzeichnete am 7. Januar 2013 insgesamt 40 Milliarden App-Downloads [App13]

Entwicklung vorangetrieben wird.

Einschränkend soll aber noch an dieser Stelle erwähnt werden, dass diese Arbeit nur den Android-Kontext beleuchtet. Die Entwicklungsumgebung für Android unterscheidet sich zu anderen mobilen Plattformen, wie iOS, teilweise sehr stark. Neben den unterschiedlichen SDKs und Lizenzmodellen, spielen hier auch noch unterschiedliche Entwicklungstools eine große Rolle. Für die Zielvorgabe dieser Arbeit ist dies von entscheidender Bedeutung, so dass die erforschten Erkenntnisse nur teilweise auf andere Systeme übertragen werden können.

## **1.1 Gegenstand und Ziel der Master-Thesis**

Die Softwareentwicklung wächst stetig. Erst vor kurzem wurde die eine Milliarde-Grenze von weltweit verwendeten Smartphones geknackt. Analytiker prognostizieren, dass in drei Jahren die zweite Milliarde erreicht sein wird [Hbl12]. Es ist also immer noch ein enormer Wachstumsmarkt vorhanden. Die Entwickler sind demnach im ständigen Zugzwang schnell neue Software zu entwickeln. In der noch jungen Epoche der mobilen Softwareentwicklung werden die Entwickler jedoch mit einer neuen Situation konfrontiert. In der Vergangenheit wurde ausschließlich Software entwickelt, bei der der Standort sowie andere Parameter, wie sie in modernen Handys eingesetzt werden, keine Rolle gespielt haben. In der modernen Softwareentwicklung ist es jedoch so, dass solche Parameter nicht nur eine Rolle spielen, sondern meist im Vordergrund stehen. Diese Tatsache hat für die Entwickler eine enorme Auswirkung auf das Testen der Software. In der Praxis hat sich gezeigt, dass mobile Software weit nicht so gut und umfassend wie „stationäre“ Software getestet wird. Dies liegt zum einen an dem bereits erwähnten Leistungsdruck und zum anderen an den noch unzureichenden Möglichkeiten, die Software mit mobilen Umgebungsänderungen (Batteriestand, Lokalisierung, verschiedene Netzwerkverbindungen) automatisiert zu testen.

Das Ziel dieser Arbeit ist es, einen Weg zu finden diese Problematik zu lösen. Im nächsten Unterkapitel wird erklärt, wie dieses Ziel erreicht werden soll bzw. wie Android-Programme auf umgebungsändernde Parameter automatisiert getestet werden können.

## **1.2 Verlauf und Aufbau der Master-Thesis**

Die Arbeit beginnt mit den technischen Grundlagen in Kapitel 2. Dort werden wichtige Begriffe und Technologien eingeführt, um die späteren Schritte und Entscheidungen der Arbeit nachvollziehen zu können. In Kapitel 3 sollen zunächst bestehende Lösungen untersucht werden.

Die Analyse soll dabei aufzeigen, warum die bisherigen Lösungen die hier vorgegebene Problemstellung nicht lösen können. In Kapitel 4 findet deshalb eine Identifikation möglicher neuer Lösungsansätze statt. In diesem Kapitel sollen technische Lösungswege gefunden werden, um die Simulation von mobilen Umgebungsvariablen zu ermöglichen. Dem erfolgversprechendsten Lösungsansatz soll nachgegangen und entsprechend in Kapitel 5 umgesetzt werden. In diesem Kapitel sollen neben der eigentlichen Implementierung der Lösung auch die vorher auftretenden Hindernisse und Probleme erläutert werden, die es im Zuge der Implementierung zu lösen gilt. In Kapitel 6 findet die Evaluierung der Lösung statt, bei der ermittelt werden soll, wie gut sie die vorgegebene Problemstellung löst. Im bestmöglichen Fall werden mit ihr alle Parameter unterstützt, die für die mobile Softwareentwicklung entscheidend sind. Nach der Evaluierung folgt ein Zwischenkapitel, indem kurz erläutert wird, wie die neue Lösung in der Praxis eingesetzt werden kann. Abschließend fasst das Kapitel 8 diese Arbeit noch einmal zusammen und gibt einen kurzen Ausblick über die weiteren zukünftigen Entwicklungsmöglichkeiten sowie das mögliche Potenzial der neuen Lösung.

### 1.3 inovex GmbH

Diese Arbeit entsteht im Auftrag der Firma **inovex GmbH**<sup>4</sup>. Die inovex GmbH ist ein inhabergeführtes IT-Projekthaus. Sie beschäftigt zur Zeit ungefähr 120 Mitarbeiter an vier verschiedenen Standorten innerhalb Deutschlands (Pforzheim, Karlsruhe, München und Köln). inovex ist spezialisiert auf IT-Projekte im Internet und Enterprise Umfeld und betreut branchenübergreifend verschiedene Kunden wie z.B. Daimler, ProSiebenSat1 Media AG, Porsche, 1&1, maxdome, Bosch, etc.

inovex besteht aus fünf Geschäftsbereichen: Consulting, Application Development, Business Intelligence, IT Engineering sowie Training. Vorträge und Publikationen haben bei inovex einen hohen Stellenwert, weshalb die Firma auf zahlreichen Konferenzen<sup>5</sup> und in diversen Magazinen<sup>6</sup> präsent ist.

---

<sup>4</sup><http://www.inovex.de>

<sup>5</sup><http://www.inovex.de/news-events/vortraege>

<sup>6</sup><http://www.inovex.de/news-events/publikationen>

## 2 Grundlagen

Um die Nachvollziehbarkeit der Arbeit zu gewährleisten, werden in diesem Kapitel alle wichtigen Begriffe, Tools und Technologien erklärt bzw. vorgestellt, auf denen die Arbeit unter anderem aufbaut. Sollten im weiteren Verlauf der Arbeit weitere Fachbegriffe auftauchen, werden sie, so weit es notwendig ist, im entsprechenden Kontext erläutert.

### 2.1 Dynamische Tests

Die Entwicklung von Software ist sehr komplex. Die Fehleranfälligkeit von Software ist dadurch sehr hoch. Aus diesem Grund ist das Testen von Software unumgänglich. Im Gegensatz zu einem statischen Test wird bei einem dynamischen Test das zu testende Programm ausgeführt. Dabei wird zwischen drei verschiedenen Vorgehensweisen unterschieden.

#### 2.1.1 Black-Box-Test

Bei einem Black-Box-Test ist die innere Struktur eines Programms nicht bekannt oder wird ignoriert. Das Verhalten des Programms wird anhand der Spezifikation getestet. Die Daten, die durch den Test generiert werden, werden mit den erwarteten Daten verglichen. Die wichtigsten Metriken eines Black-Box-Tests sind *Abdeckung aller Funktionen*, *Abdeckung aller Eingaben* und *Abdeckung aller Ausgaben*. Dies ist jedoch in der Praxis nicht zu erreichen, weil es bereits bei einer Eingabe mit einer erwarteten Ganzzahl theoretisch unendlich viele Eingabewerte geben könnte und weil die Kosten für den Aufwand eine natürliche Grenze darstellen. Daher wird mit den verbliebenen Tests angestrebt, die maximale Anzahl an Fehlern zu finden [Wet07].

#### 2.1.2 White-Box-Test

Im Unterschied zum Black-Box-Test ist bei einem White-Box-Test die innere Struktur des Programms bekannt. Der Entwickler schreibt dementsprechend Tests mit Kenntnis des Quellcodes oder auf dem darauf basierenden Algorithmus. Die wichtigsten Metriken bei einem White-Box-Test sind *Abdeckung aller Anweisungen*, *Abdeckung aller Verzweigungen* und *Abdeckung aller*

*Bedingungen bei Verzweigungen.* Ähnlich wie bei den Black-Box-Tests können auch hier die vorgegebenen Ziele in der Praxis nicht erreicht werden [Wet07].

### 2.1.3 Grey-Box-Test

Bei diesem Test vereinen sich die Vorteile des Black-Box-Tests und die des White-Box-Tests. Beim Schreiben dieser Tests kann der Entwickler einerseits Bezug auf die Implementierungsdetails nehmen, weil er die innere Struktur kennt (Vorteil des White-Box-Tests), muss ihn aber andererseits vor der eigentlichen Implementierung schreiben, so dass der eigentliche Quelltext nicht bekannt ist (Vorteil des Black-Box-Tests) [Wet07]. Bekanntester Vertreter dieses Verfahrens bildet das *Test Driven Development* (TDD) [Bec03].

## 2.2 Testarten

### 2.2.1 Stresstest

Ein Stresstest ist eine Ausprägung eines Lasttests. Die zu testende Software wird absichtlich überfordert, indem sie beispielsweise mehr Eingaben (Last) bewältigen muss als sie eigentlich kann. Das Ziel dieses Tests ist es herauszufinden, ob das zu testende Programm robust genug ist, um auch bei einer Überlastung oder falscher Bedienung fehler- und absturzfrei durchzulaufen. Ansonsten müssen Strategien oder Mechanismen entwickelt werden, mit denen das Fehlverhalten der Software vermieden werden kann. [Hof08].

### 2.2.2 UI-Test

Bei einem UI<sup>1</sup>-Test wird die grafische Benutzeroberfläche einer Software getestet. Es findet ein Dialog zwischen Mensch (Benutzer) und Maschine statt. Dabei werden korrekte und unkorrekte Eingaben des Benutzers getestet, um das grafische Verhalten der Software zu validieren. Überprüft wird beispielsweise, ob ein Dialogfenster korrekt mit dem Benutzer interagiert oder ob dynamische Buttons erzeugt und richtig dargestellt werden. Diese Tests sind meist sehr zeitaufwendig, weil die Tests aufgrund der Interaktion des Benutzers in Echtzeit ablaufen müssen [SL10].

Die Dauer und der Aufwand dieser Tests können jedoch durch die Nutzung von Testframeworks wie Robotium und UiAutomator verringert werden. Diese Frameworks simulieren die Interaktion des Benutzers, um UI-Tests zu automatisieren.

---

<sup>1</sup>User Interface



## 2.3 JUnit

JUnit ist ein weit verbreitetes Testframework für Java-Programme. Es ermöglicht die Automatisierung von Unit-Tests<sup>2</sup> (Klassen oder Methoden) [JU113].

In oberflächenbasierten Programmen wie Eclipse<sup>3</sup> werden die Testergebnisse mit einem grünen (Test war erfolgreich) oder einem roten Balken (Test war nicht erfolgreich) dargestellt (siehe Abbildung 2.1). Die Entscheidung ist demnach binär. Im nicht erfolgreichen Fall wird jedoch noch zwischen einem Fehler (*Error*) und einem unerwarteten Ergebnis (*Failure*) unterschieden. Während beim *Failure* ein erwartetes Ergebnis nicht eintritt, kommt es beim *Error* zu einem unerwarteten Fehlverhalten. In vielen IDEs<sup>4</sup> ist JUnit bereits integriert. Unter dem Begriff xUnit wird eine ganze Reihe von JUnit-ähnlichen Frameworks zusammengefasst, die neben Java auch noch viele andere Programmiersprachen unterstützen [Ham09].

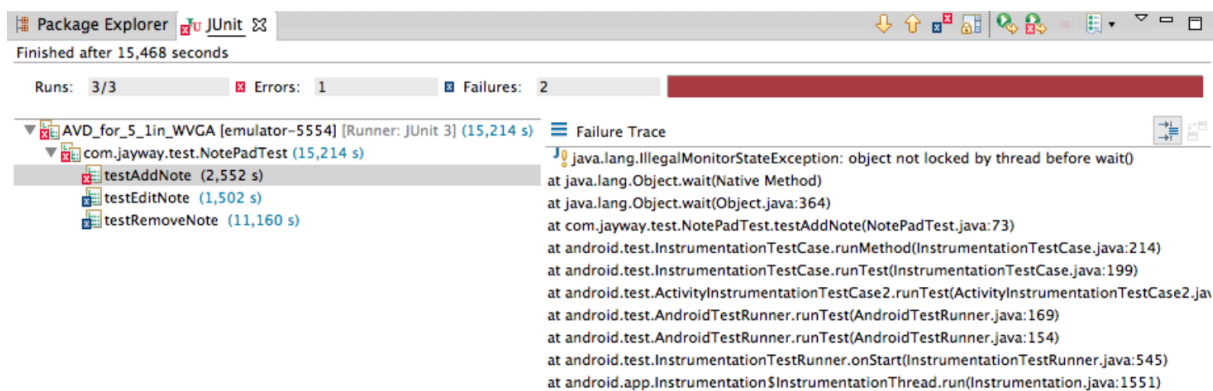


Abbildung 2.1: JUnit: Darstellung der Testergebnisse

## 2.4 Android SDK

Das Android SDK (Software Development Kit) ermöglicht das Programmieren von Android Programmen. Es enthält neben der notwendigen Java-Bibliothek noch weitere wichtige und nützliche Entwicklungstools [Goo13k]. In den nachfolgenden Abschnitten werden die wichtigsten vorgestellt.

<sup>2</sup>auch Modultest, prüft funktionale Einzelteile (Module) von Computerprogrammen auf korrekte Funktionalität

<sup>3</sup><http://www.eclipse.org>

<sup>4</sup>Integrierte Entwicklungsumgebung (Abkürzung IDE, von engl. integrated development environment)

### 2.4.1 Emulator

Der Android Emulator ist ein wichtiges Tool innerhalb des SDKs. Mit diesem können Android-Geräte emuliert werden, so dass Android-Anwendungen direkt auf dem Emulator ausgeführt werden können und die Verwendung eines echten Gerätes entfällt. In Abbildung 2.2 ist beispielsweise eine Virtualisierung eines *Google Nexus One* zu sehen.

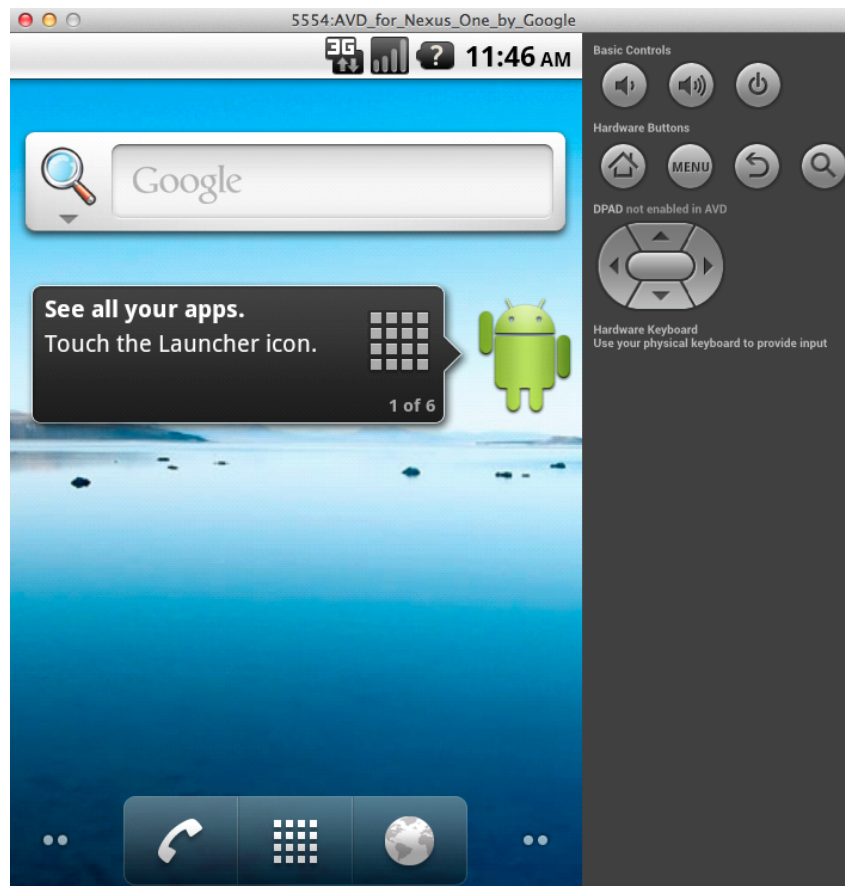


Abbildung 2.2: Android Emulator: Virtualisierung des Nexus One

### Emulator vs. Simulator

Im Vergleich zu einem Simulator, der bei der iOS-Entwicklung zum Einsatz kommt, bildet ein Emulator eine bestimmte Hardware und dem darin laufenden System ab. Alle notwendigen Hardware- und Softwarekomponenten werden virtuell bereitgestellt. Im Android Emulator startet ein komplettes Android-Betriebssystem. Die Simulation der Komponenten wie Prozessor, Speicher, Netzwerkschnittstellen, etc. übernimmt die darum liegende Emulationssoftware. Der Simulator hingegen emuliert keine Hardware, sondern stellt benötigte APIs bereit, simuliert eine Laufzeitumgebung sowie bestimmte Ereignisse, die sonst vom Betriebssystem erzeugt werden. [Hel13].

## Konfiguration

Die Konfiguration des Android Emulators ist sehr umfangreich. Durch eine Vielzahl an Parametern, die dem Entwickler bei der Auswahl zur Verfügung steht, können nahezu beliebige Gerätekonfigurationen emuliert werden. Dabei sind die wichtigsten Parameter:

- Android-Betriebssystem-Version (z.B. 2.3.3)
- Physikalische Größe des Displays (z.B. 4,7 Zoll)
- Auflösung des Displays (z.B. 1280x800)
- Größe des Arbeitsspeichers (z.B. 512 MByte)
- CPU-Architektur (z.B. ARM)
- Hardwarekomponenten (z.B. Kamera)
- Vorhandene Eingabekomponenten (z.B. Keyboard)
- Größe des Festspeichers, der SD-Karte (z.B. 512 MByte)

Verwaltet werden diese und weitere Parameter in sogenannten Android-Virtual-Device-Konfigurationen (AVDs). Dabei stellt jede AVD ein eigenständiges Gerät mit eigenem internem Speicher und getrennter SD-Karte dar. Starten lassen sich die AVDs mithilfe des Virtual Device Managers (siehe nächster Abschnitt) oder über die Konsole mit dem Aufruf [Goo13d]:

```
$ emulator -avd [avdname]
```

Die Auflistung aller verfügbaren AVDs ermöglicht der Befehl:

```
$ android list avd
```

### 2.4.2 Virtual Device Manager

Dieses Tool bietet dem Entwickler eine angenehme Möglichkeit virtuelle Android-Geräte zu konfigurieren und zu verwalten. Abbildung 2.3 zeigt das Konfigurationsfenster, in dem die Geräteeigenschaften angegeben werden können.

Ist die gewünschte AVD erstellt, wird sie im AVD Manager aufgelistet und kann per Knopfdruck über den Android Emulator gestartet werden.

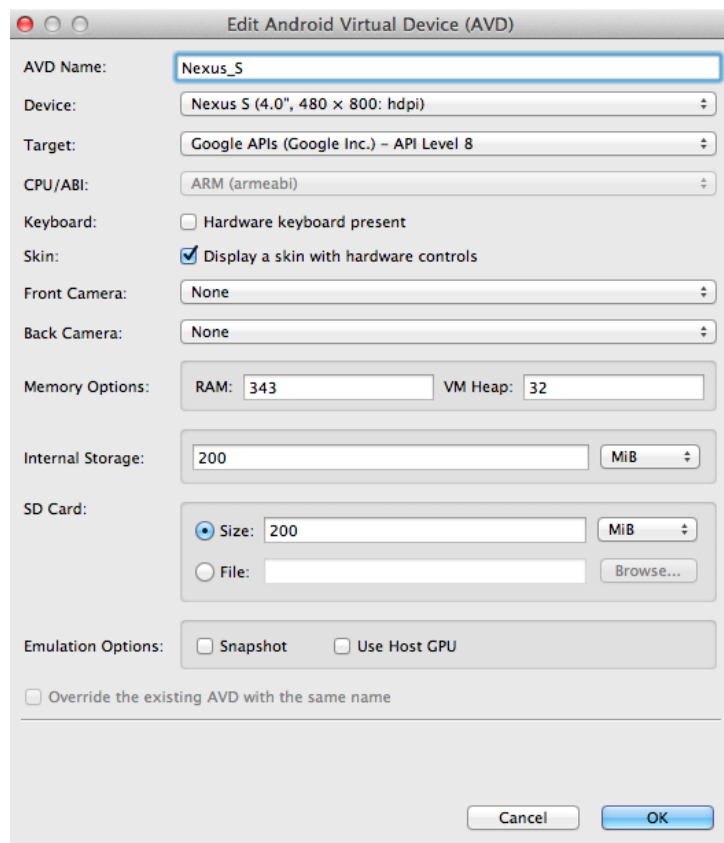


Abbildung 2.3: Konfigurationsfenster des AVD Managers

### 2.4.3 Debug Bridge

Die Android Debug Bridge (adb) ist ein Kommandozeilenprogramm, welches die Kommunikation mit dem Android Emulator oder einem angeschlossenen Android-Gerät ermöglicht. Zu den Kernaufgaben dieses Programmes gehören Installieren und Starten von Programmen, Datenaustausch, Ausführen von Shell-Befehlen, Weiterleiten von Ports (port forwarding), Auslesen von Logdateien und die Verwaltung der Geräteinstanzen [Goo13a].

## 2.5 Android Activity Lifecycle

Im Gegensatz zum Desktop-Rechner muss ein Smartphone auf viele äußere Ereignisse reagieren. Um auf diese angemessen zu reagieren, entscheidet das mobile Betriebssystem in einigen Fällen eigenständig, wann ein Programm geschlossen und ein anderes aufgerufen wird. Wenn ein Anruf eingeht wird die Anwendung, die zur Zeit im Vordergrund läuft, umgehend geschlossen und die Telefonanwendung wird gestartet. Damit bei diesem Vorgang nicht ständig Daten verloren gehen, durchläuft jede Android-Anwendung einen ganz bestimmten Lebenszyklus (Activity Lifecycle). Dieser sorgt dafür, dass die Software auf solche oder ähnliche

Ereignisse reagieren kann. Wie der Lebenszyklus im Detail aussieht zeigt Abbildung 2.4.

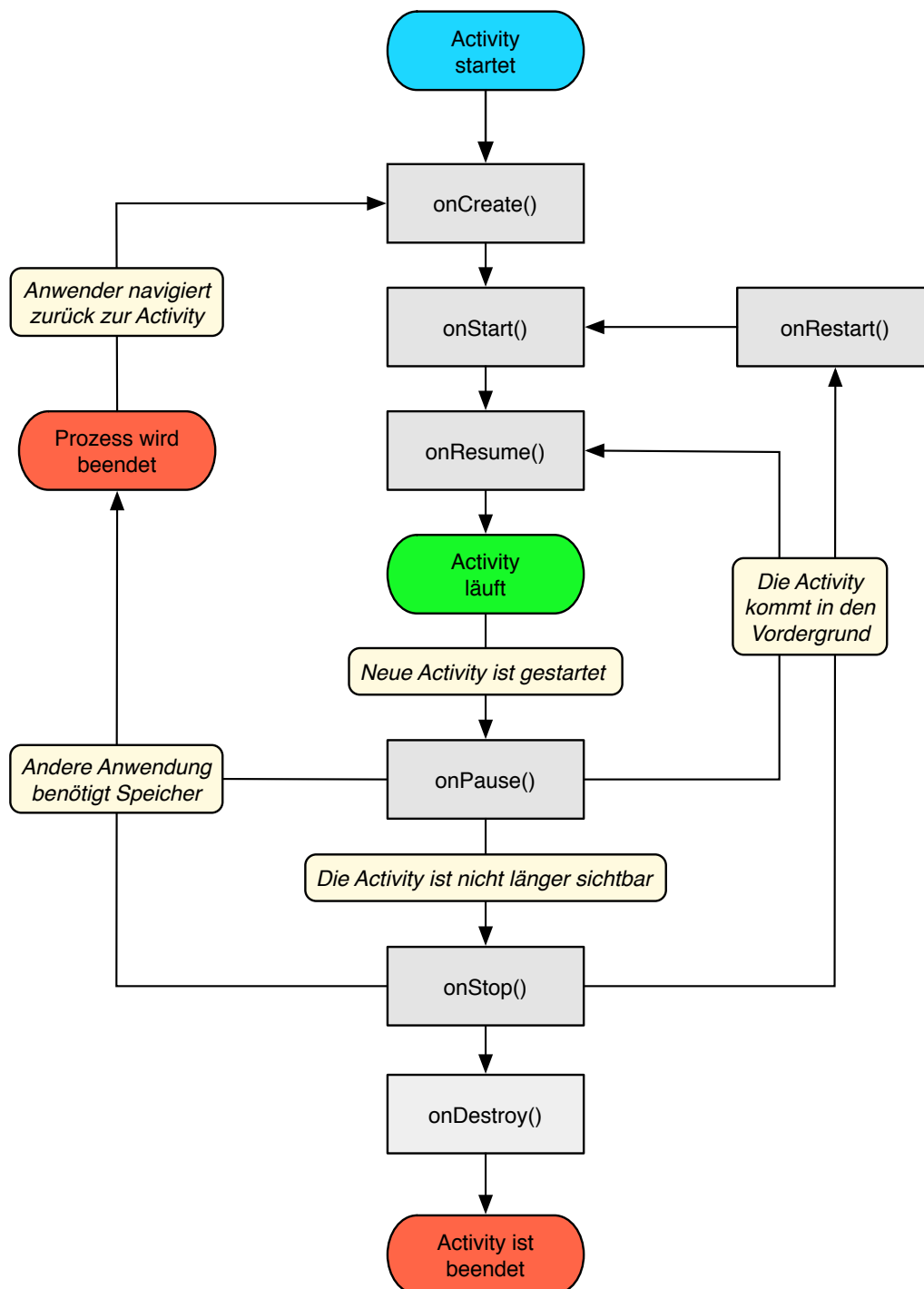


Abbildung 2.4: Aktivitätsdiagramm vom Android Activity Lifecycle (Angelehnt an [Raj12])

Wenn das Programm erzeugt, (wieder) gestartet, fortgesetzt, pausiert, gestoppt oder zerstört wird, folgt ein Aufruf der passenden `on`-Methode. Diese kann der Entwickler nutzen, um beispielsweise beim Beenden des Programms ungesicherte Daten zu persistieren [Goo13c].

## 2.6 LogCat

LogCat ist ein standardisierter, systemweiter Logging-Mechanismus, der jedem Java- und C/C++-Code zur Verfügung steht. Die angezeigten Log-Nachrichten können sehr einfach und nach Dringlichkeit (Detailreich, Fehlerbehebung, Information, Warnung, Fehler und Forderung) oder nach Tags gefiltert werden. Der Inhalt des LogCats kann über den Kommandozeilenbefehl:

```
$ adb logcat
```

oder über Eclipse eingesehen werden (siehe Abbildung 2.5) [Gar11].

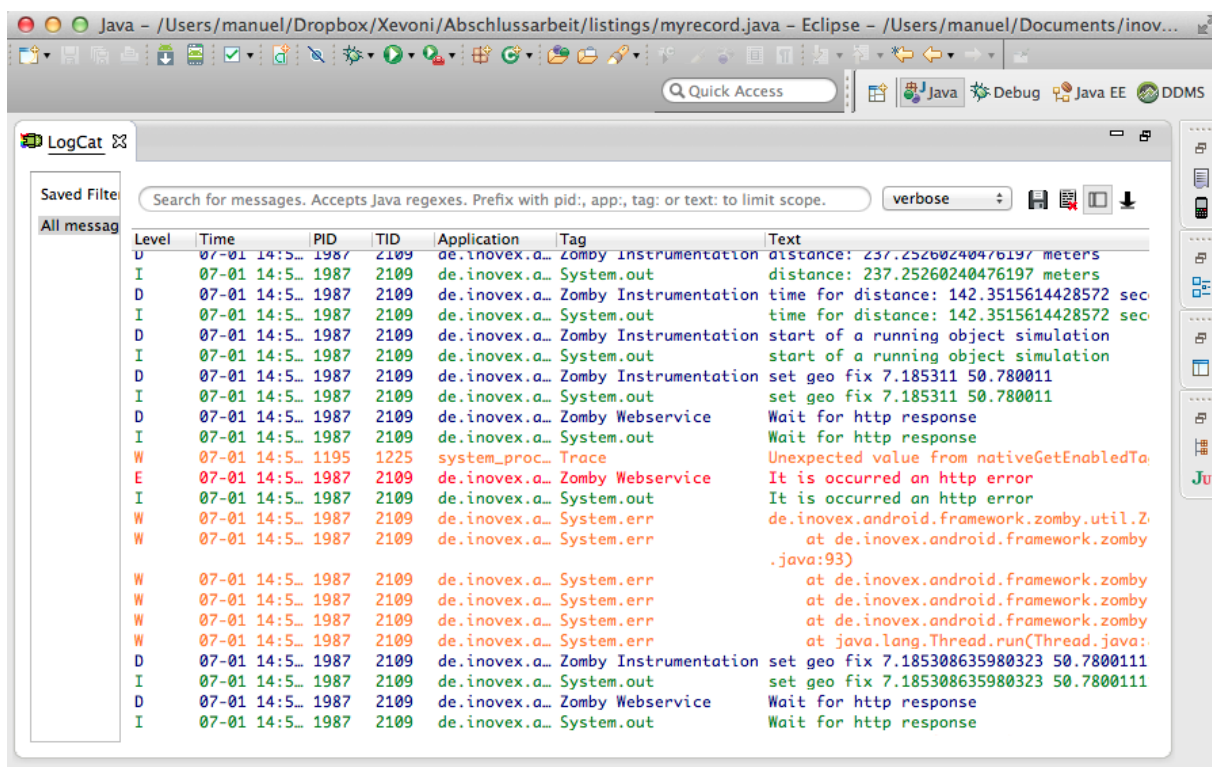


Abbildung 2.5: LogCat-Ausgabe über Eclipse

## 2.7 Telnet

Mit Telnet (Telecommunication Network) wurde in den 70er Jahren ein einheitlicher Standard (RFC 854, 1983) geschaffen, der heute noch wegen seiner Einfachheit vereinzelt für fern-administrative Zwecke<sup>5</sup> benutzt wird, obwohl er aus sicherheitstechnischer Sicht überholt ist. Telnet ist ein einfaches Protokoll, das eine interaktive Verbindung zwischen zwei entfernten

<sup>5</sup>z.B. Mail via SMTP/POP3

vernetzten Rechnern regelt. Um per Telnet auf einen Rechner zuzugreifen, muss auf dem Zielrechner ein Telnet-Server aktiv sein. Der Client kann ihn dann über den Port 23 kontaktieren [Olb03].

## **3 Analyse bisheriger Lösungen**

Bevor nun ein neuer Lösungsansatz identifiziert werden kann, muss erst einmal der aktuelle Markt analysiert werden. Unterkapitel 3.2 zeigt welche Möglichkeiten dem Android-Entwickler aktuell zur Verfügung stehen, um automatisiert testen zu können. Entscheidend bei dieser Betrachtung ist die Perspektive. Aus diesem Grund wird in Unterkapitel 3.1 erläutert, welche Kriterien für die Analyse von Interesse sind. Wie gut die Lösungen die betrachtenden Parameter automatisiert testen können, erfolgt abschließend in Unterkapitel 3.3.

### **3.1 Kriterien der Analyse**

Da sich diese Arbeit nur auf das automatisierte Testen von umgebungsabhängigen Parametern beschränkt, werden lediglich die Parameter betrachtet, die dafür von Bedeutung sind. Beispielhafte Szenarien zeigen dabei, wie mögliche automatisierte Tests aussehen könnten.

#### **3.1.1 Netzwerkverbindung**

Es gibt generell zwei Möglichkeiten eine Netzwerkverbindung aufzubauen, und zwar entweder per WLAN oder per Mobilfunknetz. Beim Mobilfunknetz stehen, je nach Standort, verschiedene Netze mit unterschiedlichen Netzwerkgeschwindigkeiten zur Verfügung. Gerade in diesem Kontext muss eine Software, die Daten synchronisiert, besonders flexibel sein. Beispielsweise eine News-App könnte abhängig von der Netzwerkgeschwindigkeit, entweder nur Text (bei EDGE), Text und niedrig aufgelöste Bilder (UMTS) oder Text und hoch aufgelöste Bilder (WLAN) auf den Client laden. Um diese drei Szenarien automatisiert testen zu können, müssen diese Verbindungen jeweils simuliert werden. Dabei sollte jedoch zusätzlich auch noch der Flugmodus betrachtet werden. In der Praxis kommt es häufiger mal vor, dass dieser Anwendungsfall nicht berücksichtigt wird und es später zum Absturz des Programms kommt.



### 3.1.2 Batterie

In der Energiebetrachtung sollten folgende Szenarien in Erwägung gezogen werden, auf die eine Software reagieren sollte:

- Die Batterie wird geladen (hängt am Netzteil)
- Die Batterie ist voll
- Die Batterie ist halbvoll
- Die Batterie hat noch 20%, 10%, 5% Ladekapazität

Bei kritischen Prozessen, ein Firmwareupdate zum Beispiel, sollte das Gerät am Netzteil hängen, um zu verhindern, dass das Gerät unerwarteterweise ausgeht. Aber auch bei nicht so kritischen Prozessen ist es wichtig, wie hoch die aktuelle Ladekapazität ist. Eine Software mit einem Update-Dienst, beispielsweise ein Chatprogramm, das alle paar Minuten im Hintergrund Daten abfragt, verbraucht verhältnismäßig viel Energie. Solange die Batterie noch mindestens halbvoll ist, ist dies für den Benutzer noch zu verschmerzen. Kommt der Ladezustand aber in den kritischen Bereich, so sollte die Software bei 20% Restakku nur noch alle 10-15 Minuten synchronisieren und später unter 10% Restakku ganz darauf verzichten und den Hintergrundprozess beenden. Die Software aktualisiert dann nur noch, wenn der Benutzer sie aktiv in den Vordergrund stellt.

### 3.1.3 Lokalisierung

Die Lokalisierung ist wohl einer der wichtigsten Anwendungsfälle, wenn es um mobile Geräte geht. Neben der klassischen Navigation (Straßenverkehr) existieren noch zahlreiche weitere Szenarien, bei denen GPS genutzt werden kann. Der GPS-Dienst wird mittlerweile sehr ausgiebig von modernen Apps genutzt. Entsprechend häufig und relevant sind GPS verbundene Tests. Die Lokalisierung kann über unterschiedliche Wege erfolgen. In den modernen mobilen Geräten ist meist ein physischer GPS-Chip integriert. Dieser kann auch ohne Internetverbindung seinen Standort per Satellit bestimmen. Daneben kann zusätzlich eine Standortbestimmung über das WLAN und Handynetz erfolgen.

Für automatisierte Tests ist es entscheidend, dass alle drei Verfahren simuliert werden können, um eine entsprechende Software darauf zu testen. Die Software könnte bei allen drei Verfahren anders reagieren, falls beispielsweise die Standortbestimmung per GPS-Chip sehr lange dauert. Ein weiteres wichtiges Szenario ist das Testen an verschiedenen Standorten. Normalerweise sind dafür Feldtests notwendig, um die entsprechenden GPS-Daten zu erhal-

ten. Um jedoch im Labor automatisiert testen zu können, müssen GPS-Daten auf Knopfdruck simulierbar sein.

### 3.1.4 Sensoren

In Smartphones und Tablets sind sehr viele Sensoren verbaut. Es folgt eine Auswahl der für die Praxis wichtigsten Sensoren<sup>1</sup>. Die Analyse beschränkt sich auf eine möglichst hohe Abdeckung der Sensortests, die in der Praxis tatsächlich durchgeführt werden.

#### Beschleunigungssensor

Der Beschleunigungssensor (*accelerometer*) misst die Beschleunigung im Raum [MS12]. Für diesen Sensor gibt es auch zahlreiche Anwendungsfälle. Der häufigste Anwendungsfall ist das Kippen des Geräts, um von der Portrait-Ansicht in die Landscape-Ansicht zu wechseln.

#### Gyroskop

Das Gyroskop beinhaltet Sensoren, die die Lage des Geräts im Raum angeben, indem es sechs Freiheiten misst [MS12]. In Spielen wird es oft als Joystick genutzt. Die Spieler kippen das Gerät vor und zurück, um so in einem Rennspiel Gas zu geben oder zu bremsen bzw. neigen es, um die Richtung des Autos anzugeben.

#### Kompass

Der Kompass gibt die Himmelsrichtung an und hilft zusätzlich zur Lokalisierung bei der Navigation. Es dient in der Praxis meist Fußgängern bei der Navigation. Mithilfe des Kompasses können sie herausfinden, in welche Richtung sie laufen müssen.

#### Näherungssensor

Der Näherungssensor (*proximity sensor*) misst über einen Infrarotsensor die Entfernung von einem sich nähernden Objekt zum Handy [MS12]. Wenn das Handy beim Telefonieren ans Ohr gehalten wird, schaltet sich das Display ab, um einerseits Energie zu sparen und andererseits unabsichtliche Eingaben zu verhindern.

---

<sup>1</sup>neben den schon erwähnten WLAN- und GPS-Sensor

## 3.2 Auflistung bestehender Lösungen

Es folgt nun eine Auflistung aller auf dem Markt bestehenden Lösungen, die automatisierte Black-Box Tests im Android-Kontext ermöglichen. Sie werden kurz vorgestellt und ihre Stärken und Schwächen erläutert. Kommerzielle Lösungen wie beispielsweise das Framework for Automated Software Testing (FAST) von Wind River [Win13] oder das Android Test Automation Framework (ATAF) von L&T Infotech [LT 13] werden hier aus Kostengründen nicht betrachtet.

Zusätzlich zu den Testframeworks werden im letzten Abschnitt auch Tools vorgestellt, die Sensordaten simulieren bzw. Sensordaten von echten Geräten weiterleiten können.

### 3.2.1 Instrumentation Framework

Das Android Instrumentation Framework (AIF) ist Bestandteil des Testing Frameworks innerhalb des Android SDKs (siehe Abbildung 3.1). Es stellt Testmethoden zur Verfügung, um eine Android-Anwendung in einem Testmodus zu starten, auszuführen, zu kontrollieren und zu beenden. Eine Android-Anwendung durchläuft normalerweise den *Android Lifecycle*. Dieser gibt vor, wann welche Funktion aufgerufen wird. Beispielsweise startet der *Activity-Lebenszyklus* durch einen Aufruf eines *Intents*. Zuerst wird die *onCreate()*-Methode aufgerufen, gefolgt von den Methoden *onStart()*, *onResume()* usw. (siehe Abschnitt 2.5). Das Android-Framework bietet keinen Weg an, diese Methoden direkt aufzurufen, das Instrumentation Framework jedoch schon [Goo13g].

Listing 3.1 gibt ein Beispiel an, bei dem eine *Activity* getestet wird.

```
public class NotepadTest extends ActivityInstrumentationTestCase2<
    Notepad> {
    // the activity under test
    private Notepad mActivity;

    public NotepadTest() {
        super("de.inovex.android.notepad", Notepad.class);
    }
    @Override
    protected void setUp() throws Exception {
        super.setUp();
        mActivity = this.getActivity();
    }
    public void testLifecycle() {
        callActivityOnCreate(mActivity);
    }
}
```

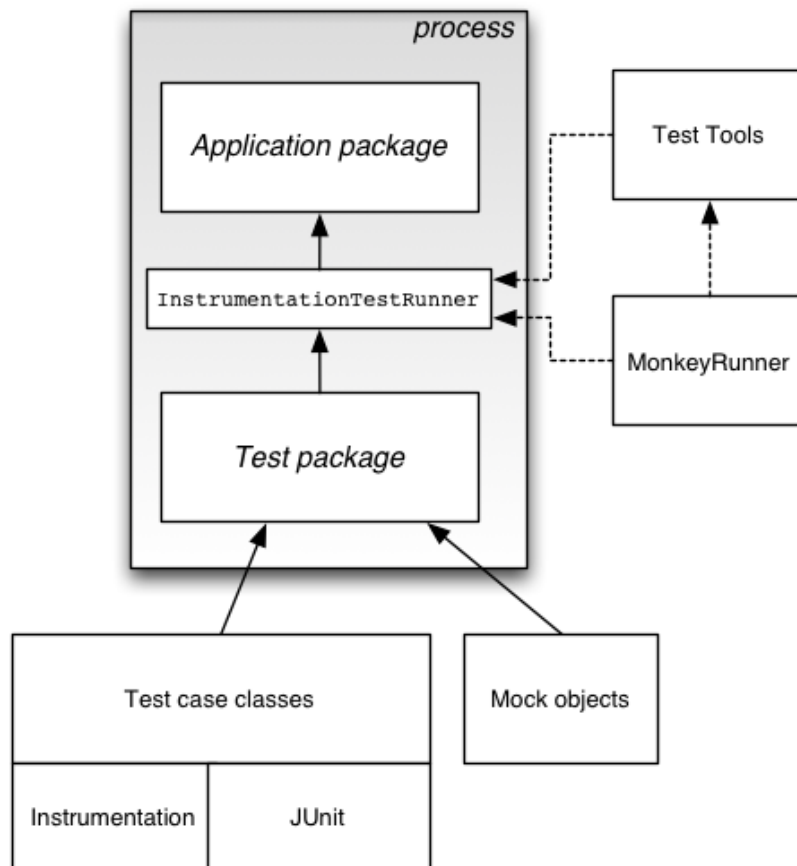


Abbildung 3.1: Zusammenfassung des Android-Testframeworks (Bildquelle: [Goo13g])

```

    callActivityOnStart(mActivity);
    callActivityOnResume(mActivity);
    callActivityOnPause(mActivity);
    callActivityOnResume(mActivity);
}

public void testAddPerson () {
    sendKeySync(KeyEvent.KEYCODE_M); //Type a letter
    sendKeySync(KeyEvent.KEYCODE_A); //Type a letter
    sendKeySync(KeyEvent.KEYCODE_R); //Type a letter
    sendKeySync(KeyEvent.KEYCODE_C); //Type a letter

    Button saveButton = (Button) myActivity.findViewById(R.id.
        button);
    saveButton.performClick();
    saveButton.setEnabled(false); //set UI element status
    Assert.assertFalse(saveButton.isEnabled());
}

```

Listing 3.1: Testbeispiel mit *Instrumentation*

Die Testmethode `testLifecycle()` testet einen kompletten Lifecycle, indem es alle entsprechenden Funktionen selbst aufruft. Die Methoden der Art `callActivityOn` machen es möglich. In der zweiten Testmethode `testAddPerson` wird eine neue Person namens „Marc“ hinzugefügt. Die Methode `sendKeysSync` simuliert das Drücken eines Buchstabens und die Methode `performClick()` das Drücken eines Buttons.

Das Framework besitzt eine Low-Level API. Obwohl damit komplexe und tiefe Tests möglich sind, bedeutet dies aber auch gleichzeitig einen größeren Aufwand für den Entwickler. Die Methode `sendKeysSync(KeyEvent.KEYCODE_M)` lässt nur die Eingabe einzelner Buchstaben zu. Für eine Eingabe eines ganzen Wortes benötigt der Entwickler abhängig von der Wortlänge entsprechend viele Zeilen Code. Zum Vergleich, die Eingabe ganzer Wörter ist mit einer High-Level API in nur einer einzigen Zeile Code möglich.

Neben *Activity* unterstützt das Framework auch noch das Testen von *Content Provider* und *Service*. Zudem setzt es auf JUnit auf, um die *assert*-Funktionalität zu nutzen. Später wird erläutert, dass die meisten Testframeworks diesen Architekturansatz wählen. Das Klassendiagramm des Instrumentation Frameworks ist in Abbildung 3.2 zu sehen.

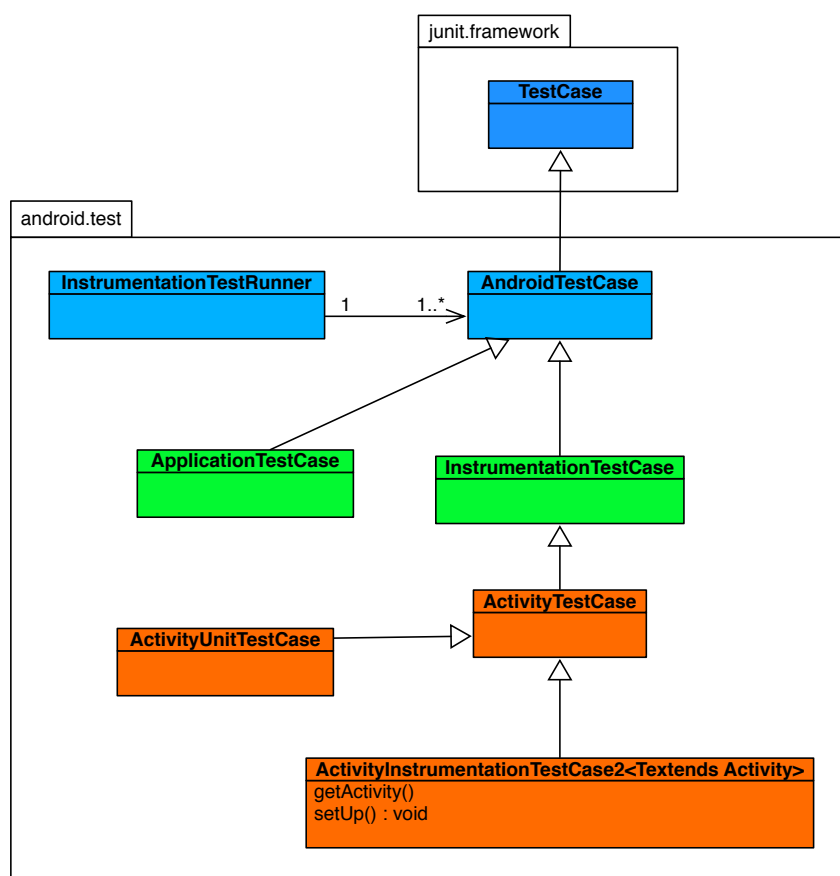


Abbildung 3.2: Android Instrumentation Framework Klassendiagramm (Angelehnt an [KM10])

### 3.2.2 Positron

Das Positron Framework setzt auf das Android Instrumentation Framework auf, um *Activity*-Ressourcen zu behandeln. Gleichzeitig bildet es eine Erweiterung des AIFs. Es bietet einen Selenium-ähnlichen High-Level-Ansatz, um Testfälle zu schreiben und auszuführen. Es benutzt ein Client-Server-Modell bei dem jeder Testfall einen Client darstellt und in seinem eigenen Thread läuft. Die *Activity* fungiert dabei als Server. Jeder Client erstellt seine eigene *Activity*-Instanz, mit der er mit dem Server kommuniziert. Als Kommunikationsschnittstelle dient dabei die Android Debug Bridge (adb). Genauso wie beim AIF setzt Positron auf JUnit auf. Die Threadsicherheit<sup>2</sup> des Frameworks ermöglicht die Kontrolle der UI-Elemente und deren Events in einem separaten Thread [Pos13].

Der Zugriff auf *Activity*-Ressourcen geschieht mithilfe eines Pfades, beginnend mit der aktuellen *Activity*. Die Struktur ist dabei hierarchisch aufgebaut. Die Auflösung des Pfades übernimmt eine Variante der *At*-Methode, beispielsweise *stringAt* oder *booleanAt*, wie es in Listing 3.2 zu sehen ist.

```
public class MyStories extends TestCase {

    @Test
    public void shouldSeeNoteEditorActivity() throws
        InterruptedException {
        startActivity("de.inovex.notepad", "de.inovex.notepad.
            NoteEditor");
        pause();
        assertEquals("NoteEditor", stringAt("class.simpleName"));
    }

    @Test
    public void addANote() {
        menu(NotesList.MENU_ITEM_INSERT, 0);
        press("NoteEditor is cool!", BACK);
        assertEquals("NoteEditor is cool!", stringAt("listView.0.text")
            );
    }
}
```

Listing 3.2: Testbeispiel mit *Positron*

Die Testmethode *shouldSeeNoteEditorActivity()* startet die Activity *NoteEditor* und überprüft

---

<sup>2</sup>Gleichzeitige mehrfache Ausführung von verschiedenen Programmbereichen möglich, ohne dass diese sich gegenseitig behindern

anschließend den Klassennamen auf Gleichheit mit einer *assert*-Methode. In der zweiten Testmethode wird der Menüpunkt `MENU_ITEM_INSERT` aufgerufen, um einen neuen Eintrag zu erstellen. Er enthält den Text „*NoteEditor is cool!*“. Ob der Vorgang erfolgreich durchgeführt wurde, wird wieder mithilfe von *assertEquals* getestet.

Positron hat seine eigenen Selenium-ähnlichen Kommandos implementiert, so dass zudem automatisierte High-Level GUI-Test-Suites erstellt werden können. Es unterstützt die Funktionalität, zwischen der zu testenden Anwendung und den auszuführenden Tests, zu synchronisieren. Dabei können die Tests gleichzeitig während des Testszenarios auf GUI-Elemente zugreifen [Pos13].

Zum Schluss muss noch erwähnt werden, dass das Positron Framework schon seit längerer Zeit nicht mehr gepflegt wurde. Der letzte Eintrag ist aus dem Jahr 2009<sup>3</sup>. Eine Weiterentwicklung oder ein Support sind demnach nicht mehr zu erwarten. Die Dokumentation ist zudem auch stark begrenzt.

### 3.2.3 UI/Application Exerciser Monkey

Der UI/Application Exerciser Monkey ist ein Kommandozeilenprogramm innerhalb des Android-SDKs, das Android-Programme auf dem Emulator oder auf einem Gerät ausführt und dabei zufällig irgendwelche User-Events simuliert. Diese können Klicken, Berühren, Touchgesten oder System-Level-Events sein. Mit diesem Tool können Stresstests durchgeführt werden [Goo13m].

Der folgende Befehl bewirkt, dass das Programm bzw. das Package *com.example.android.notepad* für 100<sub>ms</sub> ausgeführt wird.

```
$ adb shell monkey -p com.example.android.notepad -v 100
```

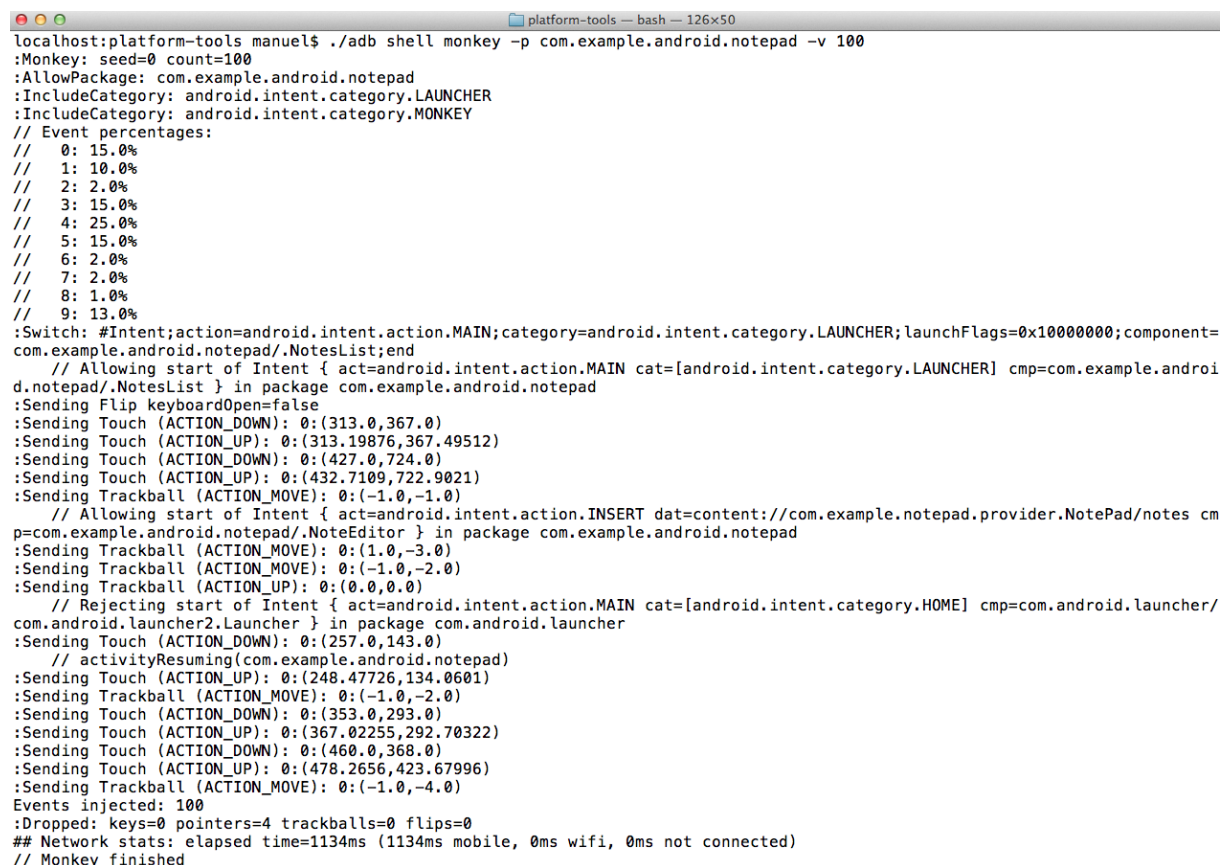
In Abbildung 3.3 ist ein Konsolenaufruf abgebildet. In der Ausgabe werden die jeweiligen zufällig generierten Events angezeigt.

### 3.2.4 monkeyrunner

Der *monkeyrunner* ist eine Test-API, die eine ausführbare Umgebung mitbringt, um Android-Programme zu testen. Die API beinhaltet Funktionen für den Zugriff auf das Gerät, das Installieren und Deinstallieren von Paketen, Bildschirmaufnahmen, den Vergleich zweier Bilder und

---

<sup>3</sup>festgestellt am 01.03.2013



```

localhost:platform-tools manuel$ ./adb shell monkey -p com.example.android.notepad -v 100
Monkey: seed=0 count=100
:AllowPackage: com.example.android.notepad
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
// 0: 15.0%
// 1: 10.0%
// 2: 2.0%
// 3: 15.0%
// 4: 25.0%
// 5: 15.0%
// 6: 2.0%
// 7: 2.0%
// 8: 1.0%
// 9: 13.0%
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10000000;component=
com.example.android.notepad/.NotesList;end
// Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.example.androi
d.notepad/.NotesList } in package com.example.android.notepad
:Sending Flip keyboardOpen=false
:Sending Touch (ACTION_DOWN): 0:(313.0,367.0)
:Sending Touch (ACTION_UP): 0:(313.19876,367.49512)
:Sending Touch (ACTION_DOWN): 0:(427.0,724.0)
:Sending Touch (ACTION_UP): 0:(432.7109,722.9021)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,-1.0)
// Allowing start of Intent { act=android.intent.action.INSERT dat=content://com.example.notepad.provider.NotePad/notes cm
p=com.example.android.notepad/.NoteEditor } in package com.example.android.notepad
:Sending Trackball (ACTION_MOVE): 0:(1.0,-3.0)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,-2.0)
:Sending Trackball (ACTION_UP): 0:(0.0,0.0)
// Rejecting start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.HOME] cmp=com.android.launcher/
com.android.launcher2.Launcher } in package com.android.launcher
:Sending Touch (ACTION_DOWN): 0:(257.0,143.0)
// activityResuming(com.example.android.notepad)
:Sending Touch (ACTION_UP): 0:(248.47726,134.0601)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,-2.0)
:Sending Touch (ACTION_DOWN): 0:(353.0,293.0)
:Sending Touch (ACTION_UP): 0:(367.02255,292.70322)
:Sending Touch (ACTION_DOWN): 0:(460.0,368.0)
:Sending Touch (ACTION_UP): 0:(478.2656,423.67996)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,-4.0)
Events injected: 100
:Dropped: keys=0 pointers=4 trackballs=0 flips=0
## Network stats: elapsed time=1134ms (1134ms mobile, 0ms wifi, 0ms not connected)
// Monkey finished

```

Abbildung 3.3: Konsolenaufwurf des UI/Application Exerciser Monkeys

das Laufen eines Testpakets gegen eine Anwendung. Wie schon der Exerciser Monkey ist auch der monkeyrunner ein Kommandozeilenprogramm [Goo13e].

Die API ist sehr übersichtlich. Es gibt drei Klassen - *MonkeyDevice*, *MonkeyRunner* und *MonkeyImage* - deren Klassennamen Rückschluss auf deren Aufgaben geben. Die enthaltenen Methoden bieten jedoch nur wenig Funktionalität und unterstützen daher lediglich primitivere Testmöglichkeiten. Zudem ist die Testauswertung durch das bloße Vergleichen von Screenshots nicht sehr komfortabel. Listing 3.3 zeigt, dass monkeyrunner-Tests nicht in Java, sondern in Python geschrieben werden. In der Java-dominierten Welt der Android-Umgebung stellt dies sicherlich eine Besonderheit dar.

```

# Imports the monkeyrunner modules used by this program
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice

# Connects to the current device, returning a MonkeyDevice object
device = MonkeyRunner.waitForConnection()

# Installs the Android package
device.installPackage('NotePad/bin/NotePad.apk')

```



```
# sets a variable with the package's internal name
package = 'com.example.android.notepad'

# sets a variable with the name of an Activity in the package
activity = '.NoteEditor'

# sets the name of the component to start
runComponent = package + '/' + activity

# Runs the component
device.startActivity(component=runComponent)

# Wait for few seconds
MonkeyRunner.sleep(2)

# Presses the Menu button
device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)

# Touch the new status button
device.touch(200, 390, 'DOWN_AND_UP')

# Wait for few seconds
MonkeyRunner.sleep(2)

# Takes a screenshot
result = device.takeSnapshot()

# Writes the screenshot to a file
result.writeToFile('NotePad/shot1.png', 'png')
```

Listing 3.3: Testbeispiel mit *monkeyrunner* [Goo13f]

Als erstes werden die Klassen *MonkeyRunner* und *MonkeyDevice* importiert, gefolgt von der Initialisierung des Geräts, des Paketnamens und Namens der Activity, auf die getestet werden soll. Nachdem die Activity mit der Methode `startActivity` gestartet und zwei Minuten gewartet wurde, erfolgt ein Klick auf den Menü-Button mit der Methode `press`. Das Drücken einer Berührungsgeste wird mit der Methode `touch` simuliert. Auf der Bildschirmkoordinate (200, 390) liegt ein Status-Buttons, der durch den Aufruf der Methode gedrückt wird. Nach einer zweisekündigen Pause wird von ein Foto dem Bildschirm gemacht und in „NotePad/shot1.png“ gespeichert.

Das Framework bietet zudem eine Schnittstelle für Plugins, um die monkeyrunner-Entwicklungsumgebung zu erweitern.

### 3.2.5 Robotium

Robotium ist ein Black-Box-Test Tool für Android. Es ist ein Open Source Projekt und liegt aktuell<sup>4</sup> in Version 3.6 vor. Dieses Framework bietet dem Entwickler die Möglichkeit, Benutzereingaben zu simulieren. Dem Entwickler stehen Befehle wie „klicke auf den Menüeintrag 'Optionen'“, „gehe zurück“, „dreh das Gerät in den Landscapemodus“, „schreib 'Achim' in das Textfeld“, „mache einen Screenshot“, etc. zur Verfügung [Rot13].

Da Robotium auf JUnit aufbaut, können die Tests in einer JUnit-Umgebung automatisiert ausgeführt werden. Listing 3.4 zeigt ein Testbeispiel.

```
public class NotePadTest extends ActivityInstrumentationTestCase2<
    NotesList>{

    private Solo solo;

    public NotePadTest() {
        super(NotesList.class);
    }

    @Override
    public void setUp() throws Exception {
        solo = new Solo(getInstrumentation(), getActivity());
    }

    public void testAddNote() throws Exception {
        solo.clickOnMenuItem("Add note");
        // Assert that NoteEditor activity is opened
        solo.assertCurrentActivity("Expected NoteEditor activity", "
            NoteEditor");
        // Add text in field 0
        solo.enterText(0, "NoteEditor is cool!");
        solo.clickOnButton("Save");
        // Assert that note are found
        assertTrue(solo.searchText("NoteEditor is cool!"));
        // Take a screenshot
```

---

<sup>4</sup>Stand 18.12.2012

```
solo.goBackToActivity("NotesList");
solo.takeScreenshot();
}

@Override
public void tearDown() throws Exception {
    solo.finishOpenedActivities();
}
}
```

Listing 3.4: Testbeispiel mit *Robotium* (Angelehnt an [ETP12])

Die Klasse *NotePadTest* erbt von der generischen JUnit-Klasse *ActivityInstrumentationTestCase2*. In ihr wird die Activity des zu testenden Programms angegeben, in diesem Fall *NotesList*. Vor dem Test wird die Methode `setUp()` aufgerufen. Dort wird das Solo-Objekt instanziiert. Die Klasse *Solo* enthält alle Testmethoden des Frameworks<sup>5</sup>. Die Methode `tearDown()` wird nach dem Test aufgerufen und schließt alle noch offenen Aktivitäten. Der eigentliche Test findet in der Methode *testAddNote* statt. Diese ruft den Menüpunkt „Add note“ auf. Anschließend wird als aktuelle Activity *NoteEditor* erwartet, was mit der Methode `assertCurrentActivity` überprüft wird. Auch hier wird die Notiz mit dem Text „NoteEditor is cool!“ gefüllt und mit dem Klick auf den Button „Save“ gespeichert. Ob das Speichern des neuen Eintrages erfolgreich war, wird mit `assertTrue` überprüft, indem die Anweisung `solo.searchText("NoteEditor is cool!")` ein `true` zurückliefert, wenn der Eintrag gefunden wurde. Abschließend wird eine Bildschirmaufnahme gemacht, nachdem vorher auf die Activity *NotesList* zurückgewechselt wurde.

Das Framework bietet volle Android 4.2 Unterstützung für *Activities*, *Dialogs*, *Toasts*, *Menus* und *Context Menus*. Die High-Level API enthält mächtige Methoden, durch die das Schreiben von Testcode vereinfacht wird. Der Testcode ist kurz und gut lesbar, nahezu selbsterklärend.

Die Automatisierung von Tests ist sehr komfortabel aufgrund von folgenden Funktionen:

- Automatisches Finden von *Views*
- Automatisches Folgen der aktuellen *Activity*
- Automatisches Timing

Das automatische Timing beinhaltet das Verzögern, bis ein bestimmtes Ereignis abgeschlossen ist. Es besitzt darüber hinaus ein intelligentes Verhalten, das eigene Entscheidungen fällt, wie beispielsweise das Scrollen einer Webseite, um einen bestimmten Link zu drücken.

---

<sup>5</sup>Solo ist die einzige Klasse des Testframeworks

### 3.2.6 Robolectric

Robolectric ist ebenfalls ein Open Source Framework, dass das Erstellen von Black-Box-Tests ermöglicht. Im Unterschied zu Robotium kommt es aber ohne Android-Gerät oder Emulator aus. Die Entwickler von Robolectric haben sogenannte *Shadow*-Klassen entwickelt, die das Verhalten von „echten“ Android-Klassen simulieren. Dafür wird das Programm nicht über die Dalvik VM ausgeführt, sondern über die JVM (Java Virtual Machine). Dadurch sind die Tests sehr performant, weil sie direkt auf dem Entwicklungsrechner ausgeführt werden. Damit dies aber funktioniert und die Android-Klassen in diesem Fall keine Exceptions werfen, werden sogenannte *Shadow*-Objekte genutzt. Diese Objekte arbeiten ähnlich wie ein Proxy oder ein Adapter. Meist gibt es für jede Android-Klasse eine entsprechende Shadow-Klasse, beispielsweise *Button* und *ShadowButton*, andernfalls kann die Adapter-Methode *shadowOf*<sup>6</sup> benutzt werden. Robolectric unterstützt keinen Android Lifecycle (siehe Abschnitt 2.5), stattdessen werden die für den Test benötigten Android Komponenten gemockt<sup>7</sup>. Einerseits führt das zwar zu einfacheren Unit-Tests, bei denen auf das Android-Verhalten nicht weitergegangen werden muss und als Folge einen wartbareren Testcode hat. Andererseits erschwert es dadurch den Aufwand von komplexeren Tests. UI-Tests sind zwar möglich, allerdings muss das Android-Verhalten aufgrund des Verzichts auf den Life-Cycle komplett simuliert werden, weil der Programmierer sämtliche Prozesse selber einleiten und anstoßen muss. Zudem sind auch nur Standard View-Klassen testbar, da das Framework die Alternativen wie OpenGL nicht unterstützt [Rol13].

Listing 3.5 zeigt ein kleines Testbeispiel mit drei Testmethoden. Dabei testet die erste Methode *shouldHaveAButtonThatSaysPressMe* auf einen bestimmten Text eines Buttons. Die zweite Methode *pressingTheButtonShouldStartTheListActivity* testet, ob eine *ListActivity* beim Klicken eines Buttons gestartet wird. Mit der dritten Methode *shouldHaveALogo* wird getestet, ob ein Logo sichtbar ist und ein bestimmtes Bild referenziert wird.

```
@RunWith(RobolectricTestRunner.class)
public class HomeActivityTest {

    private HomeActivity activity;
    private Button pressMeButton;
    private ImageView pivotalLogo;

    @Before
    public void setUp() throws Exception {
        activity = new HomeActivity();
    }
}
```

<sup>6</sup>Gibt für die meisten Android-Objekte das passende Shadow-Objekt zurück

<sup>7</sup>Ausdruck für das Vortäuschen von echten Objekten durch Platzhalter

```

        activity.onCreate(null);
        pressMeButton = (Button) activity
            .findViewById(R.id.press_me_button);
        pivotalLogo = (ImageView) activity
            .findViewById(R.id.pivotal_logo);
    }

    @Test
    public void shouldHaveAButtonThatSaysPressMe() throws Exception {
        assertThat((String) pressMeButton.getText(),
            equalTo("Tests Rock!"));
    }

    @Test
    public void pressingTheButtonShouldStartTheListActivity() throws
        Exception {
        pressMeButton.performClick();

        ShadowActivity shadowActivity = shadowOf(activity);
        Intent startedIntent = shadowActivity.getNextStartedActivity();
        ShadowIntent shadowIntent = shadowOf(startedIntent);
        assertThat(shadowIntent.getComponent().getClassName(),
            equalTo(NamesActivity.class.getName()));
    }

    @Test
    public void shouldHaveALogo() throws Exception {
        assertThat(pivotalLogo.getVisibility(), equalTo(View.VISIBLE));
        assertThat(shadowOf(pivotalLogo).getResourceId(),
            equalTo(R.drawable.pivotallabs_logo));
    }
}

```

Listing 3.5: Testbeispiel mit *Robolectric* (Angelehnt an [Git13])

Die Annotation `@RunWith(RobolectricTestRunner.class)` sorgt dafür, dass nicht der Testrunner von JUnit aufgerufen wird, um den Test auszuführen, sondern der Testrunner von Robolectric. Auch hier wird die Methode `setUp()` vor den Testmethoden ausgeführt. Die Testmethoden sind mit der Annotation `@Test` gekennzeichnet. Das zu testende Programm wird nicht wie bei Robotium per Klassennamen im Quellcode eingebunden, sondern es werden die Java-Dateien der zu testenden Klassen über das Testprojekt referenziert [Rls13].

Ein Nachteil des Frameworks ist es, dass keine Hardware-spezifischen Tests möglich sind, weil der Zugriff auf echte oder emulierte Geräte nicht vorhanden ist.

### 3.2.7 UiAutomator

Der UiAutomator ist von allen vorgestellten Frameworks das Neueste<sup>8</sup>. Das Framework besteht aus zwei Komponenten, dem *uiautomatorviewer* und dem *uiautomator*. Der *uiautomatorviewer* ist ein grafisches Oberflächentool, das die UI-Komponenten einer Android-Anwendung scannt und analysiert. Der *uiautomator* hält neben einer API, mit der das Schreiben von funktionalen UI-Tests möglich ist, auch noch eine Ausführungsmaschine (*execution engine*) für das Automatisieren und Ausführen von Tests bereit [Goo13h].

Zur Zeit existiert noch keine Eclipse-Unterstützung, so dass alle Schritte wie z. B. das Übersetzen, die Übertragung auf das Gerät und die Ausführung im Moment nur über die Konsole möglich sind. Die Integration in Eclipse ist jedoch nur eine Frage der Zeit.

Dieses Framework unterscheidet sich von den anderen, weil es sich nicht auf Paketgrenzen beschränkt, sondern quer über das ganze Android OS System operiert. Eine der Folgen daraus ist unter anderem, dass keine explizite Verbindung zum Projekt aufgebaut werden muss, da es keinen Quelltext benötigt. Der Testcode kann deswegen leicht wiederverwendet werden. Ausdrucksstarke Methodennamen gestalten das Lesen von UiAutomator-Tests sehr einfach, wie Listing 3.6 zeigt.

```
public class LaunchSettings extends UiAutomatorTestCase {

    public void testDemo() throws UiObjectNotFoundException {

        // Simulate a short press on the HOME button.
        getUiDevice().pressHome();

        // We are now in the home screen. Next, we want to simulate
        // a user bringing up the All Apps screen.
        // If you use the uiautomatorviewer tool to capture a snapshot
        // of the Home screen, notice that the All Apps button's
        // content-description property has the value "Apps". We can
        // use this property to create a UiSelector to find the button.
        UiObject allAppsButton = new UiObject(new UiSelector()
            .description("Apps"));

        // Simulate a click to bring up the All Apps screen.
        allAppsButton.clickAndWaitForNewWindow();
    }
}
```

---

<sup>8</sup>Ende November 2012 von Google veröffentlicht

```
// In the All Apps screen, the Settings app is located in the
// Apps tab. To simulate the user bringing up the Apps tab,
// we create a UiSelector to find a tab with the text
// label "Apps".
UiObject appsTab = new UiObject(new UiSelector().text("Apps"));

// Simulate a click to enter the Apps tab.
appsTab.click();

// Next, in the apps tabs, we can simulate a user swiping until
// they come to the Settings app icon. Since the container view
// is scrollable, we can use a UiScrollable object.
UiScrollable appViews = new UiScrollable(new UiSelector()
    .scrollable(true));

// Set the swiping mode to horizontal (the default is vertical)
appViews.setAsHorizontalList();

// Create a UiSelector to find the Settings app and simulate
// a user click to launch the app.
UiObject settingsApp = appViews.getChildByText(new UiSelector()
    .className(android.widget.TextView.class.getName()),
    "Settings");
settingsApp.clickAndWaitForNewWindow();

// Validate that the package name is the expected one
UiObject settingsValidation = new UiObject(new UiSelector()
    .packageName("com.android.settings"));
assertTrue("Unable to detect Settings", settingsValidation
    .exists());
}
}
```

Listing 3.6: Testbeispiel mit *UiAutomator* [Goo13i]

Die Testklasse muss von der Klasse *UiAutomatorTestCase* abgeleitet werden. Die Methode *testDemo()* testet das Aufrufen der Systemeinstellungen (Settings). Dafür wird zunächst der Home-Button mit der Methode *pressHome()* gedrückt. Mit *UiSelector().description("Apps")* wird der Button mit der Beschreibung "Apps" gesucht, um den Bildschirm mit allen Apps mit der Methode *clickAndWaitForNewWindow()* aufzurufen. Um das Drücken des App-Tabs zu simulieren, wird nach dem Objekt ge-

sucht, dessen Text „Apps“ lautet und mit `click()` aufgerufen. Mithilfe der Methoden `UiSelector().scrollable(true)` und `setAsHorizontalList()` wird ein *UiScrollable*-Objekt erzeugt, das die neue Auswahl in eine Knotenbaumstruktur bringt. Dieses Objekt wird anschließend mit `getChildByText` durchsucht, um das Objekt der App zu finden, das den Klassennamen „Settings“ trägt. Nach der Zuweisung erfolgt der Aufruf wieder mit `clickAndWaitForNewWindow()`. Zum Schluss des Tests fragt die `assertTrue`-Methode noch ab, ob das Objekt mit dem Paketnamen „com.android.settings“ existiert, ansonsten wird ein Failure mit dem Text „Unable to detect Settings“ ausgelöst.

Es gibt jedoch noch eine große Einschränkung in der Benutzung des Frameworks, was gleichzeitig auch den größten Nachteil gegenüber den anderen Frameworks darstellt. Es unterstützt nämlich nur Android-Programme ab Version 4.1 (Jelly Bean) oder höher.

### 3.2.8 SensorSimulator

Der im Jahr 2011 veröffentlichte SensorSimulator ist bereits in der Version 2.0 verfügbar. Obwohl er kein Testframework ist, bietet er die Möglichkeit mobile Parameter zu simulieren. Die Idee des SensorSimulators ist es die Sensordaten künstlich zu erzeugen, anstatt sie vom Betriebssystem anzufordern, die entweder von einem echten Gerät oder von emulierter Hardware (wie z.B. Android Emulator) stammen. Dafür muss die zu testende Android-Anwendung umgeschrieben werden. Der `SensorManager` des Android-Frameworks<sup>9</sup> muss durch den `SensorManager`<sup>10</sup> des SensorSimulators ersetzt werden (siehe Listing 3.7).

```
public class SensorSimulatorDemoActivity extends Activity {
    private SensorManagerSimulator mSensorManager;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        mSensorManager = SensorManagerSimulator
            .getSystemService(this, SENSOR_SERVICE);
        mSensorManager.connectSimulator();
        ...
    }
    ...
}
```

Listing 3.7: Initialisierung des *SensorSimulators* (Quelle: [Sen13])

---

<sup>9</sup>android.hardware.SensorManager

<sup>10</sup>org.openintents.sensorsimulator.hardware.SensorManagerSimulator



Nach der Initialisierung ist die Android-Anwendung bereit, um künstliche Sensordaten des SensorSimulators zu empfangen. Unterstützt werden dabei Basis-Sensoren (*accelerometer*, *magnetic field*, *orientation*), erweiterte Sensoren (*linear acceleration*, *gravity*, *rotation vector*, *gyroscope*), Umgebungssensoren (*temperature*, *light*, *proximity*, *pressure*) sowie andere Sensoren (*barcode reader*). Die Erzeugung übernimmt dafür ein Java-Programm, siehe Abbildung 3.4. Die Kommunikation zwischen der Android- und Java-Anwendung erfolgt über eine Socketverbindung<sup>11</sup>. Erst wenn diese Verbindung steht, können mithilfe einer grafischen Bedienoberfläche die erwünschten Sensoreinstellungen geändert werden. Darüber hinaus existiert auch ein Aufnahmerekorder, der bestimmte Szenarien aufnehmen kann, um sie später einfach abspielen zu lassen. Über eine Telnetverbindung lassen sich auch der Batteriestand und die GPS-Koordinaten manipulieren. Dabei geschehen alle Manipulationen in Echtzeit.

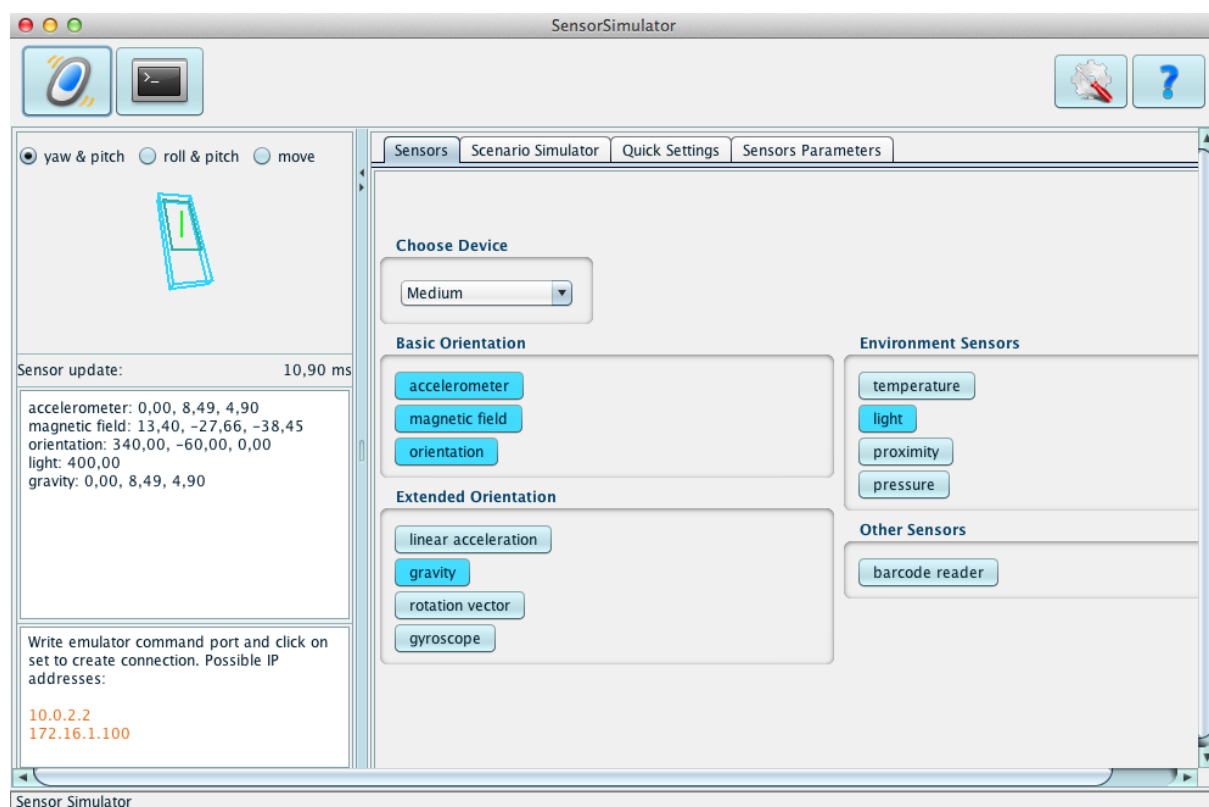


Abbildung 3.4: Grafische Bedienoberfläche des SensorSimulators

Es gibt weitere ähnliche Lösungen wie zum Beispiel der SDK Controller [Sol13] oder der SDK Controller Sensor [Zak13]. Sie verwenden die Sensordaten eines echten Gerätes, um sie dem Android Emulator zur Verfügung zu stellen. Ein per USB angeschlossenes Gerät dient als Quelle, um die Sensordaten in Echtzeit an den Emulator zu übertragen. Die Manipulation findet dann über das Gerät statt.

<sup>11</sup>Port 8010

### 3.3 Ergebnis der Analyse

Mit Robotium und UiAutomator existieren bereits sehr leistungsfähige Testframeworks (siehe Tabelle 3.1), die dem Entwickler High-Level-Methoden an die Hand geben, um umfangreiche und komplexe Tests zu schreiben, die unter anderem die Simulation von Benutzereingaben und System-Events ermöglichen. Das automatisierte Testen ist dank des JUnit-basierten Aufbaus nicht nur problemlos möglich, sondern auch sehr komfortabel.

Testframework	Vorteile	Nachteile
Instrumentation	Komplexe Tests möglich JUnit-Unterstützung	Keine High-Level-API Unterstützt keine System-Events
Positron	Erweiterung von Instrumentation Einfache Bedienung (High-Level-API)	Kein Support seit 2009 Unterstützt keine System-Events
UI/Application Exerciser Monkey	Unterstützt User- und System-Events	Kommandozeilenprogramm Nur Stresstests möglich
monkeyrunner	Übersichtliche API Über Plugin-Schnittstelle erweiterbar	Geringe Funktionalität Tests in Python geschrieben
Robotium	Volle Android 4.2 Unterstützung Mächtige API JUnit-Unterstützung	
Robolectric	Hohe Performanz Kein Emulator oder Gerät notwendig JUnit-Unterstützung	Nutzen von Shadow-Objekten Tests sind nicht authentisch
UiAutomator	Sehr mächtige API Operiert systemweit JUnit-Unterstützung	Unterstützt erst ab Android 4.1 Testen nur über Konsole möglich
SensorSimulator	Simuliert mobile Sensordaten Komfortable Bedienung (GUI)	Hat keinen Testrunner Anpassung des Quellcodes nötig

Tabelle 3.1: Zusammenfassung der bisherigen Lösungen

Obwohl Robotium keine gravierende Nachteile besitzt, bietet es sowie alle anderen analysierten Lösungen keine Möglichkeit automatisierte Testen auf mobile Parameter zu schreiben. Zwar kann auf mobile Parameter getestet werden, jedoch gibt es keine Möglichkeit, diese Parameter während der Testlaufzeit<sup>12</sup> zu verändern. Da die Tests im Labor<sup>13</sup> stattfinden, wo sich die Bedingungen nie oder kaum ändern. Allerdings haben mobile Parameter die natürliche Ei-

<sup>12</sup>Die Zeit, in der das Programm ausgeführt wird

<sup>13</sup>Arbeitsplatz mit der entsprechenden Entwicklungsumgebung

genschaft, sich ständig zu ändern - je nach Standort und Zeitpunkt. Die bisherigen Lösungen können keine mobilen Parameter simulieren<sup>14</sup>, so dass auf diese Parameter nicht automatisiert getestet werden kann.

Die einzige Ausnahme bildet der SensorSimulator<sup>15</sup>. Er ermöglicht, mobile Sensordaten zu emulieren und sie während der Laufzeit zu ändern. Der SensorSimulator ist jedoch kein Testframework und bietet nicht die Möglichkeit, automatisierte Tests zu schreiben.

Im nächsten Kapitel wird daher eine neue Lösung entwickelt, die die geforderte Funktionalität ermöglicht.

---

<sup>14</sup>Einzige Ausnahme bildet das Wechseln des Portrait- und Landscape-Modus

<sup>15</sup>oder ähnliche Lösungen

## 4 Identifikation möglicher neuer Lösungsansätze

In diesem Kapitel wird zunächst erklärt, welche Ansatzmöglichkeiten existieren, um das Simulieren von wechselnden Umgebungsänderungen zu ermöglichen. Danach werden entsprechende Lösungsansätze vorgestellt, von denen der Beste im letzten Unterkapitel ermittelt wird, um ihn im nächsten Kapitel dann umzusetzen.

### 4.1 Möglichkeiten der Simulation wechselnder Umgebungsänderungen

In Unterkapitel 3.3 wurde erläutert, dass automatisierte Tests immer dann nicht möglich sind, wenn die zu testende Software auf wechselnde Bedingungen reagieren muss. Bei einer Zugfahrt beispielsweise ändert sich ständig die Netzwerkverbindung. Eine Software mit einem Datendienst muss demnach ständig darauf reagieren. Um diesen Anwendungsfall im Labor nun automatisiert testen zu können, muss dieses Verhalten simuliert werden. Für die Simulation gibt es grundsätzlich zwei Ansatzmöglichkeiten, die hier kurz erläutert werden.

Im vorherigen Kapitel wurde das Framework *Robolectric* vorgestellt. Dieses nutzt nicht das native Android-System, sondern simuliert nur das Verhalten des Systems, das für den Kontext des jeweiligen Tests notwendig ist. Ein Lösungsansatz sieht demzufolge so aus, dass das entsprechende Verhalten eines Android-Systems so simuliert wird, als ob eine Veränderung der Hardware stattgefunden hätte. Wenn der Batteriestand beispielsweise so verändert wird, dass er von einem unkritischen in einen kritischen Bereich fällt, müsste dann die Simulation die Rolle des nativen Systems übernehmen und alle entsprechenden Events auslösen, die in dieser Situation normalerweise vom nativen System aufgerufen werden.

Der andere Lösungsansatz wäre, dass das native Verhalten von Android erhalten bleibt, dafür aber die entsprechenden Änderungen an der Hardware vorgenommen werden. Dadurch muss das native Android-System selbst auf die Veränderungen der Hardware reagieren.

## 4.2 Lösungsansätze

Es folgen nun konkrete Lösungsansätze, um die geforderte Funktionalität zu gewährleisten. Der erste Ansatz verfolgt das Simulieren eines angepassten Android-Verhaltens, während die anderen zwei versuchen die Hardware zu verändern.

### 4.2.1 Android manipulieren

Unterkapitel 3.2 hat gezeigt, dass das Kontrollieren des Android-Lifecycles und das Auslösen der System- und Benutzerevents nur mit einem Testframework möglich sind. Aus diesem Grund soll ein bestehendes Testframework erweitert werden, um das Android-Verhalten so zu manipulieren, dass es auf imaginäre Hardwareänderungen reagiert.

Es gibt zwei Gründe, weshalb dieser Ansatz ausschließlich zu einer Erweiterung eines bestehenden Testframeworks führt. Erstens wäre es innerhalb dieser Arbeit ein zu großer Aufwand, ein von Grund auf neues Framework zu schreiben. Zweitens funktioniert das Kombinieren von mehreren Testframeworks deswegen nicht, weil jedes Testframework seinen eigenen *Testrunner* und seine eigene Testumgebung benötigt.

Um zu ermitteln welches Framework für die Erweiterung am besten geeignet ist, werden zunächst alle ausgeschlossen, die nicht geeignet sind. Das Instrumentation Framework und das Positron Framework sind nicht geeignet, weil beide das Auslösen von Systemevents nicht unterstützen. Der UI/Application Exerciser Monkey ist auch ungeeignet, weil mit ihm nur Stresstests möglich sind und dies eine zu hohe Einschränkung seitens des Entwicklers darstellt.

Nach dem Ausschluss bleiben nur noch monkeyrunner, Robotium, Robolectric und UiAutomator übrig. Zwei dieser Frameworks haben zudem noch besondere Nachteile. Der monkeyrunner ist nämlich in der Programmiersprache Python geschrieben und Robolectric läuft nur auf der JVM.

Alle diese Frameworks haben einen offenen Quellcode, damit sie mithilfe der Community weiterentwickelt werden können. Java ist die Entwicklungsprogrammiersprache im Android-Kontext, weil das entsprechende Framework in Java geschrieben ist. Um von einer möglichst großen Community zu profitieren, sollte auch die Erweiterung in Java geschrieben werden. Es ist ein Alleinstellungsmerkmal des Robolectric-Frameworks, dass Android-Tests in der JVM laufen können. In der Praxis laufen demnach alle anderen Tests, die nicht mit Robolectric geschrieben wurden, auf dem nativen Gerät oder im Android Emulator. Die Anzahl der Tests, die für das native Android geschrieben wurde, ist wesentlich höher als die für das „gemockte“

Android. Der potenzielle Aufwand des Umschreibens der Tests wäre demnach deutlich größer, wenn die Erweiterung kein natives Android unterstützen würde.

Mit der Vorgabe, dass eine mögliche Erweiterung in Java implementiert werden soll und die späteren Testanwendungen auf einem echten Gerät bzw. im Emulator laufen sollen, wären somit die Frameworks Robotium und UiAutomator am besten geeignet. Sie besitzen eine ausreichend mächtige API, mit der es möglich ist, das gewünschte Verhalten zu simulieren.

#### 4.2.2 Geräte manipulieren

Der im ersten Moment intuitivste Ansatz ist die Manipulation von echten Geräten selbst. Schließlich kommen ja auch in den manuellen Feldtests echte Geräte zum Einsatz. Natürlich hat dieser Ansatz auch einen unschlagbaren Vorteil gegenüber allen anderen Ansätzen, denn hier braucht nichts simuliert oder emuliert zu werden, da hier mit den echten Geräten gearbeitet wird. Während beim Emulator so gut wie jede Gerätekonfiguration getestet werden kann, beschränkt sich dieser Ansatz auf ein konkretes Gerät. Dies ist jedoch kein Nachteil, weil beim Testen auf mobile Parameter lediglich das Verhalten der Software überprüft wird und nicht das UI-Design. Das Testgerät benötigt hierfür nur die erforderlichen Eigenschaften, die für die volle Unterstützung der Software notwendig sind.

Es gibt jedoch keine Möglichkeit, echte Geräte zu manipulieren, weil die Manipulation von echten Geräten von den Herstellern nicht vorgesehen ist und daher auch von deren Seite nicht unterstützt wird. Android selbst bietet auch nur wenige Manipulationsmöglichkeiten, die im Testfall unterstützt werden können. Wenn auf dem Gerät der Entwicklermodus aktiviert ist, können beispielsweise „gefakte“ GPS-Koordinaten gesetzt werden. Die Manipulation in dem Grad, wie es hier erforderlich wäre, ist jedoch nicht möglich. Aus technischer Sicht gäbe es mit *Android Rooting*<sup>1</sup> eventuell Mittel und Wege das Gerät zu „knacken“ (auch wenn es dabei zur Erlöschung der Herstellergarantie käme), aber es gibt auch noch eine rechtliche Einschränkung, die die Manipulation von echten Android-Geräten verbietet. In den USA müssen *Software Defined Radio*<sup>2</sup>-Geräte von der *Federal Communications Commission* (FCC) zertifiziert werden. Teil der Zertifizierung beinhaltet die Vorgabe, dass es dem Endbenutzer des Gerätes nicht gestattet ist, die Arbeitsweise der Funkeinheit oder deren benutzte Frequenzen zu verändern [Yag13, S. 157]. Die Manipulation des GSM-Moduls bei einem echten Gerät ist demnach verboten. Da es in Deutschland ähnliche Vorgaben gibt, kann die Manipulation nicht über die Geräte erfolgen. Damit ist dieser Ansatz nicht realisierbar.

---

<sup>1</sup>Erlaubt dem Endbenutzer den Zugriff über das Android Betriebssystem mittels Administratorenrechte

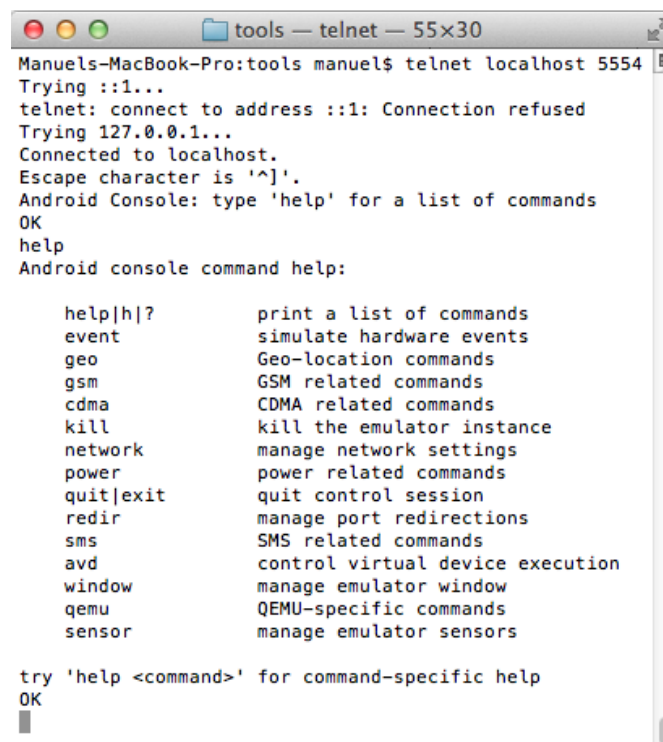
<sup>2</sup>Das Bestreben die gesamte Signalverarbeitung in Software abzubilden

### 4.2.3 Emulator manipulieren

Dass der Android Emulator verschiedene Einstellungen eines mobilen Endgeräts simulieren kann, wurde bereits in Abschnitt 2.4.1 festgestellt. Um nun die wechselnden Bedingungen der Realität im Labor nachzustellen, müssen sich die Einstellungen des Emulators zur Laufzeit der Software verändern. Der Emulator bietet eine Schnittstelle zur Änderung seiner Einstellungen über Telnet an. Das Aufbauen einer Telnetverbindung kann über einen Konsolenaufruf geschehen (siehe Abbildung 4.1). Hier der passende Befehlsaufruf:

```
$ telnet localhost 5554
```

Dabei entspricht `localhost` dem lokalen Rechner und `5554` der Portnummer der Android-Instanz. Eine Instanz kann entweder ein Androidgerät sein, das per USB an den lokalen Rechner angeschlossen ist oder eine Instanz des Android Emulators. Dabei hat jede Instanz seine eigene Portnummer.



```
Manuels-MacBook-Pro:tools manuel$ telnet localhost 5554
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
help
Android console command help:

      help|h|?      print a list of commands
      event         simulate hardware events
      geo           Geo-location commands
      gsm           GSM related commands
      cdma          CDMA related commands
      kill          kill the emulator instance
      network       manage network settings
      power         power related commands
      quit|exit     quit control session
      redir         manage port redirections
      sms           SMS related commands
      avd           control virtual device execution
      window        manage emulator window
      qemu          QEMU-specific commands
      sensor        manage emulator sensors

try 'help <command>' for command-specific help
OK
```

Abbildung 4.1: Telnetschnittstelle des Android Emulators

Mithilfe dieser Schnittstelle sind fast alle Parameter setzbar bzw. veränderbar. Tabelle 4.1 auf Seite 37 gibt eine Übersicht über die Befehle, die die Manipulation von mobilen Parametern ermöglichen.

Befehl	Subbefehl	Parameter
geo	nmea	<sentence>
	fix	<longitude> <latitude> [<altitude> [<satellites>]]
gsm	list	
	call   cancel	<phonenumber>
	busy   hold   accept	<remoteNumber>
	data   voice	<state>
	status	
	signal	<rssi> [<ber>]
cdma	ssource	<source>
	prl_version	<version>
	status	
	speed	<speed>
	delay	<delay>
	capture	start   stop
network	status	
	speed	gsm   hscsd   gprs   edge   umts   hsdpa   full
	delay	<latency>
	capture	start   stop
power	display	
	ac	on   off
	status	unknown   chaging   discharging   not-charging   full
	present	true   false
	health	unknown   good   overheat   dead   overvoltage   failure
	capacity	<percentage>
sensor	status	
	get	<sensorname>
	set	<sensorname> <value-a>[:<value-b>[:<value-c>]]

Tabelle 4.1: Übersicht über Manipulationsbefehle des Android Emulators

Durch das Setzen der einzelnen Parameter können so die unterschiedlichsten Bedingungen im Labor simuliert werden.



### 4.3 Ermittlung des besten Lösungsansatzes

Nachdem nun mehrere Lösungsansätze vorgestellt wurden, muss an dieser Stelle die Entscheidung getroffen werden, welcher Ansatz im nächsten Kapitel implementiert werden soll. Neben der Lösung des Hauptproblems sind die wichtigsten Kriterien dabei Flexibilität, Komfort und Effizienz der Lösung. Um den besten Lösungsansatz zu ermitteln, wird deshalb insbesondere darauf geachtet, wie groß der Anwendungsradius ist, wie viele Entwickler von ihr profitieren, wie flexibel sie eingesetzt werden kann und wie komfortabel bzw. einfach Tests geschrieben bzw. erweitert werden können und wie hoch der Aufwand ihrer Umsetzung ist.

Um nun eine Entscheidung zu treffen, fasst Tabelle 4.2 die jeweiligen Ansätze unter Berücksichtigung dieser Auswahlkriterien noch einmal zusammen.

Lösungsansatz	Vorteil	Nachteil
Android manipulieren	Hoher Testkomfort	Frameworkerweiterung notwendig Auf ein Framework beschränkt
Gerät manipulieren	Realitätstreue Tests	Rechtlich nicht erlaubt
Emulator manipulieren	Effiziente Umsetzung Unterstützt alle Android-Versionen Extrem hoher Testkomfort Ergänzt andere Frameworks	Neues Framework notwendig Auf Emulator beschränkt

Tabelle 4.2: Zusammenfassung der Lösungsansätze

Um Android zu manipulieren (Ansatz aus Abschnitt 4.2.1), müsste ein Framework wie Robolectric erweitert werden. Es müsste mit entsprechenden Eigenschaften ergänzt bzw. weiterentwickelt werden. Ein Framework auf diese Art zu erweitern, ist mit einem recht hohen Aufwand verbunden. Der mögliche Komfort des Entwicklers wäre jedoch extrem hoch, weil das neue Feature ein Teil des Frameworks wäre. Die Flexibilität wäre jedoch eingeschränkt, da das neue Feature nur in diesem einen Framework nutzbar wäre. Tests von anderen Frameworks müssten umgeschrieben werden.

In Abschnitt 4.2.2 wurde bereits erläutert, dass die Manipulation von echten Geräten aus rechtlichen Gründen nicht zulässig ist. Dieser Ansatz fällt also weg, weil mit ihm das Kernproblem nicht gelöst werden kann.

Die Manipulation des Emulators (Ansatz aus Abschnitt 4.2.3) bietet gleich mehrere Vorteile gegenüber den anderen Ansätzen. Der Android Emulator bietet eine standardisierte Schnittstelle, die die Manipulation von vielen Parametern erlaubt. Eine mögliche Lösung wäre sehr

effizient umsetzbar. Zusätzlich unterstützt dieser Ansatz sämtliche Android-Versionen und ist nicht von einem Testframework abhängig, sondern einzig und allein vom Emulator. Das Testen wäre dann allerdings auf den Emulator beschränkt. Andererseits ermöglicht dieser Ansatz eine Ergänzung zu bereits bestehenden Testframeworks, wie es in Abbildung 4.2 angedeutet ist<sup>3</sup>. Dadurch ergibt sich eine maximale Flexibilität, weil sämtlicher geschriebener Testcode so bleiben kann wie er ist. Mögliche Umschreibungen wie im ersten Ansatz wären nicht notwendig. Die einzige Ausnahme bilden die Ergänzungen von möglichen Manipulationsanweisungen, die die Änderung der Emulatoreinstellungen bewirken sollen.

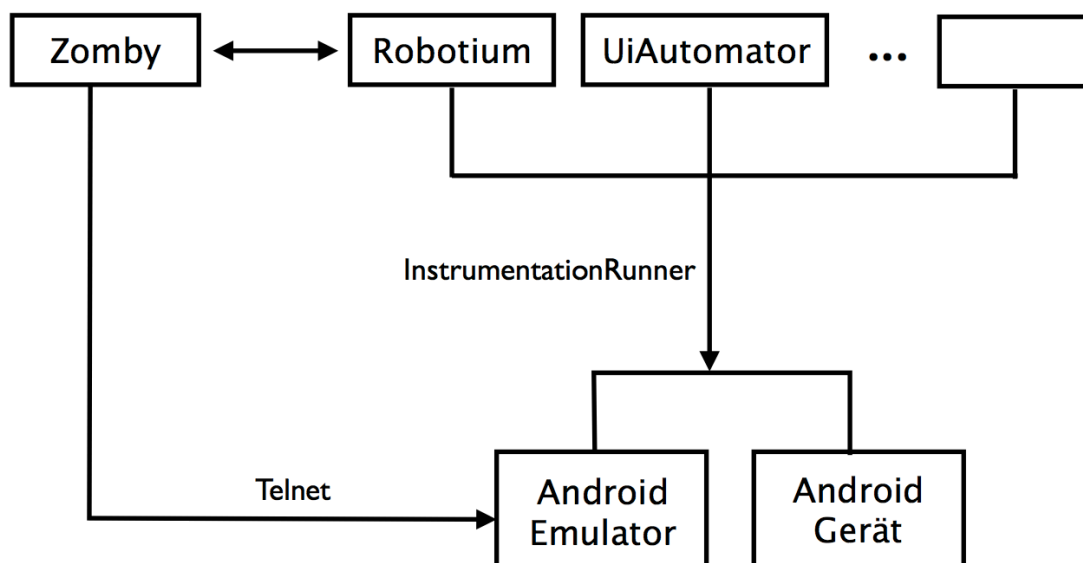


Abbildung 4.2: Möglicher Ansatz ergänzt bestehende Testframeworks

Nachdem nun die Lösungsansätze unter Berücksichtigung wichtiger Auswahlkriterien betrachtet wurden, fällt die Wahl recht einfach aus. Aufgrund einer effizienten Umsetzung und seiner hohen Flexibilität stellt der Ansatz, den Emulator zu manipulieren, beinahe eine Ideallösung dar. Daher folgt nun im Anschluss die Implementierung des Lösungsansatzes aus Abschnitt 4.2.3.

<sup>3</sup>Der Name des neuen Frameworks lautet Zomby

## 5 Implementierung der neuen Lösung

In diesem Kapitel folgt nun die Implementierung des Lösungsansatzes aus Abschnitt 4.2.3. Dafür wird ein eigenes Framework namens *Zomby* geschrieben, das den Emulator innerhalb des Testprogramms manipulieren soll. Der Testentwickler kann dann innerhalb des Testcodes zusätzlich zu den normalen Testmethoden auch noch Anweisungen geben, wie z. B. "*setze Batterie auf 88 Prozent*", "*setze Netzwerkgeschwindigkeit auf EDGE*" oder "*setze GPS-Koordinaten*". Damit sind präzise Anwendungsfälle kontrolliert und automatisiert testbar.

Neben der gerade beschriebenen Grey-Box-Testvariante, bei der der Testentwickler den Quellcode kennt, existiert auch noch eine Black-Box-Testvariante. Die könnte so aussehen, dass ein von außen gesteuertes Programm zufällige Parameter ändert, worauf das zu testende Programm reagieren muss. Damit wären Stresstests möglich, um die Robustheit des Programms zu testen, indem es sich unter den veränderlichen Bedingungen konform verhält und nicht abstürzt<sup>1</sup>.

Dieses Konzept ermöglicht dem Entwickler zwar die maximale Kontrolle, führt aber bei der Implementierung zu gewissen Problemen. Welche dies genau sind, wird im nächsten Unterkapitel erläutert.

### 5.1 Problemanalyse

Wird der bestehende Testcode nun mit Anweisungen zur Emulatormanipulation erweitert, ergibt sich dadurch ein erweiterter Testprozessablauf, wie Abbildung 5.1 zeigt.

Der normale Testprozess wird nicht verändert, sondern lediglich erweitert. Die Testanwendung muss zunächst pausiert werden, damit die Manipulationsanweisungen des Entwicklers ausgeführt werden können. Nachdem die Emulatoreinstellungen sich dementsprechend geändert haben, wird die Testanwendung wieder fortgesetzt. Dies führt zu drei Problemfeldern, die nachfolgend erläutert werden.

---

<sup>1</sup>Diese Variante wird hier aus mangelnder Praxisrelevanz nicht implementiert

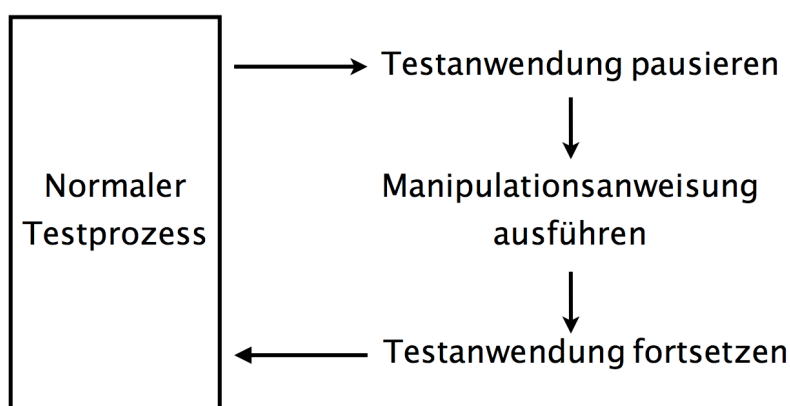


Abbildung 5.1: Erweiterung des Testprozessablaufs durch zusätzliche Manipulationsanweisungen

### 5.1.1 Synchronisierung

Der Testprozess muss an den Stellen unterbrochen werden, an denen der Testentwickler die Einstellungen des Emulators ändern möchte. Beispielsweise möchte er einen Synchronisierungsprozess testen, indem er zuerst die Netzwerkverbindung auf HSDPA setzt. Nachdem der Prozess gestartet wird und anfängt Daten zu übertragen, gibt der Testentwickler die Anweisung „*schalte die Netzwerkeinstellung aus*“, um ein Funkloch zu simulieren. Erst nach der Manipulation soll die Testanwendung dann wieder an der unterbrochenen Stelle weiterlaufen. Damit ist gewährleistet, dass die nachfolgenden Testanweisungen unter den veränderten Bedingungen getestet werden können. Ansonsten könnten die nachfolgenden `assert`-Methoden unter Umständen zu unterschiedlichen Ergebnissen kommen. Für die Simulation ist es also entscheidend, dass der Emulator während der Testlaufzeit manipuliert wird.

Das Testprogramm muss ein Zeichen bekommen, wenn die Manipulationsänderungen erfolgreich durchgeführt wurden, ansonsten weiß es nicht, wann es wieder fortsetzen kann. Das Problem besteht also nicht beim Stoppen, sondern beim Fortsetzen.

### 5.1.2 Manipulation

Für die Ausführung der Manipulationsanweisung muss geklärt werden, auf welche Weise dies geschehen kann. Die einzige Schnittstelle, die der Android Emulator bietet, um dessen Einstellungen zu ändern, ist über das Netzwerkprotokoll Telnet (siehe Abschnitt 2.4.1). Aus diesem Grund wird das Testumfeld noch einmal genauer untersucht. Abbildung 5.2 zeigt einen Überblick über den schemenhaften Aufbau der Testumgebung.

Das Testprogramm instrumentalisiert das zu testende Programm innerhalb der Dalvik VM. Die Dalvik VM wird wiederum innerhalb des Emulators ausgeführt und dieser vom Entwicklungs-

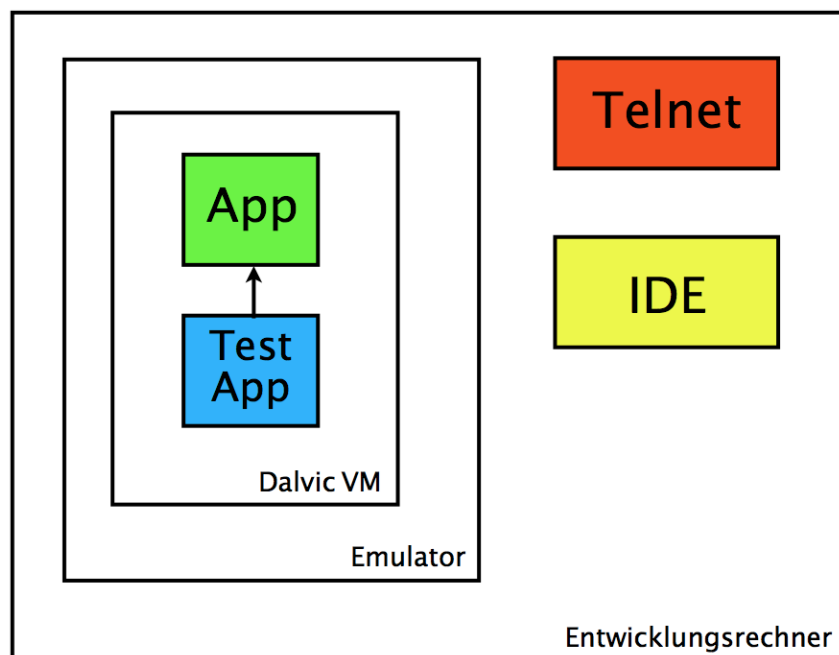


Abbildung 5.2: Schemenhafter Aufbau der Testumgebung

rechner. Daraus ergeben sich theoretisch mehrere Wege, den Emulator zu manipulieren.

Die Manipulation von Innen wäre sicherlich die komfortabelste Lösung, weil so keine Kommunikation mit dem lokalen Rechner notwendig wäre. Jedoch werden die Apps aus Sicherheitsgründen in einer Sandbox<sup>2</sup> ausgeführt. Zugriffe außerhalb der Sandbox sind somit nicht möglich. Abbildung 5.3 zeigt die Fehlermeldung des LogCat-Protokolls beim Versuch eine Telnetverbindung zum Emulator innerhalb der Dalvik VM aufzubauen.

```

ActivityMana... Displayed com.example.android.notepad/.NotesList: +95ms
NetworkManag... setKernelCountSet(10012, 0) failed with errno -2
ActivityMana... START {act=android.intent.action.INSERT dat=content://com.example.notepad.provider.NotePad/
notes cmp=com.example.android.notepad/.NoteEditor} from pid 2655
WindowManager Failure taking screenshot for (230x383) to layer 21015
PhoneStatusBar setLightsOn(true)
ActivityMana... Displayed com.example.android.notepad/.NoteEditor: +129ms
dalvikvm GC_CONCURRENT freed 393K, 10% free 7057K/7815K, paused 0ms+0ms
dalvikvm GC_FOR_ALLOC freed 241K, 10% free 7059K/7815K, paused 3ms
System.err Exception while connecting:socket failed: EACCES (Permission denied)
dalvikvm GC_CONCURRENT freed 239K, 5% free 6621K/6919K, paused 1ms+0ms
dalvikvm threadid=10: still suspended after undo (sc=1 dc=1)
dalvikvm GC_CONCURRENT freed 436K, 8% free 7072K/7623K, paused 0ms+0ms
Zygote Process 2655 exited cleanly (1)
WindowManager WIN DEATH: Window{b69e7638 com.example.android.notepad/com.example.android.notepad.NoteEdit
or paused=false}
ActivityMana... Process com.example.android.notepad (pid 2655) has died.
WindowManager WIN DEATH: Window{b69bccd0 com.example.android.notepad/com.example.android.notepad.NotesLis
t paused=false}

```

Abbildung 5.3: Telnetverbindungsversuch zum Emulator innerhalb der Dalvik VM (LogCat-Ausschnitt)

Genau so kann der Android Emulator aus sich selbst heraus nicht manipuliert werden. Daher bleibt nur noch die Manipulation von Außen übrig. Dies erfolgt, wie in Abbildung 4.1 bereits

<sup>2</sup>Isolierter Bereich, in dem jede Maßnahme keinerlei Auswirkung auf die äußere Umgebung hat

gesehen, mithilfe der Kommandozeile, indem von Außen (außerhalb des Android Emulators) über den lokalen Rechner eine Telnetverbindung aufgebaut wird. Durch das Absetzen entsprechender Telnetbefehle kann so der Emulator manipuliert werden.

Nachdem nun festgestellt wurde, dass die Manipulation nur von Außen erfolgen kann, wird zunächst eine Komponente benötigt, die die Anweisungen von der Testanweisung entgegennimmt, dann die Telnetverbindung zum Emulator aufbaut sowie die entsprechenden Befehle absetzt und zum Schluss die erfolgreiche Durchführung protokolliert. Danach muss sie den Prozess anstoßen, der die Testanwendung veranlasst weiterzulaufen.

### 5.1.3 Kommunikation

Es muss eine geeignete Schnittstelle gefunden werden, um innerhalb der Testanwendung mit dem lokalen Rechner zu kommunizieren. Abbildung 5.4 zeigt das entsprechende Ablaufdiagramm.

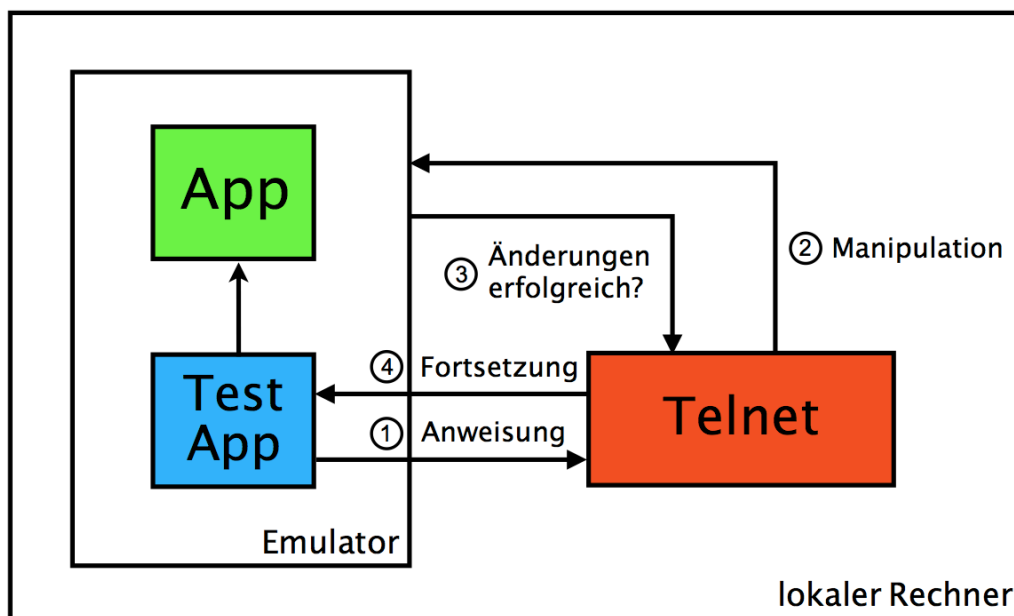


Abbildung 5.4: Aktivitätsdiagramm des erweiterten Testprozesses

- ① Testprogramm übergibt Manipulationsanweisung an den lokalen Rechner
- ② Anweisung wird über Telnet abgesetzt, um den Emulator zu manipulieren
- ③ Überprüfung, ob die Änderungen erfolgreich durchgeführt wurden
- ④ Wenn alles gut gegangen ist, wird das Testprogramm fortgesetzt

Die Kommunikation muss in zwei Richtungen erfolgen. Einerseits von der Testanwendung hin zum lokalen Rechner, um die Manipulationsanweisungen dem Telnetclient zu übergeben und andererseits vom lokalen Rechner wieder zurück zur Testanwendung, um bei erfolgreicher Manipulation der Testanwendung ein Zeichen zum Weiterlaufen zu geben. Die Einschränkungen

der Dalvik VM und des Android Emulators machen dies zu einer Herausforderung, und zwar insbesondere die Kommunikation von Außen nach Innen.

## 5.2 Lösungsweg

In diesem Unterkapitel werden sämtliche Probleme gelöst, die in der Problemanalyse aufgetreten sind. Die folgenden Abschnitte behandeln jeweils einen Problemschwerpunkt aus der Analyse. Dabei wird fließend ein Lösungsweg beschrieben, der im nächsten Unterkapitel implementiert wird.

### 5.2.1 Prozesssynchronisierung

In Java gibt es viele Konzepte, um Prozesse zu synchronisieren. Die Synchronisierung zwischen den Testanweisungen und den Manipulationsanweisungen fällt unter die Kategorie „*Warten und Benachrichtigen*“. Für diese Fälle enthalten alle Java-Objekte seit der Version 1.0 die Methoden `wait()` und `notify()`. Der erste Schritt ist es, die Testprozesse nach der Manipulationsanweisung auf eine Semaphore<sup>3</sup> warten zu lassen. Der Lösungsansatz dafür ist denkbar einfach, wie Listing 5.1 zeigt.

```
synchronized (Zomby.getSemaphore()) {
    try {
        Zomby.getPower().setUncriticalPower();
        Log.d(TAG, "TestApp is stopped!");
        Zomby.getSemaphore().wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
Log.d(TAG, "TestApp is continued!");
```

Listing 5.1: Beispiel eines synchronen Manipulationsaufrufs

Innerhalb eines *synchronized*-Blocks wird die Manipulationsanweisung ausgeführt, um anschließend den Testprozess mit Hilfe der Methode `wait()` warten zu lassen. Die Synchronisierung erfolgt immer über dasselbe Java-Objekt, worauf mit der statischen Methode `getSemaphore()` zugegriffen wird. Dieses muss auch später für den synchronisierten Aufruf von `notify()` bzw. `notifyAll()` verwendet werden.

<sup>3</sup>Eine Semaphore ist eine Datenstruktur, die das Verwaltung beschränkter Ressourcen ermöglicht, auf die mehrere Prozesse oder Threads zugreifen wollen

Der zweite Schritt die Testprozesse wieder aufzuwecken, ist hingegen etwas schwieriger. Dafür ist ein zweiter Prozess notwendig. Dieser muss zudem auch noch von Außen gestartet werden, weil die Telnetanweisungen vom lokalen Rechner aus abgesetzt werden. Das Android Framework bietet einen Mechanismus an, um dies zu gewährleisten. Damit eine laufende Android-Anwendung auf bestimmte Ereignisse reagieren kann, registriert es der sogenannte *Broadcast Receiver*. Dieser meldet sich sobald das überwachende Ereignis eingetroffen ist. In diesem Fall tritt das Ereignis auf, wenn die Emulatoreinstellungen geändert wurden. Wenn dies geschieht, sollen alle Testanwendungen wieder geweckt werden. Listing 5.2 zeigt die entsprechende Implementierung.

```
public class EmulatorChangeReceiver extends BroadcastReceiver {
    private final String TAG = "Zomby Administration";

    @Override
    public void onReceive(Context arg0, Intent arg1) {
        Log.d(TAG, "Receiver is started");
        synchronized (Zomby.getSemaphore())
        {
            Zomby.getSemaphore().notifyAll();
        }
    }
}
```

Listing 5.2: Aufruf des BroadcastReceivers weckt alle schlafenden Testprozesse auf

Listing 5.3 zeigt abschließend noch das Registrieren bzw. das Abmelden des Broadcast Receivers anhand eines Robotium-Tests.

```
public void testAddNote() throws Exception {
    solo.clickOnMenuItem("Add note");

    // Broadcast receiver registered
    Zomby.registeredEmulatorChangeReceiver(context);
    // set critical battery level
    Zomby.getPower().setCriticalPower();
    //Broadcast receiver unregistered
    Zomby.unregisterEmulatorChangeReceiver(context);

    solo.assertCurrentActivity("Expected NoteEditor activity", "
        NoteEditor");
    solo.enterText(0, "Note 1");
}
```



```
        solo.goBack();  
    }
```

Listing 5.3: Erweiterung eines Robotiumtests mit Manipulationsanweisungen

## 5.2.2 Kommunikation durch Intents

Das Aufrufen von Broadcast Receivern mithilfe des lokalen Rechners, wie im vorherigen Abschnitt gefordert, ermöglicht das Android SDK Tool *Android Debug Bridge* (siehe Abschnitt 2.4.3). Damit können *Intents* auf der Konsole ausgelöst werden. Hier der passende Aufruf:

```
$ adb shell am broadcast -a de.inovex.android.EMULATOR_IS_CHANGED
```

`EMULATOR_IS_CHANGED` ist das entsprechende Intent, das den *EmulatorChangeReceiver* auslöst. Damit wurde nicht nur eine Möglichkeit gefunden, wie die schlafenden Testprozesse wieder aufgeweckt werden können, sondern auch ein Weg, wie der lokale Rechner mit dem Testprogramm kommunizieren kann.

## 5.2.3 Manipulationskomponente

Um den Emulator auf vorgegebene Weise zu manipulieren, muss die Manipulationskomponente folgende Aufgaben erfüllen:

1. Manipulationsanweisungen empfangen
2. Telnetverbindung zum Emulator aufbauen
3. Die entsprechenden Befehle ausführen
4. Durchführung protokollieren
5. Weck-Prozess anstoßen

Da die Komponente gleich mehrere Aufgaben bewältigen muss, gestaltet sich die Lösungsfindung etwas komplizierter. Dafür führen aber gleich mehrere Wege zum Ziel. Die Erfolgversprechendsten werden nun kurz vorgestellt, um im Anschluss den besten zu ermitteln.

### Aufbau der Telnetverbindung erfolgt über Script

Die Idee hinter diesem Lösungsansatz ist es, dass das Android-Testprogramm ein Script auf dem lokalen Rechner ausführt, um so die entsprechenden Telnetbefehle abzusetzen. Mithilfe

von Tools wie Expect<sup>4</sup>, die das Eingeben von Telnetparametern automatisieren können, bietet dieser Ansatz eine komfortable Lösung für die ersten drei Aufgabenpunkte an. Ein Nachteil dieser Lösung ist jedoch die einseitige Kommunikation. Das Testprogramm führt unter Android ein Java-Execute-Befehl „in the dark“ aus, um das Script zu starten. Das bedeutet, der Befehl gibt kein Feedback, ob das Ausführen des Scripts erfolgreich war oder ob ein Fehler aufgetreten ist. Dieser Nachteil kann jedoch umgangen werden, indem das Ergebnis der Ausführung durch den Intent des Broadcast Receivers an das Testprogramm weitergegeben wird. Sollte das Script nicht aufgerufen werden, der Prozess blockieren oder ähnliche Probleme auftreten, könnte nach einem Time-Out eine Fehlermeldung aufgerufen werden.

Ansonsten, wenn Telnet nach der Ausführung eines Befehls ein „OK“ zurückgibt, war die Manipulation des Emulators erfolgreich und der Weck-Prozess kann angestoßen werden. Dafür muss lediglich ein Konsolenbefehl ausgeführt werden, wie es in Abschnitt 5.2.2 beschrieben ist. Der letzte Punkt ist also wieder problemlos möglich.

### Daemon liest LogCat der TestApp aus

Ein weiterer Ansatz ist ein Daemon<sup>5</sup>, der den LogCat der TestApp ausliest. Der Prozessablauf ist in Abbildung 5.5 dargestellt.

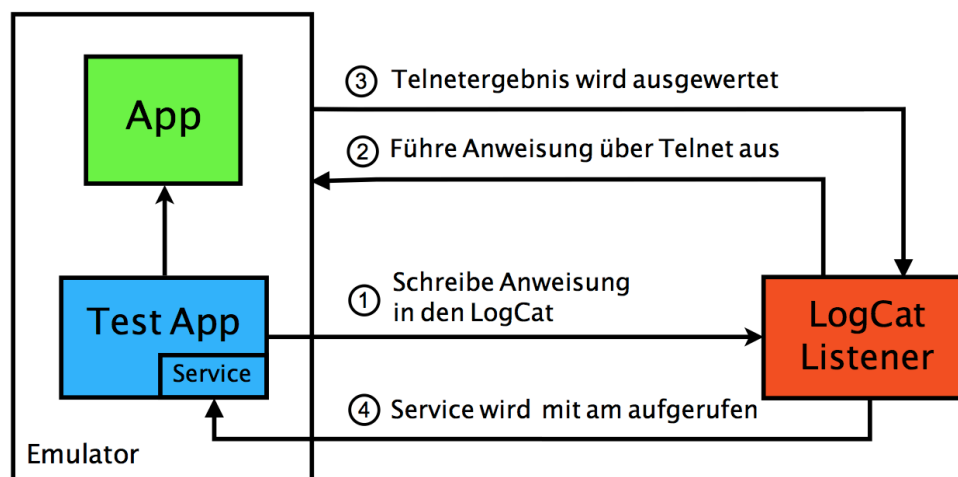


Abbildung 5.5: Prozessablauf: LogCat auslesen

- ① Testprogramm schreibt Manipulationsanweisung in den Log
- ② Daemon liest LogCat aus und setzt Manipulationsanweisungen über Telnet ab, nachdem sie erkannt wurden
- ③ Telnetergebnis wird ausgewertet
- ④ Ist alles gutgegangen, wird der Broadcast Receiver aufgerufen, um das Testprogramm fortzusetzen

<sup>4</sup><http://www.nist.gov/el/msid/expect.cfm>

<sup>5</sup>Ein Hintergrundprozess, der bestimmte Dienste anbietet

Auf diese Weise kann der Daemon (LogCat-Listener) die als Manipulationsanweisung gelabelten Log-Nachrichten lesen und darauf hin die entsprechende Anweisung ausführen. Auch hier treten wieder die gleichen Kommunikationsprobleme (einseitige Kommunikation) auf, wie bei der Lösung zuvor.

Neben dem Auslesen und Analysieren bestimmter Muster, die die Manipulationsanweisung erkennbar machen sollen, muss zusätzlich ein Weg gefunden werden, der eine erneute Ausführung der bereits abgesetzten Anweisungsbefehle verhindert, die sich immer noch im LogCat befinden. Dieser Ansatz stellt damit auch technisch eine größere Herausforderung dar.

### Aufbau der Telnetverbindung über lokalen Webdienst

Eine weitere Möglichkeit bietet der Zugriff über einen lokalen Webservice. Eine Android-Anwendung bzw. eine Testanwendung hat die Möglichkeit, auf einen lokalen Webserver zuzugreifen (siehe Abbildung 5.6).

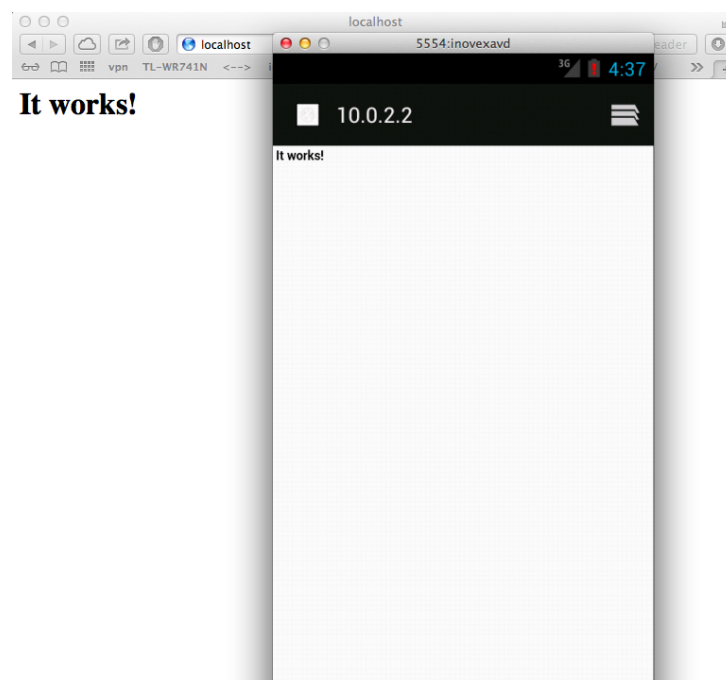


Abbildung 5.6: Zugriff mit Android-Browser auf lokalen Webserver

Dank dem Emulator Networking entspricht die Netzwerkadresse 10.0.2.2 des Emulators durch *Portforwarding* (Portweiterleitung) der, des `localhosts`<sup>6</sup> vom lokalen Rechner (siehe Abbildung 5.7) [Goo13b].

Im Gegensatz zu den zwei Ansätzen zuvor bietet dieser durch *Request*<sup>7</sup> und *Response*<sup>8</sup> ei-

<sup>6</sup>Netzwerkadresse 127.0.0.1

<sup>7</sup>Anfrage vom Client an den Server

<sup>8</sup>Antwort des Servers auf eine Anforderung eines Clients

Network Address	Description
10.0.2.1	Router/gateway address
10.0.2.2	Special alias to your host loopback interface (i.e., 127.0.0.1 on your development machine)
10.0.2.3	First DNS server
10.0.2.4 / 10.0.2.5 / 10.0.2.6	Optional second, third and fourth DNS server (if any)
10.0.2.15	The emulated device's own network/ethernet interface
127.0.0.1	The emulated device's own loopback interface

Abbildung 5.7: Netzwerkadressraum des Android Emulators (Bildquelle [Goo13b])

ne komfortable Kommunikation in zwei Richtungen an. Bisher ist dies der einzige Ansatz, bei dem eine synchrone Verarbeitung möglich ist, weil hier der Umweg über den Broadcast Receiver entfällt. Der Austausch zwischen der Testanwendung und dem lokalen Rechner kann über denselben Weg erfolgen. Ein weiterer Vorteil ist die Standardisierung der Webschnittstelle. Sie ermöglicht einen plattformübergreifenden Einsatz. Der einzige Nachteil ist, dass jeweils eine zusätzliche Internet-Permission (Benutzerberechtigung) für die Android- und die Testanwendung notwendig wäre.

Der Aufbau und die Ausführung der Telnetverbindung sind dabei Teil des Webdienstes. Die Auswertung erfolgt dabei innerhalb des Requests und liefert einen entsprechenden Response mit dessen Ergebnis zurück. Wegen der synchronen Verarbeitung entfällt das Wecken der Testanwendung.

Obwohl nun drei unterschiedliche Varianten vorgestellt wurden, um die geforderten Aufgaben zu bewältigen, fällt die Wahl dennoch relativ leicht aus. Aufgrund der einfacheren Kommunikation und des universelleren Einsatzes hat sich die letzte Variante ganz klar als die beste und sauberste Lösung herausgestellt. Sie bietet zudem auch potenziell bessere Weiterentwicklungsmöglichkeiten, weil Java sehr viele Klassen für die Webschnittstelle zur Verfügung stellt.

### 5.3 Entwicklung eines neuen Frameworks

In diesem Unterkapitel folgt nun die Entwicklung des neuen Zomby-Frameworks. Für die Implementierung des Frameworks wurden im Laufe der vorigen Kapitel zahlreiche Entscheidungen getroffen, welche die Handhabung und die technische Umsetzung des Frameworks vorgeben. Daraus ergibt sich folgender Prozess, der in Abbildung 5.8 in Form eines Aktivitätsdiagramms dargestellt ist.

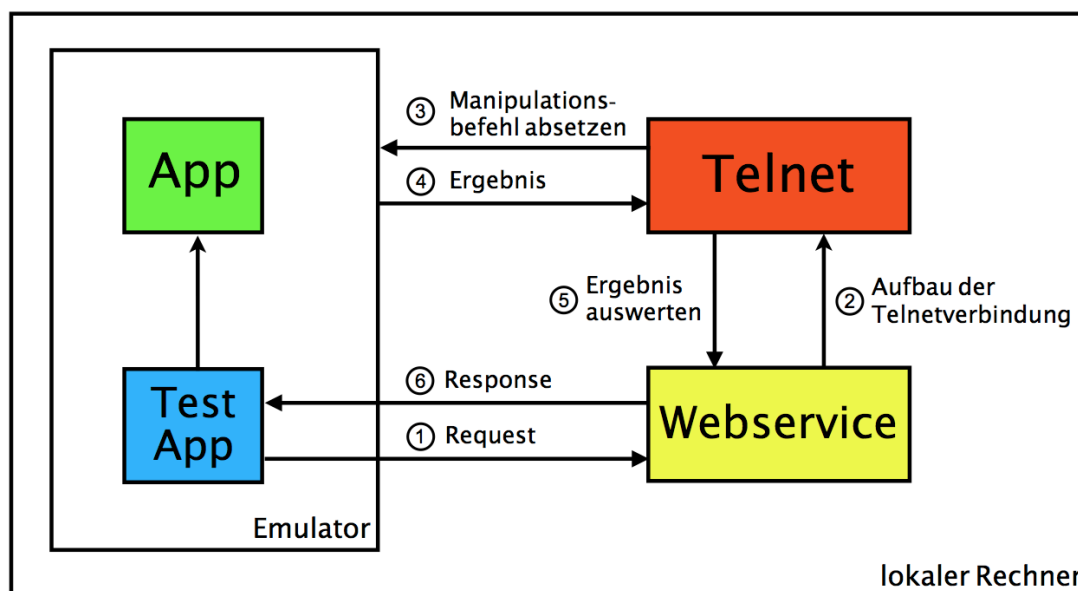


Abbildung 5.8: Aktivitätsdiagramm des Zomby-Frameworks

- ① Testanwendung (Client) übergibt die Manipulationsanweisung mithilfe eines Requests an den Webservice (Server)
- ② Webservice baut eine Telnetverbindung auf
- ③ Telnetkomponente führt die Anweisung aus
- ④ Telnetkomponente bekommt Telnetergebnis zurück
- ⑤ Webservice wertet das Ergebnis aus und setzt entsprechenden Response
- ⑥ Testanwendung läuft anhand des Responses entweder weiter oder gibt einen Fehler aus

Damit der Entwickler seine Manipulationsanweisungen innerhalb der Testanwendung machen kann, muss eine clientseitige API entwickelt werden. Um den Wartungsaufwand des Frameworks so gering wie möglich zu halten, wird daher nicht noch zusätzlich eine weitere serverseitige API entwickelt. Dies ist auch nicht nötig, wenn der Webservice den kompletten, auszuführenden Telnetbefehl von der clientseitigen API mithilfe des Requests bekommt. Die Befehle brauchen dann nur noch eins-zu-eins vom Webservice ausgeführt und die Ergebnisse des Telnetbefehls per Response an das Testprogramm zurückgegeben zu werden. Die Fehlerauswertung kann dann ebenfalls über die clientseitige API erfolgen (siehe Abschnitt 5.3.4). Dadurch benötigt die API neben den Methoden mit den passenden Telnetbefehlen nur noch eine Komponente, die den Webzugriff bereitstellt.

Bevor das neue Framework implementiert werden kann, muss zunächst noch auf die Architektur der API eingegangen werden.

### 5.3.1 Architektur

In Unterkapitel 3.2 wurden die Vor- und Nachteile von Low- und High-Level APIs bereits aufgezeigt. Um tiefgehende und flexible Tests zu schreiben, benötigt der Entwickler eine Low-Level API. Deren Befehle spielen sich auf der untersten Ebene ab. Dem Entwickler steht der maximale Freiheitsgrad zur Verfügung. Dennoch kommt er nur mühevoll voran, weil er immer nur kleine Schritte machen kann. Das Pendant dazu ist die High-Level API. Deren Befehle haben eine hohe Abstraktionsebene, mit denen ein Entwickler schnell vorankommt. Der Nachteil ist dass, dadurch die Flexibilität leidet.

In diesem Fall ist eine grundsätzliche Entscheidung jedoch nicht notwendig, weil hier beide APIs unterstützt werden können.

#### Low-Level API

In Tabelle 4.1 auf Seite 37 sind nur Telnetbefehle aufgelistet, die Werte der mobilen Parameter setzen können. Sie enthält jedoch nicht alle zur Verfügung stehenden Befehle. Weitere wichtige Befehle, die ebenfalls Teil der Low-Level API sein sollen, folgen ergänzend in Tabelle 5.1.

Befehl	Subbefehl	Parameter
event	send	<type>:<code>:<value>
	types   codes	<type>
	text	<message>
sms	send	<phonenumber> <text message>
	pdu	<hexstring>
avd	start   stop   status   name   snapshot	

Tabelle 5.1: Übersicht über Manipulationsbefehle des Android Emulators (Ergänzung)

#### High-Level API

Mithilfe der Low-Level API kann jetzt eine High-Level API aufgebaut werden. Aus `capacity(3)`, `capacity(30)` und `capacity(100)` werden beispielsweise `criticalPower()`, `uncriticalPower()` und `fullPower()`. Der Testcode wird damit lesbarer. Die High-Level API bietet Befehle an, die Anwendungsfall-orientiert sind. Natürlich sind es nicht nur Kapselungen oder Kombinationen von Low-Level-Methoden,

die möglich sind, sondern auch komplexe Anwendungsfälle, beispielsweise die Simulation eines gehenden Fußgängers oder eines fahrenden Autos. Diese API kann später auch von Entwicklern erweitert werden, so dass das Framework zukünftig immer mehr Anwendungsfälle unterstützen kann.

### 5.3.2 Implementierung des Webdienstes

Für die Implementierung des Webdienstes wurde ein passendes Servlet<sup>9</sup> geschrieben, das auf einem beliebigen Webserver (Servlet-Container) laufen kann, wie beispielsweise Tomcat<sup>10</sup> oder GlassFish<sup>11</sup>. Es übernimmt die Entgegennahme der Manipulationsanweisungen über ein Request und führt diese über Telnet aus. Danach gibt es das Ergebnis per Response zurück.

Der Webserver sollte lokal auf dem Entwicklungsrechner laufen, um einerseits eine schnellstmögliche Bearbeitung der Abfragen und andererseits den Zugriff zum Emulator zu ermöglichen. Da die Verarbeitung mit dem Webserver ausschließlich für das Zomby-Framework vorgesehen ist, wird das Servlet nur Post-Requests unterstützen. Die Servlet-Klasse besteht daher nur aus einer überschriebenen `doPost`-Methode, wie in Listing 5.4 zu sehen ist. Die Verarbeitung über die URL mithilfe eines Get-Requests ist damit ausgeschlossen.

```
@WebServlet(name = "ZombyServlet", urlPatterns = {"/ZombyServlet"})
public class ZombyServlet extends HttpServlet {
    ...
    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        ...
    }
    ...
}
```

Listing 5.4: Aufbau der Klasse ZombyServlet

### Aufbau der Telnetverbindung

Um die Klassen für die Telnetkomponente des Webdienstes nicht doppelt zu entwickeln, wird hier die *Google Android Tools ddmlib r10 API* verwendet [GAT13]. Diese Java-Bibliothek enthält

<sup>9</sup>Java-Klassen-Instanz, die innerhalb eines Webserver Anfragen von Clients entgegennimmt und beantwortet

<sup>10</sup><http://tomcat.apache.org>

<sup>11</sup><https://glassfish.java.net>

Klassen, die den Aufbau, die Absetzung und die weitere Verarbeitung einer Telnetverbindung zu einem Android-Gerät bzw. einer Emulatorinstanz ermöglichen. Listing 5.5 zeigt zunächst den Verbindungsaufbau einer adb-Verbindung zu einer Android-Instanz.

```
if(bridge == null) {
    AndroidDebugBridge.init(false);
    bridge = AndroidDebugBridge.createBridge(ADB_PATH, true);
    waitForDevices();
}
// select first device by default
if (bridge.getDevices().length == 0)
    throw new Exception("There are no attached devices");
currentDevice = bridge.getDevices()[0];
//initial a emulator console
EmulatorConsole eCon = EmulatorConsole.getConsole(currentDevice);
```

Listing 5.5: Aufbau einer Telnetverbindung

Die Klasse `AndroidDebugBridge` ist Teil der API und initialisiert mit der statischen Methode `init` alle notwendigen Variablen, um danach mithilfe der Fabrikmethode `createBridge` eine passende Bridge-Instanz zu erhalten. Die Hilfsmethode `waitForDevices` gehört nicht zur API und wartet lediglich eine gewisse Zeit ab, bis die Liste aller in Frage kommenden Instanzen zurückgegeben wird. Sollte die Erstellung der Liste zu lange dauern oder sie keine Geräte enthalten, endet die Verarbeitung mit einem Fehler (Exception). Andernfalls wird das erste Gerät aus der Liste ausgewählt und in die Variable `currentDevice` gespeichert. Diese dient für die Methode `getConsole` der Klasse `EmulatorConsole` als Übergabeparameter. Auch diese ist eine Fabrikmethode, die eine passende `EmulatorConsole`-Instanz zurückgibt. Mithilfe dieser Instanz können nun Telnetbefehle ausgeführt werden.

### Ausführung und Auswertung der Telnetverbindung

Listing 5.6 zeigt die Ausführung und die Auswertung der Telnetverbindung. Auch sie ist Teil der `doPost`-Methode.

```
// init response
response.setContentType("text/html;charset=UTF-8");
PrintWriter out = response.getWriter();
String telnetAnswer = "";

// execute telnet commands
```



```
String[] telnetCommands = request.getParameterValues("telnet");
for (String telnetCommand : telnetCommands) {
    telnetAnswer = eCon.processCommand(telnetCommand);
    try {
        // set response
        if(telnetAnswer != EmulatorConsole.RESULT_OK)
            out.println(telnetAnswer);
        else
            out.println("TELNET_COMMAND_WAS_SUCCESSFUL");
    } finally {
        out.close();
    }
}
```

Listing 5.6: Ausführung und Auswertung einer Telnetverbindung

Es werden Variablen für den späteren Response instanziiert. Die übergebenen Telnetbefehle aus dem Request werden in das Feld `telnetCommands` gespeichert, über welches danach eine for-Schleife iteriert, um die entsprechenden Befehle über die Methode `processCommand` abzusetzen. Diese Methode führt einen Telnetbefehl aus und gibt als Rückgabewert das Ergebnis der Ausführung zurück.

War die Ausführung erfolgreich, wird der String `TELNET_COMMAND_WAS_SUCCESSFUL` in den Response geschrieben, andernfalls der entsprechende Fehlertext (siehe Abschnitt 5.3.4).

Mit dieser einen `doPost`-Methode unterstützt der Webdienst alle vorgegebenen Aufgaben. Der Webdienst bildet die Basis für die clientseitige Programmierschnittstelle, die im nachfolgenden Abschnitt implementiert wird.

### 5.3.3 Implementierung der Programmierschnittstelle

Es folgt die Implementierung der Programmierschnittstelle (API), die die Telnetbefehle zur Manipulation des Emulators bereithält und sie an den Webdienst aus Abschnitt 5.3.2 übergibt.

#### Webservice

Wenn das Zomby-Framework innerhalb eines Android-Testprogramms aufgerufen wird, erfolgen die Manipulationsanfragen des Frameworks an den Webdienst auch innerhalb des Android-Testprogramms. Die Webservice-Komponente des Frameworks muss demzufolge mithilfe von Android-Methoden implementiert werden, damit die Http-Anfragen im Testprogramm

aufgerufen werden können. Listing 5.7 zeigt eine *WebService*-Methode, die einen Telnetbefehl per Post-Request mithilfe von Android-Methoden an den Webdienst sendet. Die Klasse *WebService* repräsentiert die Webservice-Komponente des Zomby-Frameworks.

```
private void sendHttpPostRequest(String telnetCommand) {
    // Create a new HttpClient and Post Header
    HttpClient httpClient = new DefaultHttpClient();
    HttpPost httpPost = new HttpPost(HOST+TELNET_SERVLET);
    HttpResponse response = null;

    List<NameValuePair> nameValuePairs =
        new ArrayList<NameValuePair>(1);
    nameValuePairs.add(new BasicNameValuePair("telnet",
        telnetCommand));

    // Execute HTTP Post Request
    try {
        httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
        response = httpClient.execute(httpPost);
        telnetAnswer = EntityUtils.toString(response.getEntity());
    } catch (Exception e) {
        e.printStackTrace()
    }
}
```

Listing 5.7: Implementierung eines Post-Requests mit Android-Methoden

Der Aufruf der Http-Anfrage erfolgt über die Klasse *DefaultHttpClient*. An den Konstruktor der Klasse *HttpPost* wird die URL des Webdienstes (*ZombyServlet*) übergeben. Der Telnetbefehl wird mithilfe einer Liste an die *HttpPost*-Instanz übergeben und diese mit der *execute*-Methode ausgeführt. Das Ergebnis wird als String in die Variable *telnetAnswer* für die weitere Fehlerbehandlung gespeichert.

## Low-Level-Klassen

Für die Implementierung der Low-Level API werden ausschließlich Klassen geschrieben, deren Namen der jeweiligen Telnetbefehle entsprechen. Ergänzt wird der Name nur noch mit dem Präfix "Core", um zu symbolisieren, dass es sich um eine Low-Level-Klasse handelt. Die Namen und Eingabewerte der Klassenmethoden entsprechen denen der Subbefehle. Listing 5.8 zeigt mit der Klasse *CoreSMS* beispielhaft, wie alle Klassen der Low-Level API aufgebaut sind.

```
public class CoreSMS {

    private static final String TAG = "Zomby Instrumentation";
    private String telnetCommand = "";
    private WebService webservice = new WebService();

    /**
     * allows you to simulate a new inbound sms message
     * @param phonenumber
     * @param text
     * @throws ZombyException
     */
    public void send(String phonenumber, String message) throws
        ZombyException {
        telnetCommand = "sms send " + phonenumber + " " + message;
        ZombyLog.logTelnetCommand(TAG, telnetCommand);
        webservice.sendTelnetCommand(telnetCommand);
    }

    /**
     * allows you to simulate a new inbound sms PDU<br>
     * (used internally when one emulator sends SMS messages to
     * another instance)<br>
     * you probably don't want to play with this at all
     * @param hexstring
     * @throws ZombyException
     */
    public void pdu(String hexstring) throws ZombyException {
        telnetCommand = "sms send " + hexstring;
        ZombyLog.logTelnetCommand(TAG, telnetCommand);
        webservice.sendTelnetCommand(telnetCommand);
    }
}
```

Listing 5.8: Aufbau einer Low-Level-Klasse

Der Telnetbefehl *sms* enthält zwei Subbefehle, *send* und *pdu*. Daraus ergibt sich die Klasse *CoreSMS* mit den Methoden *send(String phonenumber, String message)* und *pdu(String hexstring)*. Die Übergabeparameter ergeben sich aus den Übergabewerten der Subbefehle.

Der Telnetbefehl mit den jeweils übergebenen Parametern wird in die Variable `telnetCommand` geschrieben und mithilfe der *WebService-Methode* `sendTelnetCommand` an den Webdienst geschickt.

## High-Level-Klassen

Die Klassennamen der High-Level API entsprechen auch den Namen der Telnetbefehle, jedoch ohne den Präfix "Core". Anders als bei der Low-Level API orientieren sich die Methodennamen der High-Level API an den Anwendungsfällen. Auch hier wird in Listing 5.9 mit der Klasse *Power* beispielhaft gezeigt, wie die restlichen High-Level-Klassen aufgebaut sind.

```
public class Power {

    private static final String TAG = "Zomby Instrumentation";

    /**
     * allows you to set battery capacity in a critical area
     */
    public void setCriticalPower() throws ZombyException {
        ZombyLog.logMessage(TAG, "set critical power");
        Zomby.getCorePower().capacity(3);
    }

    /**
     * allows you to set battery capacity in an uncritical area
     */
    public void setUncriticalPower() throws ZombyException {
        ZombyLog.logMessage(TAG, "set uncritical power");
        Zomby.getCorePower().capacity(30);
    }

    /**
     * allows you to set battery capacity on 50 percent
     */
    public void setHalfPower() throws ZombyException {
        ZombyLog.logMessage(TAG, "set half power");
        Zomby.getCorePower().capacity(50);
    }

    /**
```

```

    * allows you to set battery capacity on 100 percent
    */
public void setFullPower() throws ZombieException {
    ZombieLog.logMessage(TAG, "set full power");
    Zombie.getCorePower().capacity(100);
}

/**
 * allows you to simulate a whole battery life in <msec>
 */
public void simulateWholeBatteryLife(final int msec) throws
    ZombieException {
    simulateBatteryLife(100, 0, msec);
}

/**
 * allows you to simulate the battery life from <start> to <end>
 * in <msec>
 */
public void simulateBatteryLife(final int start, final int end,
    final int msec) throws ZombieException {
    new Thread(new Runnable() {
        @Override
        public void run() {
            for(int battery=start; battery>=end; battery--) {
                try {
                    synchronized(this) {
                        Zombie.getCorePower().capacity(battery);
                        wait(msec/100);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();
}
}

```

Listing 5.9: Aufbau einer High-Level-Klasse

In dieser Klasse sind beide Arten von Methoden vertreten, die mit der High-Level API verfolgt werden. Einerseits Fassadenmethoden, wie `setCriticalPower()`, die den Testcode lesbarer machen und dem Entwickler die Verwendung von Low-Level-Methoden mit entsprechenden Übergabewerten ersparen. Andererseits Methoden wie `simulateWholeBatteryLife(msec)`, die in eigenen Threads laufen müssen, weil sie mobile Eigenschaften simulieren, die einen zeitlichen Verlauf haben. In diesem Fall ein Batterie-verlauf, der von anfangs voll in einer vorgegebenen Zeit (msec) auf komplett leer sinkt.

## Zomby-Klasse

Eine besondere Klasse bildet die *Zomby*-Klasse. Sie enthält statische Methoden, die die Benutzung des Frameworks für den Entwickler erleichtern sollen. Statt für jede der zahlreichen Klassen, die das Framework bietet, eine eigene Instanz zu bilden, werden Fabrikmethoden verwendet. Dies macht den Testcode lesbarer, weil sich so die Zomby-Methoden gegenüber dem restlichen Testcode klar absetzen können, wie in Listing 5.10 zu sehen.

```
public void testNote() throws Exception {
    solo.clickOnMenuItem("Add note");
    solo.enterText(0, "Note 1");

    Zomby.getPower().simulateWholeBatteryLife(25000);
    Zomby.getCoreNetwork().speed(NetworkSpeed.UMTS);
    Zomby.getCoreNetwork().delay(Latency.EDGE);
    Zomby.getCoreGeo().fix(4.5, 3.3, 2.2);

    solo.goBack();
    solo.clickOnMenuItem("Add note");
    solo.enterText(0, "Note 2");
    solo.goBackToActivity("NotesList");
    solo.takeScreenshot();
}
```

Listing 5.10: Verwendung des Zomby-Frameworks mithilfe der Klasse *Zomby*

Dieses Beispiel zeigt auch noch eine weitere Designentscheidung des Frameworks. Beim Robotium-Testframework stehen alle Methoden des Frameworks in einer einzigen Klasse (*Solo*) zur Verfügung. Beim Zomby-Framework wurde für jeden Telnetbefehl eine eigene Klasse geschrieben. So lässt sich der Kontext jeder Methode einfacher ablesen, wie in diesem Beispiel *Power*, *Network* und *Geo*. Durch den Präfix „Core“ kann zudem auch noch zwischen

Low- und High-Level-Klassen unterschieden werden.

### 5.3.4 Fehler- und Ausnahmebehandlung

Um die Ausnahmebehandlung nicht unnötig kompliziert zu machen, wird innerhalb des Zomby-Frameworks nur eine Exception geworfen. Die *ZombyException* tritt immer dann auf, wenn Fehler bei der Verwendung mit dem Webservice auftreten. Bei der Telnetverarbeitung kann das ein ungültiger Telnetbefehl, falscher Subbefehl oder eine falsche Parametereingabe sein. Der bei der Manipulation des Emulators aufgetretene Fehler wird an die Exception weitergegeben, so dass der Entwickler den Fehler eindeutig identifizieren kann. Abbildung 5.9 zeigt einen JUnit-Test, der wegen einer *ZombyException* fehlschlägt, weil ein falscher Telnet-Subbefehl eingegeben wurde. Sollte der Webservice nicht erreichbar sein, weil er beispielsweise nicht gestartet wurde, wird ebenfalls eine *ZombyException* geworfen.

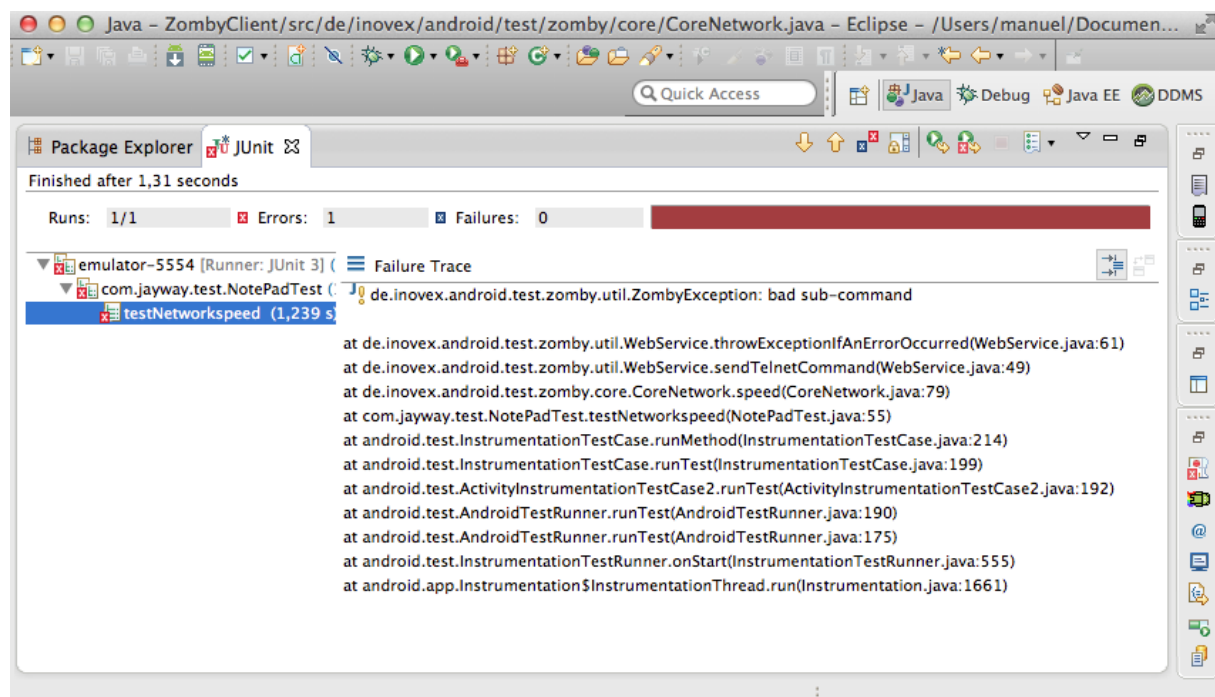


Abbildung 5.9: *ZombyException*: JUnit-Test schlägt wegen falscher Telneteingeabe fehl

## 5.4 Implementierung einer Rekorder-App

Um das Schreiben der Tests zu vereinfachen und die Tests gleichzeitig realistischer zu gestalten, folgt hier die Implementierung einer Aufzeichnungs-App (*Zomby Recorder*). Sie zeichnet Sensordaten eines echten Gerätes auf, damit die Daten anschließend als Eingabeparameter des Zomby-Frameworks dienen können.

Für die Implementierung des Rekorders wurden nur Sensoren berücksichtigt, die von dem Framework auch unterstützt werden. In Abbildung 5.10, auf der die Startoberfläche der Anwendung zu sehen ist, sind die entsprechenden Sensoren mithilfe von Auswahlboxen aufgelistet. Der Anwender hat damit die Wahl, welche Sensordaten er aufzeichnen möchte und welche nicht.

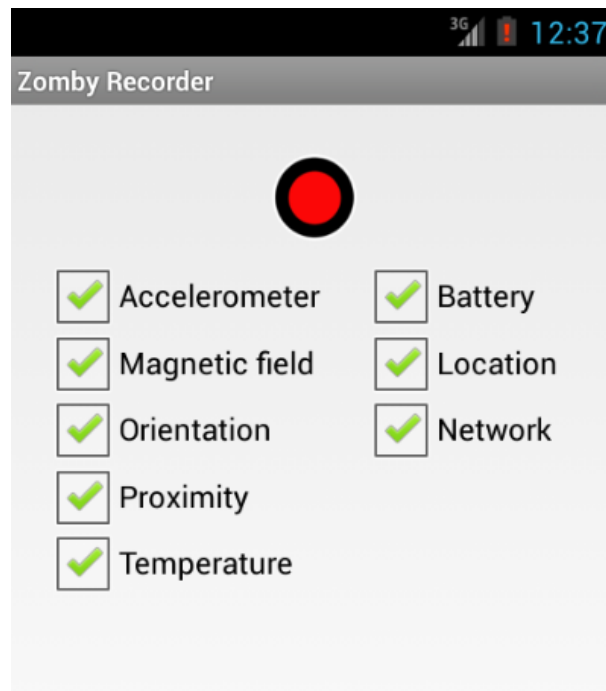


Abbildung 5.10: Ausschnitt des Startbildschirms der Zomby Recorder App

Sobald die Aufnahme gestartet wird, indem der rote Aufnahmeknopf gedrückt wurde, werden alle Daten in einer Liste gespeichert. Um die verschiedenen Parameter referieren zu können, hat jeder Sensor sein eigenes Label, das zusätzlich zu den reinen Nutzdaten mitaufgezeichnet wird. Jeder Listeneintrag bekommt einen Zeitstempel, um somit den exakten Abstand zwischen den einzelnen Datenerhebungen zu ermitteln. Dies ist notwendig, um die Aufnahme in Echtzeit zu simulieren. Beim Beenden der Aufnahme wird die Liste abschließend in eine Datei geschrieben. Diese Dateien haben den Postfix ".zf" (Zomby File).

Wie der Inhalt einer solchen Datei aussehen kann, ist in Listing 5.11 zu sehen.

```
1368533020639;TEMPERATURE;21.0
1368533020790;NETWORK;CONNECTED
1368533020790;NETWORK;UMTS
1368533020890;ORIENTATION;282.0,-39.0,-1.0
1368533026950;LOCATION;7.19509789,50.78706178
1368533027925;LOCATION;7.19472321,50.78618038
1368533030487;ACCELEROMETER;-1.2258313,6.8237944,7.4639506
```



```
1368533030494;BATTERY;37,CHARGING
1368533030593;PROXIMITY;9.0
1368533030674;MAGNETIC_FIELD;31.0625,-27.9375,-36.8125
1368533030782;ACCELEROMETER;-0.3405087,6.4015636,7.3549876
1368533030787;ORIENTATION;283.0,-41.0,-2.0
```

Listing 5.11: Inhalt einer Zomby Recorder-Datei

Jede Zeile entspricht einer Datenerhebung und besteht aus drei Elementen: Zeitstempel<sup>12</sup>, Sensorlabel und Sensordaten. Diese Hauptelemente sind durch ein Semikolon von einander getrennt. Falls ein Sensor mehr als einen Parameter benötigt, wie z. B. der Lokalisierungssensor oder der Orientierungssensor, werden dessen Parameter durch ein Komma getrennt. Diese Konvention muss strikt eingehalten werden, damit die Daten später wieder richtig eingelesen werden können.

Für diese Aufgabe wird das Framework um die Methode `playMyRecord` erweitert. Sie sorgt dafür, dass die entsprechenden Aufzeichnungsdaten in passende Manipulationsbefehle umgewandelt werden und die zeitliche Abfolge eingehalten wird. Die Methode erwartet einen `InputStream` der Aufzeichnungsdatei, einen Zeitfaktor (optional) und eine Liste der Sensoren (optional), die simuliert werden sollen. Der Zeitfaktor dient dazu, die Echtzeit zu umgehen, um die Tests schneller und effizienter zu gestalten. Mit der Liste kann der Entwickler vorgeben, welche Sensordaten bei dem Test berücksichtigt werden sollen und welche nicht. Dies soll flexiblere und zielgerichtete Tests ermöglichen. Listing 5.12 zeigt so ein Beispiel.

```
public void testTemperatureAndNetwork() throws Exception {
    InputStream input = getInstrumentation().getContext()
        .getAssets().open("myrecord.zf");
    List<DataElement> list = new ArrayList<DataElement>();
    list.add(DataElement.TEMPERATURE);
    list.add(DataElement.NETWORK);
    Zomby.getZombyPlayer().playMyRecord(input, list, 0.01);

    ...
}
```

Listing 5.12: Zomby: Aufgezeichnete Sensordaten wieder abspielen

In diesem Beispiel werden die Daten von der Datei *myrecord.zf* eingelesen und nur die Sensoren für Temperatur und Netzwerk in hundertfacher Beschleunigung simuliert, während alle anderen ignoriert werden.

<sup>12</sup>Wird dadurch berechnet, wie viele Millisekunden seit dem 01.01.1970 vergangen sind

Für Testframeworks bei denen es nicht möglich ist eine Datei einzulesen, wie es beim UiAutomator-Framework der Fall ist, generiert der ZombyRecorder neben der .zf-Datei noch eine weitere .java-Datei. Diese enthält eine fertige Testmethode mit allen aufgezeichneten Werten mit den passenden Zomby-Methoden (siehe Listing 5.13).

```
public void testMyRecord() throws Exception {
    Zomby.getCoreSensor().set(Sensorname.TEMPERATURE, 21.0);
    waitForRealtime(151);
    Zomby.getCoreNetwork().speed(NetworkSpeed.valueOf("CONNECTED"));
    Zomby.getCoreNetwork().speed(NetworkSpeed.valueOf("UMTS"));
    waitForRealtime(100);
    Zomby.getCoreSensor().set(Sensorname.ORIENTATION,
        282.0, -39.0, -1.0);
    waitForRealtime(6060);
    Zomby.getCoreGeo().fix(7.19509789, 50.78706178);
    waitForRealtime(975);
    Zomby.getCoreGeo().fix(7.19472321, 50.78618038);
    waitForRealtime(2562);
    Zomby.getCoreSensor().set(Sensorname.ACCELERATION,
        -1.2258313, 6.8237944, 7.4639506);
    waitForRealtime(7);
    Zomby.getCorePower().capacity(37);
    Zomby.getCorePower().status(BatteryState.CHARGING);
    waitForRealtime(99);
    Zomby.getCoreSensor().set(Sensorname.PROXIMITY, 9.0);
    waitForRealtime(79);
    Zomby.getCoreSensor().set(Sensorname.MAGNETIC_FIELD,
        31.0625, -27.9375, -36.8125);
    waitForRealtime(108);
    Zomby.getCoreSensor().set(Sensorname.ACCELERATION,
        -0.3405087, 6.4015636, 7.3549876);
    waitForRealtime(5);
    Zomby.getCoreSensor().set(Sensorname.ORIENTATION,
        283.0, -41.0, -2.0);
}

private void waitForRealtime(long time) throws InterruptedException
{
    double realtimeFactor = 1.0;
    long timeToWait = Math.round(time*realtimeFactor);
    synchronized(this) {
```

```
        if (timeToWait > 10)
            wait (timeToWait);
    }
}
```

Listing 5.13: Zomby-Recorder: Generierte Testmethode mit aufgezeichneten Sensordaten

Es bildet dieselbe Aufzeichnung aus Listing 5.11 ab. Um auch hier die Echtzeit zu bewahren, werden zusätzlich `waitForRealtime`-Methoden eingefügt. Innerhalb dieser Methode kann mithilfe der Variable `realtimeFactor` die Echtzeitgeschwindigkeit gesteuert werden, in der der Test ablaufen soll. Der Standardwert ist 1.0 und entspricht der Echtzeit.

Auf diese Weise ist es auch für *UiAutomator* möglich, einen aufgezeichneten Anwendungsfall mit Zomby zu simulieren. Dadurch bietet es dem Entwickler eine zusätzliche Flexibilität, einen Zomby-Test zu bearbeiten. Der Nachteil ist jedoch der längere Testcode, der unter Umständen die Lesbarkeit stark einschränken kann.

## 6 Evaluierung des Zomby-Frameworks

In dieser Arbeit sollte eine Lösung für das automatisierte Testen von mobiler Software durch Simulation wechselnder Umgebungsänderungen im Android-Kontext entwickelt werden. In diesem Kapitel wird nun evaluiert, wie gut das in dieser Arbeit entwickelte *Zomby Framework* (inklusive *Zomby Recorder*) diese Anforderung erfüllt.

Im nächsten Unterkapitel wurden dafür authentische und praxisnahe Tests geschrieben, die relevante und häufige Anwendungsfälle abdecken, um die Simulation wechselnder Umgebungsänderungen zu veranschaulichen. Zudem wird erklärt, welche technischen Grenzen dieser Lösung unterlegen sind. Danach folgt ein Fehlerbericht, der im Moment noch auftretende Fehler auflistet, die die Nutzung des Frameworks beeinträchtigen. Am Ende des Kapitels werden noch einmal die wesentlichsten Punkte zusammengefasst.

### 6.1 Relevante Anwendungsfälle jetzt automatisiert testbar

In diesem Unterkapitel werden verschiedene Anwendungsfälle beschrieben, die normalerweise einen Feldtest erfordern oder nur manuell getestet werden können. Danach folgt der entsprechende Zomby-Testcode, mit dem dieser Anwendungsfall im Labor automatisch simuliert werden kann.

#### 6.1.1 Batteriezyklus

Manchmal ist es wichtig, dass eine Software auf den aktuellen Batteriestand reagiert, um kritische Prozesse nicht im falschen Moment zu starten oder um die Bedienbarkeit des Gerätes nicht unnötig einzuschränken. In diesem Anwendungsfall fällt der Batteriestand von einem unkritischen in einen kritischen Bereich. Zomby bietet dafür passende Methoden an, wie in Listing 6.1 zu sehen.

```
@Override
public void setUp() throws Exception {
    ...
}
```

```
// set battery power on uncritical
Zomby.getPower().setUncriticalPower();
}

public void testPower() throws Exception {
    ...
    // set battery power on critical
    Zomby.getPower().setCriticalPower();
    ...
}
```

Listing 6.1: Zomby-Test: Automatisierung der Batteriewerte

In der Init-Methode `setUp` wird am Anfang des Tests die Batterie auf einen unkritischen Wert gesetzt, um später im laufenden Test auf einen kritischen Wert zu fallen. Danach kann getestet werden, ob die Software auf die neue Bedingung reagiert.

Neben dem Setzen von bestimmten Batteriestati, ermöglichen zudem die Methoden `simulateBatteryLife` und `simulateWholeBatteryLife` die Simulation eines Batterieverlaufs. Während `simulateWholeBatteryLife` einen kompletten Batteriezyklus von 100% bis 1% simuliert, kann mit `simulateBatteryLife` ein kleinerer Bereich angegeben werden. Die Dauer des jeweiligen Verlaufs gibt ein zusätzlicher Parameter in *ms* an. In Listing 6.2 wird ein Batterieverlauf von 30% bis 3% innerhalb von fünf Sekunden simuliert.

```
public void testPower() throws Exception {
    // simulate battery progress
    Zomby.getPower().simulateBatteryLife(30, 3, 5000);
    ...
}
```

Listing 6.2: Zomby-Test: Automatisierung eines Batterieverlaufs

### 6.1.2 Schwankende Datenverbindung mit plötzlichem Funkloch

Aufgrund des heterogen aufgebauten Handynetzes wechseln die Stärke und Geschwindigkeit der Datenverbindung zum Teil erheblich. Das Testen der Netzwerkverbindung bei mobilen Anwendungen mit Datenanbindung ist daher ein relevanter Testfall. Der Entwickler muss darauf achten, dass die Software auf alle möglichen Bedingungen reagieren kann, ansonsten kann es zum Absturz derselben führen.

In diesem Anwendungsfall wird daher eine schwankende Datenverbindung mit einem plötzlich

auftretenden Funkloch simuliert. Die notwendigen Methoden liefert das Zomby-Framework. Listing 6.3 zeigt den entsprechenden Testcode.

```
public void testNetworkConnection() throws Exception {  
    ...  
    Zomby.getNetwork().setChangingNetworkSpeed(5, 3000);  
    ...  
    Zomby.getGSM().setDeadZone();  
    ...  
}
```

Listing 6.3: Zomby-Test: Automatisierung einer schwankenden Datenverbindung mit plötzlichem Funkloch

Die Methode `setChangingNetworkSpeed` wechselt zufällig fünf Mal für drei Sekunden die Netzwerkverbindung, bevor die Methode `setDeadZone` die Sprach- und Datenverbindung ausschaltet, um ein Funkloch zu simulieren.

**Anmerkung:** Der Test kann momentan aufgrund eines Bugs (Fehlers) im Andorid Emulator (siehe Abschnitt 6.3.2) nicht erfolgreich durchgeführt werden.

### 6.1.3 Fußgänger

In diesem Anwendungsfall befindet sich das Android-Gerät entweder in der Hand oder in der Tasche eines Fußgängers. Dieses Szenario bietet sich für alle Entwickler an, die einen Lokalisierungsdienst testen wollen. Listing 6.4 zeigt wie einfach dies mit einem Zomby-Test simuliert werden kann.

```
public void testWalker() throws Exception {  
    // startpoint  
    double sLongitude = 7.185311;  
    double sLatitude  = 50.780011;  
  
    // endpoint  
    double eLongitude = 7.181947;  
    double eLatitude  = 50.780177;  
  
    Zomby.getGeo().simulateWalkingPedestrian(sLongitude, sLatitude,  
        eLongitude, eLatitude);  
  
    // methods to test the location service
```

```

    ...
}

```

Listing 6.4: Zomby-Test: Simulation eines Fußgängers, der eine Straße entlanggeht

Dafür müssen nur ein Startpunkt sowie ein Endpunkt initialisiert und diese der Methode `simulateWalkingPedestrian` übergeben werden. Da die Methode in einem eigenen Thread läuft, können gleichzeitig die passenden Testmethoden ausgeführt werden, um den Lokalisierungsdienst anhand des wechselnden Standorts des Fußgängers zu testen.

Alternativ gibt es noch die Methode `simulateMovingObject`, um ein sich bewegendes Objekt zu simulieren. Dafür muss zusätzlich ein weiterer Parameter für die Geschwindigkeit [in km/h] angegeben werden. Aufgrund der Geschwindigkeitsangabe müssen beide Methoden in Echtzeit ablaufen. Deshalb muss noch erwähnt werden, dass der Geschwindigkeitsparameter auch dafür verwendet werden kann, um die Simulationen insgesamt zu beschleunigen. Die Methode `simulateMovingObject(sLongitude, sLatitude, eLongitude, eLatitude, 12)` simuliert beispielsweise einen doppelt so schnellen Fußgänger<sup>1</sup>.

### 6.1.4 Zugfahrt / Autofahrt

Dieser Anwendungsfall unterscheidet sich zum vorigen nur in der Komplexität der Streckenführung. Während es bei einem Fußgänger meist ausreicht, dass er eine Straße entlanggeht, die eine Gerade bildet, verhält es sich bei einer Auto- bzw. Zugfahrt viel komplexer. Wenn die Route nicht aus wenigen Geraden, sondern aus vielen Kurven besteht, ist es für den Entwickler ratsamer, selber die Fahrt einmal abzufahren und diese mit dem Zomby Recorder mittels Lokalisierung aufzuzeichnen. In Listing 6.5 ist ein Ausschnitt einer aufgezeichneten Zugfahrt von Köln nach Sankt Augustin dargestellt.

```

1372177909087;LOCATION;6.97480815,50.94087664
1372177914099;LOCATION;6.97533411,50.94078389
1372177915086;LOCATION;6.9761956,50.94053655
1372177916109;LOCATION;6.97663956,50.94039723
1372177917083;LOCATION;6.97704304,50.94043996
...
1372178930085;LOCATION;7.2022519,50.79428565
1372178932164;LOCATION;7.20236826,50.79421549
1372178936149;LOCATION;7.20246166,50.7941418
1372178945123;LOCATION;7.20262294,50.79412373

```

<sup>1</sup>Die Geschwindigkeit des Fußgängers bei der Methode `simulateWalkingPedestrian` beträgt 6 km/h

```
1372178965080;LOCATION;7.20269732,50.79404625
```

Listing 6.5: Zomby Recorder: Lokalisierungsaufzeichnung einer Zugfahrt (Ausschnitt)

In dem Fall müssen die Start- und Endpunkte nicht mühsam per Hand initialisiert werden, sondern nur der `InputStream` einer aufgezeichneten `zf`-Datei. Der zweite übergebene Parameter der Methode `playMyRecord` sorgt dafür, dass die Zugfahrt in zehnfacher Echtzeit simuliert wird.

```
public void testTrainJourney() throws Exception {  
    ...  
    InputStream input = getInstrumentation().getContext()  
                        .getAssets().open("trainJourney.zf");  
    Zomby.getZombyPlayer().playMyRecord(input, 0.1);  
    ...  
}
```

Listing 6.6: Zomby-Test: Simulation einer Zugfahrt

Auf diese Weise können nicht nur Zugfahrten simuliert werden, sondern auch Autofahrten oder andere ähnlich komplexe Abläufe, die eine Standortbestimmung benötigen.

### 6.1.5 Sensoren

Beim Testen von Sensordaten spielt Zomby seine größte Stärke aus. Es ist für den Entwickler oft ein schwieriges Unterfangen, diejenigen Sensordaten zu produzieren, die für einen bestimmten Test erforderlich sind. In diesem Anwendungsfall wird eine Temperaturanzeige getestet, die die Temperatur von 60 bis -10 Grad anzeigen soll. Selbst im Feldtest kann ein Entwickler dies nicht reproduzieren. Listing 6.7 zeigt den passenden automatisierten Zomby-Test.

```
public void testTemperatureSensor() throws Exception {  
    ...  
    Zomby.getSensor().simulateTemperaturDrifting(60, -10, 3000);  
    ...  
}
```

Listing 6.7: Zomby-Test: Automatisierung eines Temperaturverlaufs



## 6.2 Technische Grenzen

Der gewählte Lösungsansatz des Zomby-Frameworks bietet die wenigsten Abhängigkeiten gegenüber den Alternativ-Lösungen. Genau genommen ist seine einzige Abhängigkeit die vom Android Emulator. Die sich daraus ergebenden Vorteile wurden im Unterkapitel 5.2 ausführlich erläutert, hier folgen nun die Nachteile.

Die Mächtigkeit des Frameworks ist abhängig von der Mächtigkeit des Emulators. Um diese Tatsache zu verdeutlichen, sind in Tabelle 6.2 alle Sensortypen aufgelistet, die das Android-Framework unterstützt [Goo13]. In der rechten Spalte ist symbolisiert, welche vom Zomby-Framework unterstützt werden und welche nicht. Die große Diskrepanz, dass so viele Sensortypen nicht vom Framework unterstützt werden, liegt nicht am Framework selber, sondern am Android Emulator. Dieser unterstützt im Moment nur diese fünf Sensoren. Aus diesem Grund kann das Zomby-Framework auch nur diese fünf Sensoren unterstützen.

Neben der mangelnden Sensor-Unterstützung ergeben sich noch weitere technische Einschränkungen. In den nachfolgenden Abschnitten werden einige Beispiele kurz erläutert.

### 6.2.1 Emulator unterstützt kein echtes WLAN

Der Emulator unterstützt keine virtuelle WLAN-Karte, so dass es generell nicht möglich ist, eine WLAN-Verbindung aufzubauen. Für das Testen bedeutet dies, dass nicht nach dem Handynet und dem WLAN abgefragt werden kann, sondern einzig über das Handynet. Für eine uneingeschränkte Internetverbindung sorgt zumindest der Telnetbefehl:

```
$ network speed full
```

### 6.2.2 Emulator hat keinen Telnetbefehl für den Flugmodus

Der Flugmodus kann nicht über einen Telnetbefehl aktiviert bzw. deaktiviert werden, sondern nur über den Android Emulator selbst. Über die folgenden Telnetbefehle:

```
$ gsm data off  
$ gsm voice off
```

kann zumindest der Flugmodus simuliert werden, indem die Daten- und die Sprachverbindung ausgeschaltet werden. In Zomby entspricht das der Methode `setDeadZone()` aus der Klasse














Sensor	Beschreibung	Von Zomby unterstützt
TYPE_ACCELEROMETER	Misst die Beschleunigung in m/s <sup>2</sup> , die auf allen drei physikalischen Achsen (x, y, z) des Gerätes wirkt.	
TYPE_AMBIENT_TEMPERATURE	Misst die Raumtemperatur in °C	
TYPE_GRAVITY	Misst die Schwerkraft in m/s <sup>2</sup> , die auf allen drei physikalischen Achsen (x, y, z) des Gerätes wirkt.	
TYPE_GYROSCOPE	Misst Drehrate des Geräts in rad/s um jede der drei physikalischen Achsen (x, y, z).	
TYPE_LIGHT	Misst die Umgebungshelligkeit (Beleuchtung) in lx.	
TYPE_LINEAR_ACCELERATION	Berechnet die Beschleunigungskraft ohne die Schwerkraft in m/s <sup>2</sup> , die auf allen drei physikalischen Achsen (x, y, z) des Gerätes wirkt.	
TYPE_MAGNETIC_FIELD	Misst das Umgebungserdmagnetfeld für alle drei physikalischen Achsen (x, y, z) in $\mu$ T.	
TYPE_ORIENTATION	Misst den Rotationsgrad aller drei physikalischen Achsen (x, y, z).	
TYPE_PRESSURE	Misst die Umgebungsluft in hPa oder mbar.	
TYPE_PROXIMITY	Misst die Nähe eines Objekts in cm, relativ zum Sichtschirm eines Gerätes.	
TYPE_RELATIVE_HUMIDITY	Misst die relative Luftfeuchtigkeit in Prozent.	
TYPE_ROTATION_VECTOR	Misst die Orientierung eines Geräts durch die drei Elemente des Geräte-Drehvektors.	
TYPE_TEMPERATURE	Misst die Temperatur der Vorrichtung in °C.	

Tabelle 6.1: Liste aller unterstützten Sensortypen der Android-Plattform (Angelehnt an [Goo131])

GSM. Dies kann jedoch nicht den echten Flugmodus ersetzen, weil es sich dabei betriebstechnisch um einen anderen Zustand handelt.

## 6.3 Fehlerbericht - Bugs im Android Emulator

Im vorherigen Unterkapitel wurde bereits erläutert, warum das Framework aufgrund mangelnder Funktionen des Emulators Einschränkungen hinnehmen muss. Hier folgt eine Auflistung von Bugs (Fehlern) der Emulatorsoftware, die während der Erstellung der Arbeit aufgefallen sind. Diese Fehler sorgen für weitere Einschränkungen in der Framework-Nutzung.

Die Behebung der Bugs würde hier jedoch gleichzeitig zu einer Aufhebung der entsprechenden Einschränkungen führen. Eine Anpassung des Frameworks, wie es bei einer Funktionserweiterung notwendig wäre, entfällt.

### 6.3.1 Batterieanzeige

Manchmal zeigt die Emulatorsoftware einen fehlerhaften Batteriezustand an (Abbildung 6.1). Der Zustand steht in diesem Fall auf UNKNOWN. Bei einer Zustandsänderung (per Telnet) führt dies zu einem Absturz der Emulatorinstanz. Dieser Fehler tritt jedoch nur selten auf und kann meist durch eine Neuinstallation des AVDs behoben werden.

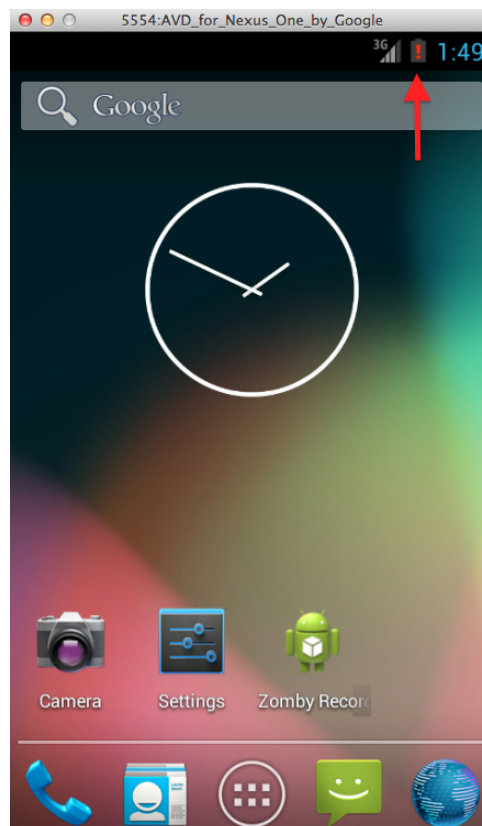


Abbildung 6.1: Fehlerhafter Batteriestatus des Android Emulators

### 6.3.2 GSM-Zustandwechsel

Beim Wechsel des GSM-Zustands von `on/home` auf einen anderen bricht aufgrund eines Emulator-Bugs die adb-Verbindung zusammen<sup>2</sup>. Dieser Bug hat weitreichende Konsequenzen. Somit ist nicht nur das uneingeschränkte Wechseln von GSM-Zuständen unmöglich und damit auch die Simulation von plötzlichen Funklöchern innerhalb eines Tests, sondern auch jede andere GSM-zustandsabhängige Aktivität, wie zum Beispiel das Telefonieren. Bei einem eingehenden Anruf wird die mobile Datenverbindung während des Anrufs auf `off` gesetzt. War diese vor dem Anruf auf `on/home`, wird auch hier die adb-Verbindung unterbrochen.

### 6.3.3 Falscher Netzwerkstatus

Beim Testen des Zomby-Recorders fiel auf, dass der Emulator plötzlich bei einem Zustandswechsel von UMTS auf EDGE der entsprechende Action-Intent nicht auslöst. Das Netzwerk steht zwar auf EDGE, jedoch kann der passende BroadcastReceiver nicht aufgerufen werden. Erst bei einem erneuten Wechsel von EDGE auf UMTS löst der "verpasste" Intent mit den alten Daten aus, so dass ab diesem Zeitpunkt die Klasse *NetworkInfo* immer den unkorrekten Netzwerkstatus ausgibt.

Durch diese Verschiebung lässt sich ein aufgezeichnetes Szenario nicht authentisch rekonstruieren.

## 6.4 Zusammenfassung

Für das automatisierte Testen von wechselnden Umgebungsvariablen im Android-Kontext ist das Zomby-Framework eine sehr gute Lösung. Es ermöglicht die Simulation sehr vieler wichtiger Anwendungsfälle, die ansonsten nicht automatisiert testbar wären. Mithilfe der Low-Level-API bietet es dem Testentwickler maximale Flexibilität und gleichzeitig mithilfe der High-Level-API großen Komfort beim Erstellen des Testcodes. Dieser ist dank des API-Designs sehr gut lesbar. Der Zomby Recorder, der für die Aufzeichnung von schwierigen bzw. realen Daten (Parametern) genutzt werden kann, rundet das Framework insgesamt ab.

Für eine abschließende Bewertung der Lösung zeigt Tabelle 6.2, wie gut das neue Framework die Umgebungsvariablen simuliert, die in Unterkapitel 3.1 für die Bewertung von bestehenden Testframeworks herangezogen wurden.

---

<sup>2</sup>Der Autor hat diesen Fehler dem Android Projekt gemeldet (Status ist über <http://code.google.com/p/android/issues/detail?id=54734> erreichbar)

Umgebungsvariable	Von Zomby unterstützt
Netzwerkverbindung	
Batterie	
Lokalisierung	
Sensoren	

Tabelle 6.2: Vom Zomby-Framework unterstützte Umgebungsvariablen

Insgesamt unterstützt das Framework alle geforderten Kriterien. Wegen der Abhängigkeit des Android Emulators mit seinen mangelnden Funktionen und seiner fehlerhaften Software ist das Framework jedoch nicht uneingeschränkt nutzbar (gelbe Haken)<sup>34</sup>. Dies sind jedoch ausnahmslos Einschränkungen, die durch eine Funktionserweiterung oder eine Fehlerbehebung des Emulators beseitigt werden könnten. Das Konzept des Frameworks stimmt grundlegend. Es bietet zudem auch ein riesiges Potenzial, das im nächsten Kapitel ausführlich beschrieben wird.

<sup>3</sup>Einschränkungen der Sensoren, siehe Tabelle 6.2

<sup>4</sup>Einschränkungen der Netzwerkverbindung, siehe Abschnitt 6.3.2

## 7 Testen mit Zomby

In diesem Zwischenkapitel wird kurz erläutert, was ein Testentwickler beachten muss, wenn er das Zomby-Framework nutzen möchte. Nach der Erklärung wie die Entwicklerumgebung einzurichten ist, folgen Testbeispiele mit verschiedenen Testframeworks, in denen Zomby integriert wurde. Bei welchen Anwendungsfällen es günstiger ist, den Zomby Recorder für die Parametereingabe zu verwenden, statt sie manuell einzugeben, zeigt ein weiterer Abschnitt. Zu guter Letzt folgt ein Erfahrungsbericht, der auf die Tücken des Frameworks aufmerksam macht und ein Gefühl dafür gibt, wie mit Zomby umzugehen ist.

### 7.1 Einrichtung der Entwicklungsumgebung

#### 7.1.1 Einrichtung des Webservers

Das Framework setzt seine Telnetbefehle über das Web-Servlet „*ZombyServlet*“ ab. Dieses muss auf einem lokalen Server auf dem Entwicklungsrechner laufen. Für die Inbetriebnahme des Servlets kann entweder die Datei „*ZombyWebservice.war*“<sup>1</sup> auf einem Application-Server eingebunden oder das ZombyServlet-Projekt in einer IDE wie Netbeans ausgeführt werden. Die einfachste Variante ist jedoch die Ausführung eines gradle-Scripts mit folgendem Konsolenbefehl<sup>2</sup>:

```
$ gradle jettyRun
```

Da das Servlet die Telnetbefehle über das Android-Tool *adb* absetzt, muss vor dem Starten des Servers noch eine System-Umgebungsvariable namens `ADB_PATH` gesetzt werden. Sie soll den entsprechenden adb-Pfad angeben. Unter Unix sieht das beispielsweise so aus:

```
$ ADB_PATH=/Users/manuel/android-sdk/platform-tools/
```

---

<sup>1</sup>Befindet sich im Ordner `server/ZombyWebservice/dist/`

<sup>2</sup>Die entsprechende build-Datei befindet sich im Ordner `server/ZombyWebservice/`

Sollte die Umgebungsvariable falsch oder gar nicht angegeben sein, werfen die Zomby-Methoden eine *ZombyException* (siehe Kapitel 5.3.4). Dies geschieht ebenfalls, sollte das *ZombyServlet* nicht erreichbar sein.

### 7.1.2 Einbindung der Framework-Bibliothek

Das Zomby-Framework kann nicht als Jar-Datei in ein bestehendes Testprojekt importiert, sondern nur als Bibliothek eingebunden werden, weil Zomby einige Klassen aus der Android-Bibliothek nutzt. Das Zomby-Projekt muss deshalb zunächst importiert und als Bibliothek gekennzeichnet werden (siehe Abbildung 7.1(a)). Erst danach kann Zomby als Bibliothek in ein Testprojekt eingebunden werden (siehe Abbildung 7.1(b)).

**Anmerkung:** Bei manchen Testframeworks, wie beispielsweise das *UiAutomator-Framework*, führt dies jedoch zu einem `java.lang.NoClassDefFoundError`. Dabei werden die Klassen beim Packen nicht mit in die .apk-Datei eingebunden. Dieses Problem kann gelöst werden, indem die Quelldateien des Frameworks mit in das Testprojekt kopiert werden<sup>3</sup>.

## 7.2 Robotium mit Zomby

Robotium ist ein sehr häufig eingesetztes Android-Testframework. Einer der Gründe zeigt sich in Tabelle 3.1 auf Seite 31. Es besitzt im Gegensatz zu allen anderen analysierten Frameworks keine gravierenden Nachteile. Wie Zomby in Robotium integriert werden kann, zeigt Listing 7.1.

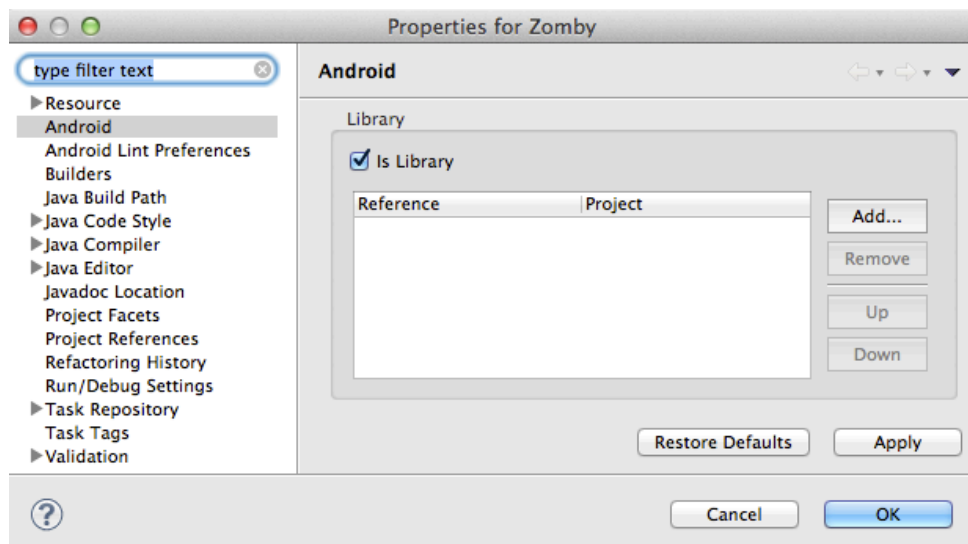
Der zu testende Teil der Anwendung besteht aus zwei Anzeigen, einer *SeekBar* und einer *RatingBar*. Die *SeekBar* zeigt mit einem Regler, dessen Werte von 0-100 gehen, den aktuellen Batteriestand an, während die *RatingBar* das aktuelle Handynetzwerk in Form von Sternen darstellt. Die Anzahl der Sterne (maximal fünf) bildet die relative Geschwindigkeit der einzelnen Netze ab (ein Stern ist GSM, drei Sterne UMTS, usw.).

Im folgenden Listing 7.1 werden die aktuellen Werte der Anzeigen einmal vor und einmal nach der Manipulation der Batterie- und Netzwerkwerte durch Zomby überprüft. Der Test soll zeigen, ob die Anzeigen die neuen Werte, wie erwartet, übernehmen oder nicht.

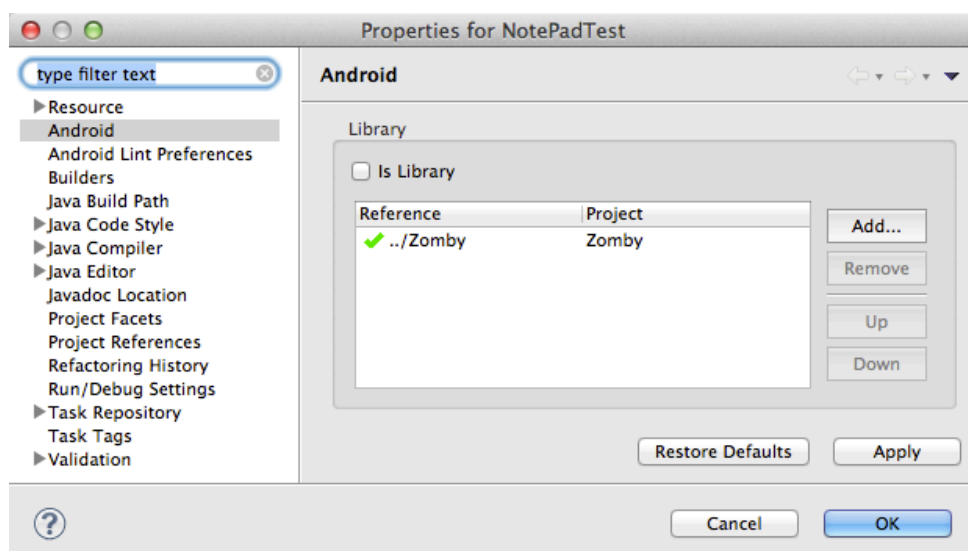
```
public class ZombyDemoAppTest extends
    ActivityInstrumentationTestCase2<ZombyDemoAppActivity>{
    private Solo solo;
    private ArrayList<SeekBar> seekBarList;
```

---

<sup>3</sup>Dient hier als Workaround, weil bessere Lösungen an dieser Stelle aus Zeitgründen nicht in Erwägung gezogen wurden



(a) Zomby-Projekt als Bibliothek kennzeichnen



(b) Zomby-Bibliothek in Testprojekt einbinden

Abbildung 7.1: Einbinden des Zomby-Frameworks mit Eclipse

```
private ArrayList<RatingBar> ratingBarList;
private SeekBar seekBar;
private RatingBar ratingBar;

int startCapacity = 100;
NetworkSpeedRating staRating = NetworkSpeedRating.EDGE;

public ZombyDemoAppTest() {
    super(ZombyDemoAppActivity.class);
}

@Override
```



```
public void setUp() throws Exception {
    solo = new Solo(getInstrumentation(), getActivity());

    // start conditions
    Zomby.getCorePower().capacity(staCapacity);
    Zomby.getCoreNetwork().speed(NetworkSpeed.EDGE);
}

@Override
public void tearDown() throws Exception {
    solo.finishOpenedActivities();
}

public void testSeekBar() throws Exception {
    solo.waitForActivity("MainActivity");

    // get seekBar object
    seekBarList = solo.getCurrentViews(SeekBar.class);
    if(!seekBarList.isEmpty())
        seekBar = seekBarList.get(0);
    else
        throw new Exception("seekBar object was null");

    // check start value
    assertEquals(staCapacity, seekBar.getProgress());

    // change current battery value
    int currentCapacity = 83;
    Zomby.getCorePower().capacity(currentCapacity);

    // check changed value
    assertEquals(currentCapacity, seekBar.getProgress());
}

public void testRatingBar() throws Exception {
    solo.waitForActivity("MainActivity");

    // get ratingBar object
    ratingBarList = solo.getCurrentViews(RatingBar.class);
    if(!ratingBarList.isEmpty())
        ratingBar = ratingBarList.get(0);
}
```

```

else
    throw new Exception("ratingBar object was null");

// check start value
assertEquals(staRating.getValue(), ratingBar.getProgress());

// change current network speed
Zomby.getCoreNetwork().speed(NetworkSpeed.UMTS);

// check changed value
NetworkSpeedRating curRating = NetworkSpeedRating.UMTS;
assertEquals(curRating.getValue(), ratingBar.getProgress());
}
}

```

Listing 7.1: Anwendungstestfall mit Robotium und Zomby

In der `setUp()`-Methode wird die Batterie vor jedem Test auf hundert Prozent und das Netzwerk auf EDGE gesetzt. Die beiden Methoden `testSeekBar()` und `testRatingBar()` haben den gleichen Aufbau. Zuerst wird mit der Methode `getCurrentViews` das jeweilige Bar-Objekt zurückgegeben und in eine passende Variable gespeichert. Danach folgt mit `assertEquals` eine Abfrage, ob der aktuelle Wert der Bar mit der jeweiligen Anfangsbedingung übereinstimmt. Im Falle von `testSeekBar()` wird im Anschluss die Batterie auf 83 Prozent gesetzt und erneut geprüft, ob das `SeekBar`-Objekt den veränderten Wert angenommen hat. Analog verhält es sich bei der `testRatingBar()`-Methode, nur dass dort auf den veränderten UMTS-Wert geprüft wird.

### 7.3 UiAutomator mit Zomby

Die Kombination von UiAutomator und Zomby ist besonders interessant, weil beide Frameworks systemweit operieren. Die Besonderheit besteht bereits darin, dass bei einem UiAutomator-Test keine zu testende Anwendung angegeben werden muss. Diese Black-Box-Tests erlauben das Testen von Anwendungen, deren Quellcode dem Entwickler nicht vorliegen muss. Listing 7.3 zeigt den vollständigen Testcode, der in Listing 6.7 in Abschnitt 6.1.5 nur angedeutet war. Bei diesem Test wird beispielsweise die Anwendung *Android Sensor Box* [IMO13] aufgerufen, deren Quellcode dem Autor nicht vorlag.

```

public class SensorTest extends UiAutomatorTestCase {
    @Override

```

```
protected void setUp() throws Exception {
    super.setUp();
    getUiDevice().pressHome();
    // launch app
    UiScrollable appViews = showApps();
    UiObject app = appViews.getChildByText(new UiSelector()
        .className(android.widget.TextView.class.getName()),
        "Android Sensor Box");
    app.clickAndWaitForNewWindow();
}

public void testTemperatureSensor() throws Exception {
    // click temperature button
    getUiDevice().click(180, 357);
    // temperatur sensor is drifting
    Zomby.getSensor().simulateTemperaturDrifting(60, -10, 3000);
    synchronized(this) {
        wait(3500);
    }
}

@Override
protected void tearDown() throws Exception {
    super.tearDown();
    getUiDevice().pressHome();
}
}
```

Listing 7.2: Zomby-Test: Simulation eines Temperaturverlaufs

Die Methode `setUp()` sorgt dafür, dass vor dem Test der Home-Button gedrückt und im Anwendungsverzeichnis die Software *Android Sensor Box*<sup>4</sup> aufgerufen wird. In der Testmethode `testTemperatureSensor()` erfolgt das Öffnen des Fensters für den Temperatursensor (siehe Abbildung 7.2) durch den Klick auf die Displaykoordinaten (180, 357). Die Zomby-Methode `simulateTemperaturDrifting(-10, 60, 3000)` simuliert einen Temperaturverlauf von 60 bis -10 Grad in drei Sekunden. Nach dreieinhalb Sekunden ist der Test beendet und die Methode `tearDown()` schließt das Programm wieder durch den Klick auf den Home-Button.

**Anmerkung:** Bei Temperaturen unter minus zehn Grad stürzt die Anwendung *Android Sensor Box* ab. Dieser Fehler wäre höchstwahrscheinlich nicht unentdeckt geblieben, wenn Zomby den

<sup>4</sup>Google Play: <https://play.google.com/store/apps/details?id=imoblife.androidsensorbox>



Abbildung 7.2: *Temperature Sensor der Android Sensor Box*

Testentwicklern schon damals zur Verfügung gestanden hätte.

## 7.4 Zomby Recorder

Der Zomby Recorder ist ein mächtiges Werkzeug mit dem sehr einfach reale Sensordaten aufgezeichnet werden können, um sie später mithilfe des Zomby-Frameworks immer und immer wieder automatisiert abspielen zu können. Möchte ein Entwickler die Lenkung eines Rennwagens innerhalb eines Rennspiels automatisiert testen, bei dem der Orientation-Sensor benutzt wird, sollte er dieses Szenario vorher mit dem Zomby Recorder aufzeichnen. Da der Orientation-Sensor jede kleinste Veränderung der Geräteposition im Raum registriert und deshalb mehrmals in der Sekunde seine Werte ändert, sind diese Sensordaten manuell kaum reproduzierbar. Zusätzlich würde deren Beschaffung für den Entwickler zu viel Aufwand bedeuten.

Daher empfiehlt sich der Recorder immer dann, wenn Sensordaten entweder möglichst real sein sollten, oder sie schwer zu ermitteln sind, oder die Parameterangabe mit sehr viel Aufwand verbunden wäre.

## 7.5 Die Tücken von Zomby

In diesem Unterkapitel werden einige Erfahrungen erläutert, die bei der Entwicklung des Frameworks gemacht wurden. Es geht um gewisse Tücken, die bei der Benutzung des Frameworks auftreten können. Dieser Erfahrungsbericht soll ein Gefühl dafür vermitteln, welche Fragen sich ein Testentwickler im Vorfeld stellen sollte, wenn er zukünftig Tests mit Zomby schreiben möchte.

### 7.5.1 Umgang mit Fehlern

Nicht alles was theoretisch mit dem Framework möglich ist, kann auch umgesetzt werden. Die Unterkapitel 6.2 und 6.3 haben bereits einen ausführlichen Einblick darüber gegeben, welche Einschränkungen dem Testentwickler unterlegen sind. Das Unterkapitel 6.3 beinhaltet nur Fehler, die während der Entwicklungsphase des Frameworks aufgefallen sind. Der *Issue Tracker* des *Android Open Source Projects* [Goo13j] listet alle von Benutzern gemeldeten Fehler bzw. Fehlverhalten von Android auf, wie der Fehler aus Abschnitt 6.3.2. Um mit Zomby korrekte Tests zu schreiben, ist es wichtig die Fehler des Android Emulators zu kennen. Ansonsten kann es zu einem unerwarteten Testverhalten kommen.

### 7.5.2 Vorsicht vor Flüchtigkeitsfehlern

Ein Test muss aber nicht nur wegen technischer Barrieren fehlschlagen. Manchmal können Flüchtigkeitsfehler ein Grund dafür sein, warum sich ein Test nicht so verhält wie erwartet. Wird beispielsweise das Laden der Batterie simuliert, indem der Batteriestatus immer weiter steigt, wobei der Batteriestatus auf `DISCHARGING` (Entladen) steht, statt auf `CHARGING` (Aufladung), kann es zu unerwünschten Nebeneffekten kommen.

Ein harmloser Flüchtigkeitsfehler beispielsweise ist in diesem Zusammenhang, den Batteriestatus auf 0 zu setzen, weil dann das Betriebssystem herunterfährt und der weitere Verlauf des Tests fehlschlägt.

### 7.5.3 Die Grenzen der Echtzeit

Der Umgang mit dem Zomby Recorder hat auch seine Grenzen. Zwar ist es prinzipiell möglich Anwendungsfälle in Echtzeit aufzunehmen und sie auch in Echtzeit abzuspielen, jedoch ist die Genauigkeit eingeschränkt. Dies hat wiederum einen technischen Hintergrund. Es kommt bei

der Kommunikation mit dem Webserver und deren Absetzung der Telnetbefehle zu unkontrollierbaren Zeitverzögerungen. Jeder der sechs Schritte, die für diese Verarbeitung notwendig sind (siehe Abbildung 5.8 auf Seite 50), bildet eine potentielle Verzögerungsquelle, da die Ausführungszeit der Schritte je nach Prozessorauslastung variieren kann. Insgesamt sorgt dies für eine inkonstante Verarbeitungszeit. Extrem zeitkritische Anwendungsfälle können so nicht getestet werden, weil sie aufgrund der unterschiedlichen Ausführungszeit nicht exakt reproduzierbar sind.

Ein Beispiel für solch einen Anwendungsfall befindet sich auf der beigefügten DVD (siehe Anhang). Es wurde ein virtuelles Kugellabyrinth getestet, indem das erste Level des Spiels mit dem Zomby Recorder aufgenommen und später per Zomby-Framework wieder abgespielt wurde. In den seltensten Fällen kam die Kugel ins Ziel<sup>5</sup>. Die meiste Zeit landete die Kugel aufgrund der Zeitverzögerungen in einem der zahlreichen Löchern.

Diese Tests sollten demnach im Idealfall so gestaltet werden, dass sie die konzeptbedingte Zeitverzögerung kompensieren können.

#### 7.5.4 Drehen an der Echtzeitschraube

Für den Testentwickler gibt es zwei „Drehschrauben“, an denen er die Verzögerung regulieren kann. Listing 7.3 zeigt dafür einen Ausschnitt aus dem Kugellabyrinth-Test.

```
public class LabyrinthTest extends UiAutomatorTestCase {

    public void testFirstLevel() throws Exception {
        Zomby.getCoreSensor().set(Sensorname.ACCELERATION,
            -2.0294318, 2.070293, 9.997335); waitForRealtime(8);
        Zomby.getCoreSensor().set(Sensorname.ACCELERATION,
            -2.1111538, 2.3699405, 9.765789); waitForRealtime(28);
        ...
        Zomby.getCoreSensor().set(Sensorname.ACCELERATION,
            -2.7513103, 1.334794, 10.501288); waitForRealtime(148);
        Zomby.getCoreSensor().set(Sensorname.ACCELERATION,
            -2.8330324, 2.152015, 10.147159);
    }

    private void waitForRealtime(long time) throws
        InterruptedException {
        //double realtimeFactor = 1.0;
```

---

<sup>5</sup>Demovideo auf der beigefügten DVD zeigt einen erfolgreichen Testlauf

```
//long timeToWait = Math.round(time*realtimeFactor);
if(time>6) {
    synchronized(this) {
        wait(time-6);
    }
}
}
```

Listing 7.3: Ausschnitt aus Kugellabyrinth-Test

Die `waitForRealtime`-Methode soll die Echtzeit garantieren, indem sie die Ausführung zwischen zwei Anweisungen verzögert. In diesem Fall sind die übergebenen Werte die Zeitabstände, die der Zomby Recorder zwischen zwei Sensordaten aufgezeichnet hat. Der aufgezeichnete Anwendungsfall kann demnach nur dann zu hundert Prozent reproduziert werden, wenn diese Abstände beim Abspielen exakt eingehalten werden. Die Methode `waitForRealtime` bietet zwei einfache Möglichkeiten, um die Verzögerung zu beeinflussen.

Die erste ermöglicht die Manipulation der Echtzeitgeschwindigkeit. Der Test könnte eventuell reproduzierbarer werden, wenn er insgesamt schneller oder langsamer als in Echtzeit abläuft. Zu beachten ist dabei, dass die Berechnung der neuen Zeitabstände auch wieder unterschiedlich lange dauern kann. Wenn die Echtzeit im Vordergrund steht, wie es beim Testen des Labyrinths der Fall ist, muss von dieser Berechnung Abstand genommen werden.

Die zweite Möglichkeit bildet die Zeitkorrektur. Jeder Aufruf der `waitForRealtime`- oder der `wait`-Methode selbst dauert einige Millisekunden. Diese Zeit muss von den echten Zeitabständen wieder abgezogen werden, um die Echtzeit des aufgezeichneten Anwendungsfalls einzuhalten. Da die Abstände jedoch meist sehr kurz sind, würde eine genaue Berechnung zu lange dauern. Daher ist es effizienter einen Pauschalwert (hier 6 Millisekunden) anzugeben, der von der Variable `time` abgezogen wird. Mithilfe einer `if`-Schleife muss dann nur noch überprüft werden, welche Abstände größer sind als der abzuziehende Wert, ansonsten würde es zu einer Fehlermeldung kommen. Allerdings kommt es auch hier zu einer Zeitabweichung, weil alle Zeitabstände, die kleiner als oder gleich groß wie der Pauschalwert sind, nicht berücksichtigt werden.

## 8 Fazit und Ausblick

In diesem Kapitel werden noch einmal alle wichtigen Ergebnisse dieser Arbeit in einem Fazit zusammengefasst. Ein Ausblick zeigt auf, wie die Firma inovex GmbH mit dem Ergebnis der Arbeit umgeht und welches Entwicklungspotenzial daraus entstehen könnte.

### 8.1 Fazit

Das Testen von mobilen Anwendungen wird immer wichtiger, weil die Anwendungen immer komplexer werden. Zusätzlich wird dieses durch die Mobilität erschwert, indem die Software auf ständig wechselnde Bedingungen reagieren muss. Mit dieser Arbeit sollte wenigstens der letzte Teil (im Android-Kontext) vereinfacht werden. Die Vorgabe war es, wechselnde Umgebungsänderungen zu simulieren, um automatisierte Tests schreiben zu können. In Kapitel 3 wurde in einer Analyse festgestellt, dass die bisher auf dem Markt zur Verfügung stehenden Lösungen diese Aufgabe nicht erfüllen. In Kapitel 4 wurden verschiedene neue Lösungsansätze untersucht, um am Ende den besten Ansatz zu ermitteln. Umgesetzt wurde diese Lösung in Kapitel 5, indem ein neues Testframework namens *Zomby* entwickelt wurde. Im Gegensatz zu allen anderen Testframeworks hat *Zomby* keinen eigenen *Testrunner*, der Einfluss auf das Betriebssystem nehmen kann, sondern nutzt stattdessen die Telnet-Schnittstelle des Android Emulators. Zur Laufzeit wird die Emulator-Instanz manipuliert, auf der gerade eine Testanwendung läuft, um so wechselnde Umgebungsänderungen zu simulieren. Die eigentliche Manipulation des Android Emulators übernimmt ein lokaler Webserver, der innerhalb des Android-Testprogramms angefragt wird. Das Testen auf echten Geräten ist mit *Zomby* daher nicht möglich. Deshalb ist es auch kein eigenständiges Testframework, sondern dient zur Ergänzung eines bestehenden Testframeworks (siehe Abbildung 4.2).

Daraus entstehen erst einmal nur Vorteile. Zum einen können die bisherigen Lösungen ihre Stärken voll ausspielen, zum anderen muss bestehender Testcode nicht umgeschrieben werden, wie es bei dem *SensorSimulator* der Fall ist (siehe Abschnitt 3.2.8). Der Testcode wird lediglich ergänzt, um auf bestimmte mobile Bedingungen zu testen. Das Framework bietet dem Entwickler eine sehr mächtige und umfangreiche API (Low- und High-Level), mit der er kom-



portable und flexible Tests schreiben kann (siehe Abschnitt 5.3.3). Darüber hinaus existiert eine Rekorder-App (*Zomby Recorder*), die das Aufzeichnen von echten Sensordaten ermöglicht. Damit wird zum einen der Schreibaufwand der Tests verringert und zum anderen die Authentizität der Tests erhöht. Der einzige Schwachpunkt ist die sehr große Abhängigkeit von dem Android Emulator. In der Evaluierung (siehe Kapitel 6) wurde ausführlich erklärt, welche Probleme diese Abhängigkeit mit sich bringen kann und wie diese gegebenenfalls behoben werden können.

Es ist jedoch letztendlich gelungen, die vorgegebene Problemstellung mithilfe des Zomby-Frameworks zu lösen. Zur Zeit geschieht dies zwar noch mit Einschränkungen, was allerdings nicht am Konzept des Frameworks liegt, sondern nur am Android Emulator (siehe Unterkapitel 6.4). Im Ausblick wird erläutert, warum auch diese Einschränkungen zukünftig wegfallen könnten.

## 8.2 Ausblick

Das hier im Auftrag der inovex GmbH entwickelte Zomby-Framework wird später<sup>1</sup> als Open-Source-Lösung der Community zur Verfügung gestellt. Erst durch diese Entscheidung kann sich das ganze Potenzial des Frameworks entwickeln. Während die Low-Level-API den funktionalen Teil des Frameworks bildet und daher nur im Falle einer Funktionserweiterung des Android Emulators weiterentwickelt werden kann, ergeben sich bei der High-Level-API unendlich viele Möglichkeiten, das Framework zu erweitern. Wie bereits in Abschnitt 5.3.1 beschrieben, soll die High-Level-API nur Methoden enthalten, die einen Anwendungsfall darstellen, damit einerseits das Schreiben von Tests vereinfacht wird und andererseits der Testcode lesbar bleibt. Jeder Anwendungsfall bildet daher eine potenzielle Methode der High-Level-API. Mithilfe der Community kann so ein breites Spektrum an Anwendungsfällen zusammengetragen werden. Damit das Framework auch in diesem Fall nicht unübersichtlich wird, verteilen sich die Methoden abhängig von ihrem Kontext auf die Klassen *Geo*, *GSM*, *CDMA*, *Network*, *Power* und *Sensor*.

Es ist durchaus möglich, dass sich Zomby später neben *Robotium* und *UiAutomator* etablieren kann. Voraussetzung dafür wäre jedoch eine starke Community, die bei seinem Entwicklungsprozess mithilft. Eine Erweiterung wäre zum Beispiel die Parallelisierung von Zomby-Tests, indem mehrere Emulator-Instanzen angesteuert werden. Des weiteren wären auch weitere Tools denkbar, die unter anderem Sensordaten generieren können (z.B. Autobahnfahrten), um das Aufzeichnen von Sensordaten zu vereinfachen.

---

<sup>1</sup>Der genaue Erscheinungstermin steht zum jetzigen Zeitpunkt noch nicht fest

Darüber hinaus könnte das Unternehmen Google dafür sorgen, dass das Framework noch leistungstärker wird, indem dessen Entwickler den Android Emulator mit noch mehr unterstützten Sensoren ausstatten und ihn gleichzeitig von seinen Fehlern befreien.

# Literaturverzeichnis

- [App13] APPLE INC.: *App Store übertrifft 40 Milliarden Downloads mit knapp der Hälfte in 2012.* <http://www.apple.com/de/pr/library/2013/01/07App-Store-Tops-40-Billion-Downloads-with-Almost-Half-in-2012.html>. Version: Januar 2013
- [Bec03] BECK, Kent: *Test Driven Development: By Example*. Addison-Wesley, Pearson Education, 2003
- [Chi13] *Apps im Store: Android überholt erstmals iOS.* [http://www.chip.de/news/Apps-im-Store-Android-ueberholt-erstmals-iOS\\_59819321.html](http://www.chip.de/news/Apps-im-Store-Android-ueberholt-erstmals-iOS_59819321.html). Version: Januar 2013
- [ETP12] *Robotium - Example Test Projekt.* [https://robotium.googlecode.com/files/ExampleTestProject\\_v3.6.zip](https://robotium.googlecode.com/files/ExampleTestProject_v3.6.zip). Version: Dezember 2012
- [Gar11] GARGENTA, Marko: *Einführung in die Android-Entwicklung*. O'Reilly Germany, 2011
- [GAT13] *Google Android Tools ddmlib r10 API.* <http://www.jarvana.com/jarvana/view/com/google/android/tools/ddmlib/r10/ddmlib-r10-javadoc.jar!/overview-summary.html>. Version: März 2013
- [Git13] *Pivotal Labs - Robolectric Sample Test Code.* <https://github.com/pivotal/RobolectricSample/blob/master/src/test/java/com/pivotallabs/HomeActivityTest.java>. Version: Februar 2013
- [Goo13a] GOOGLE INC.: *Android Developers - Android Debug Bridge.* <http://developer.android.com/tools/help/adb.html>. Version: Juli 2013
- [Goo13b] GOOGLE INC.: *Android Developers - Emulator Networking.* <http://developer.android.com/tools/devices/emulator.html#emulatonetworking>. Version: März 2013
- [Goo13c] GOOGLE INC.: *Android Developers - Managing the Activity Lifecycle.* <http://developer.android.com/training/basics/activity-lifecycle/index.html>. Version: Juli 2013

- [Goo13d] GOOGLE INC.: *Android Developers - Managing Virtual Devices*. <http://developer.android.com/tools/devices/index.html>. Version: März 2013
- [Goo13e] GOOGLE INC.: *Android Developers - monkeyrunner*. [http://developer.android.com/tools/help/monkeyrunner\\_concepts.html](http://developer.android.com/tools/help/monkeyrunner_concepts.html). Version: März 2013
- [Goo13f] GOOGLE INC.: *Android Developers - monkeyrunner Sample Test Programm*. [http://developer.android.com/tools/help/monkeyrunner\\_concepts.html#SampleProgram](http://developer.android.com/tools/help/monkeyrunner_concepts.html#SampleProgram). Version: Februar 2013
- [Goo13g] GOOGLE INC.: *Android Developers - Testing Fundamentals*. [http://developer.android.com/tools/testing/testing\\_android.html#TestAPI](http://developer.android.com/tools/testing/testing_android.html#TestAPI). Version: Januar 2013
- [Goo13h] GOOGLE INC.: *Android Developers - UI Testing*. [http://developer.android.com/tools/testing/testing\\_ui.html](http://developer.android.com/tools/testing/testing_ui.html). Version: März 2013
- [Goo13i] GOOGLE INC.: *Android Developers - UiAutomator Sample Test Code*. [http://developer.android.com/tools/testing/testing\\_ui.html#sample](http://developer.android.com/tools/testing/testing_ui.html#sample). Version: Februar 2013
- [Goo13j] GOOGLE INC.: *Android Open Source Project - Issue Tracker*. <http://code.google.com/p/android/issues/list>. Version: Juli 2013
- [Goo13k] GOOGLE INC.: *Android SDK*. <http://developer.android.com/sdk/index.html>. Version: Juli 2013
- [Goo13l] GOOGLE INC.: *Introduction to Sensors*. [http://developer.android.com/guide/topics/sensors/sensors\\_overview.html#sensors-intro](http://developer.android.com/guide/topics/sensors/sensors_overview.html#sensors-intro). Version: Mai 2013
- [Goo13m] GOOGLE INC.: *UI/Application Exerciser Monkey*. <http://developer.android.com/tools/help/monkey.html>. Version: März 2013
- [Ham09] HAMILL, Paul: *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, 2009
- [Hbl12] *Handelsblatt*. <http://www.handelsblatt.com/unternehmen/it-medien/marktforscher-eine-milliarde-smartphones-in-betrieb/7266118.html>. Version: Oktober 2012
- [Hel13] HELLEBERG, Dominik: Emulierte Roboter. In: *Javamagazin* (2013), Nr. 4, S. 108–111

- [Hof08] HOFFMANN, Dirk W.: *Software-Qualität*. Springer London, Limited, 2008
- [IMO13] IMOBLIFE INC.: *Android Sensor Box*. <https://play.google.com/store/apps/details?id=imoblife.androidsensorbox&hl=en>. Version: Juni 2013
- [JU113] *JUnit*. <http://junit.org>. Version: Juli 2013
- [KM10] KROPP, Martin ; MORALES, Pamela: Automated GUI Testing on the Android Platform. In: *IMVS Fokus Report* (2010)
- [LT 13] LT INFOTECH: *Android Test Automation Framework*. [http://www.lntinfotech.com/Brochure/Android\\_Test\\_Automation\\_Framework.pdf](http://www.lntinfotech.com/Brochure/Android_Test_Automation_Framework.pdf). Version: März 2013
- [MS12] MILETTE, Greg ; STROUD, Adam: *Professional Android Sensor Programming*. John Wiley & Sons, 2012
- [Olb03] OLBRICH, Alfred: *Netze - Protokolle - Spezifikationen: Die Grundlagen für die erfolgreiche Praxis*. Vieweg, 2003
- [Pos13] *Autoandroid - Positron Framework*. <http://code.google.com/p/autoandroid/wiki/Positron>. Version: Januar 2013
- [Raj12] RAJARAM, Rajesh: *Android Lifecycle*. <http://i.stack.imgur.com/rVnSi.png>. Version: Dezember 2012
- [Rls13] *Pivotal Labs - Robolectric Eclipse Quick Start*. <http://pivotal.github.com/robolectric/eclipse-quick-start.html>. Version: Januar 2013
- [Rol13] *Pivotal Labs - Robolectric Framework*. <http://pivotal.github.com/robolectric/>. Version: Januar 2013
- [Rot13] *Robotium Framework*. <https://code.google.com/p/robotium/>. Version: Januar 2013
- [Sen13] *Sensor Simulator*. <http://code.google.com/p/openintents/wiki/SensorSimulator>. Version: Juli 2013
- [SL10] SPILLNER, Andreas ; LINZ, Tilo: *Basiswissen Softwaretest*. dpunkt Verlag, 2010
- [Sol13] SOLUTIONS, ESE Mobile S.: *SDK Controller*. <https://play.google.com/store/apps/details?id=com.esemobilesoftware.android.tools.sdkcontroller>. Version: Juli 2013
- [Wet07] WETZLMAIER, Thomas: *Systematische Testfallgenerierung für den Black-Box-Test*. GRIN Verlag, 2007

[Win13] WIND RIVER: *Framework for Automated Software Testing*. [http://windriver.com.cn/products/product-notes/PN\\_FAST\\_Testing\\_0310.pdf](http://windriver.com.cn/products/product-notes/PN_FAST_Testing_0310.pdf).  
Version: März 2013

[Yag13] YAGHMOUR, Karim: *Embedded Android*. O'Reilly Media, 2013

[Zak13] ZAKI: *SDK Controller Sensor*. <https://play.google.com/store/apps/details?id=org.zakky.tools.sdkcontroller.sdkcontrollersensor&hl=en>. Version: Juli 2013

## A. Beigefügte DVD

Der Inhalt der beigefügten DVD ist in Abbildung .1 in Form einer Ordnerstruktur zu sehen.

Name	Änderungsdatum	Größe	Art
Abstract.pdf	Gestern 20:27	81 KB	PDF (Portable Document Format)
Literaturquellen	Heute 09:46	--	Ordner
Masterthesis.pdf	Heute 12:05	3 MB	PDF (Portable Document Format)
Zomby-Framework	Heute 14:34	--	Ordner
demo	Heute 15:16	--	Ordner
Zomby simulate ball maze player (720p)	29.07.2013 20:51	59,8 MB	MPEG-4 File
Zomby simulate train journey (720p)	29.07.2013 20:53	66,8 MB	MPEG-4 File
Zomby simulate walking pedestrian (720p)	27.07.2013 14:08	29,9 MB	MPEG-4 File
Zomby simulate sensor values (720p)	31.07.2013 15:03	23,3 MB	MPEG-4 File
examples	Heute 09:43	--	Ordner
DemoApp	27.07.2013 17:39	--	Ordner
ExampleRobotiumTest	06.08.2013 16:57	--	Ordner
ExampleUiAutomatorTest	06.08.2013 14:18	--	Ordner
License	22.07.2013 14:17	9 KB	Reines Textdokument
Lies mich	05.08.2013 12:04	681 Byte	Reines Textdokument
Read me	05.08.2013 12:04	622 Byte	Reines Textdokument
server	Heute 09:43	--	Ordner
ZombyWebservice	03.08.2013 09:47	--	Ordner
test	Heute 09:43	--	Ordner
ZombyTest	06.08.2013 16:59	--	Ordner
Zomby	03.08.2013 09:57	--	Ordner
ZombyRecorder	03.08.2013 09:57	--	Ordner

Abbildung .1: Ordnerstruktur der beigefügten DVD

Auf oberster Ebene befinden sich zwei Dateien und zwei Ordner. Bei den beiden Dateien handelt es sich um die digitale Form der Arbeit sowie eine entsprechende Zusammenfassung (Abstract). Im ersten Ordner „Literaturquellen“ befinden sich Kopien von hier verwendeten Onlinequellen.

Der zweite Ordner „Zomby-Framework“ beinhaltet neben dem eigentlichen Framework auch noch das passende Android-Projekt des ZombyRecorders. In weiteren Unterordnern befinden sich noch weitere Projekte, wie der „ZombyWebservice“ oder das Testprojekt namens „ZombyTest“. Neben Demovideos und Testbeispiel-Projekten ist nur noch die Lizenzmodeldatei und die beiden Anleitungsdokumente „Readme“ und „Lies mich“ enthalten.

# Erklärung

Name:     Manuel Schmidt

Adresse:   Europaring 86  
              53757 Sankt Augustin

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Diese Arbeit wurde bisher weder einer Prüfungsbehörde vorgelegt noch veröffentlicht.

**Ort, Datum:** Sankt Augustin, 13. August 2013

**Unterschrift:** \_\_\_\_\_