Softwaretechnologische Untersuchungen mit dem Anwendungsfall Sensorik von Android-Mobiltelefonen

Masterarbeit

verfasst von

Stefan Hohenwarter BSc.

Institut für Softwaretechnologie

der Technischen Universität Graz

Leiter: Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

Betreuer: Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

Graz, im September 2013

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..

(Unterschrift)

GLEICHHEITSGRUNDSATZ

Aus Gründen der Lesbarkeit wurde in dieser Arbeit darauf verzichtet, geschlechtsspezifische Formulierungen zu verwenden. Ich möchte ausdrücklich festhalten, dass die bei Personen verwendeten maskulinen Formen für beide Geschlechter zu verstehen sind.

DANKSAGUNG

Ich möchte mich an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben. Vor allem gilt dieser Dank meinen Eltern Ernst und Brigitte Hohenwarter, die mir das Studium ermöglicht und mich stets nicht nur finanziell, sondern auch emotional unterstützt haben. Weiters möchte ich mich bei meiner Schwester Sabine Hohenwarter bedanken, die mit ihrem Antrieb bei ihrem Studium immer ein Vorbild für mich war.

Meinem Betreuer Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany möchte ich dafür danken, dass er mich mit einem kreativen und zukunftsträchtigen Thema betraut und es mir ohne Druck zu machen ermöglicht hat, diese Masterarbeit neben meiner beruflichen Vollzeitbeschäftigung fertigzustellen.

Abschließend gilt besonderer Dank meiner Freundin Natalie Lamprecht für das Lektorat, ihren unermüdlichen Beistand und ihre mich – vor allem in der finalen Phase meiner Masterarbeit – motivierenden Worte.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

1 Kurzbeschreibung

In der folgenden Masterarbeit soll die Möglichkeit der Sensordateneinbindung in einem Projekt, das den Namen Catroid (ursprünglich Scratch 2 Android) trägt, geprüft werden. Catroid ist eine Adaption des LEGO-artigen Programmierprogramms Scratch, das vom MIT entwickelt wurde [Sc13]. Dieses Programmier-Framework ermöglicht es auch Kindern und nicht technikaffinen Menschen, kreative Geschichten, Spiele, Rätsel und vieles mehr zu erstellen. Wie der anfängliche Name des Projekts (Scratch 2 Android) vermuten lässt, wurde zu Beginn an einer Android-Umsetzung von Scratch gearbeitet. Im Zuge der Entwicklung sollten dem Programmier-Framework auch Sensoren und Effektoren zugänglich gemacht werden. Ob dies möglich ist und wie eine Einbindung vonstattengehen kann, soll, wie eingangs erwähnt, in dieser Masterarbeit untersucht werden. Weiters wird analysiert, ob man die Sensordaten von Smartphones für andere Zwecke – wie zum Beispiel verschiedene Arten von Tests (z. B. Funktions-, Unit- oder UI-Tests) oder illegale Attacken – verwenden kann. Aktuelle Smartphones verfügen über genügend Rechenleistung, Speicher, Akkulaufzeit sowie eine Vielzahl von Sensoren, wie zum Beispiel Temperaturfühler, Lautstärkenmesser sowie Magnetfeld-, Gyro- und Beschleunigungssensoren. Warum sollte man also die Sensorplattform Smartphone nicht als solche nutzen?

2 Ziele der Masterarbeit

In erster Linie sollte während des gesamten Projektverlaufs der Extreme-Programming-Ansatz angewendet und im Theorieteil der Masterarbeit aufgearbeitet werden. Dabei sollte besonders viel Wert auf die Dokumentation von Tests sowie auf die testgetriebene Entwicklung gelegt und dieses Projekt auf Android-Mobiltelefonen – als Entwicklungsgerät diente ein von der Technischen Universität Graz zur Verfügung gestelltes HTC Desire Smartphone – realisiert werden.

Weiters sollte die Einbindung von möglichst vielen nützlichen Input-Daten für das Programmierprogramm Catroid untersucht werden. Beispiele hierfür sind Keyboard- oder Touch-Eingaben, Bewegungsdaten (Orientation und Accelerometer), Mikrofon, Kamera oder auf der Effektorenseite der Lautsprecher, der Vibrator sowie diverse Lichter (Blitz der Kamera).

Außerdem sollte die Genauigkeit der Sensordaten anhand von Praxistests analysiert und die Verwendung dieser Sensordaten für funktionale Tests untersucht werden. Neben dem Testaspekt sollte zudem analysiert werden, ob es noch andere Einsatzmöglichkeiten für die Sensordaten von Mobiltelefonen gibt.

Vereinbart wurden 900 Stunden (30 ECTS à 30 Stunden), die für die Abschlussarbeit aufgebracht werden sollten, sowie ein zu jedem Projektzeitpunkt verwertbares Endergebnis. Damit die Arbeitszeit nachvollziehbar ist, wurde eine Stundenliste mit sämtlichen Tätigkeiten erstellt und ständig aktualisiert.

3 Motivation

Smartphones sind mittlerweile alles andere als ein Luxusartikel, sie sind ein essenzieller Bestandteil des Alltags. Ganz egal, welche Alters- oder Bevölkerungsgruppen man betrachtet, das Mobiltelefon ist allgegenwärtig. Daher bietet es eine weitverbreitete Plattform für die Scratch- beziehungsweise Catroid-Umsetzung.

In erster Linie wurde mit dem Catroid-Projekt versucht, das LEGO-artige Programmierprogramm Scratch auf den Android-Mobiltelefonen nutzbar zu machen. Doch auch abseits der Android-Smartphones wurde und wird beispielsweise an Portierungen für den Nintendo DS, für iOS und das Windows Phone gearbeitet. Durch diese Umsetzungen soll es Menschen jeden Alters ermöglicht werden, ohne die Anschaffung von teuren Geräten oder Know-how jederzeit und überall, einfach und schnell Computerprogramme, Spiele oder interaktive Geschichten zu entwickeln. Der Kreativität sind dabei keine Grenzen gesetzt.

Es ist spannend zu sehen, wie kreativ vor allem Kinder, aber auch nicht technikaffine Menschen sind, wenn die teils komplizierte und abschreckende Programmierbarriere wegfällt. Neben der Catroid-Entwicklung ist die Sensorik auf Smartphones ein äußerst spannendes Thema. Smartphones sind treue Wegbegleiter vieler Menschen und mit der Rechenleistung, dem Speicherplatz sowie der permanenten Konnektivität eine weitverbreitete Sensorplattform, die das Alltagsleben entscheidend erleichtern kann. So könnte das Smartphone als Kalorienzähler, als Navigationshilfe für Blinde, aber auch für robustere Funktionstests verwendet werden. Dem Ideenreichtum sind hier kaum Grenzen gesetzt, und das Smartphone wird in Zukunft wohl noch mehr in den Alltag der Anwender integriert werden und eventuell als Zahlungsdevice oder Gesundheitsdatenspeicher fungieren.

4 Die Geschichte von Catroid

Im Projekt Catroid, bei dem zu Beginn rund zehn Leute beschäftigt waren, sollte das Programmierprogramm Scratch, das vom MIT entwickelt wurde, auf Android-Mobiltelefone portiert werden. Der große Vorteil dieses Programmierprogramms ist es, dass es LEGO-artig aufgebaut ist und somit auch Kinder, die keinen Bezug zum Programmieren haben, schnell und einfach ein Spiel, eine Applikation oder eine interaktive Geschichte entwickeln können.

Während sich ein Teil des Projektteams mit der Portierung von Scratch auf die Android-Mobiltelefone befasste, kümmerte sich ein Mitarbeiter um den Aufbau einer Web-2.0-Communityseite [Ca12], auf der Erfahrungen und Programme ausgetauscht werden können. Im späteren Entwicklungsverlauf wurde der Name von Scratch 2 Android auf Catroid geändert. Ich selbst befasste mich in den Anfängen der Catroid-Entwicklung mit der Untersuchung, wie und welche Sensoren beziehungsweise Effektoren dem Programmierprogramm zugänglich gemacht werden können. Diese Sensoren/Effektoren sollten in weiterer Folge in dem Baukasten-Programmierprogramm Catroid verwendet werden können, um dem User noch mehr Möglichkeiten zum Ausleben der Kreativität zu bieten. Schnell wurde das enorme Potenzial von Catroid entdeckt. Daher wurde auch eine Vielzahl neuer Mitarbeiter rekrutiert, um an möglichst vielen Features gleichzeitig arbeiten zu können.

Damit das Team, das aus externen und internen Mitarbeitern besteht, ständig über die aktuellen Tasks und Fortschritte informiert ist, musste ein geeignetes Dokumentationstool gefunden werden. Nach reiflicher Überlegung und Beratung fiel die Wahl auf Google Wave, da es sich am Besten für die Kooperation mit den restlichen Teammitgliedern eignet. Nach einer Testphase wurden die Grenzen und gebotenen Features analysiert, und das Tool wurde für die Onlinedokumentation verwendet. Aufgrund der Einstellung des Google Wave Supports wurde im Verlauf der Catroid-Entwicklung ein eigenes Wiki-System für die Dokumentation aufgebaut. Darin wurden und werden alle wichtigen Erkenntnisse sowie Ideen für neue und bestehende Projektmitarbeiter gesammelt und aktualisiert.

Durch das Engagement des gesamten Teams wurde das Catroid-Projekt bereits zweimal in Folge als „Google Summer of Code“-Projekt (2011 und 2012) ausgewählt. Es handelt sich dabei um ein von Google organisiertes jährliches Programmierstipendium für Studenten, die an Open-Source-Projekten arbeiten (Debüt 2005). Dass das Catroid-Projekt eines der von

Google ausgewählten 175 Projekte war, die im Zuge des „Google Summer of Code“ gefördert wurden, zeigt das Potenzial des Projekts. Mittlerweile wird an weiteren Portierungen, einem Übersetzungs-Support-System, einem YouTube-Aufnahme-Feature für die Catroid-Stage und vielem mehr gearbeitet, um noch mehr Leute für die visuelle Programmiersprache Catrobat zu begeistern.

Doch auch für die Zukunft gibt es schon einige Ideen, die man im Catrobat-Team umsetzen möchte. So sollen unter anderem ein multilinguales Wiki und auch ein Forum für Catroid-Fans, eine 3D-Version sowie Designs und Konfigurationen für verschiedene Altersgruppen und Geschlechter entstehen.

5 Theorieteil

Im Theorieteil werden die beiden Programmiertechniken „Extreme Programming“ und „testgetriebene Entwicklung“ erklärt und die wichtigsten Kernelemente vorgestellt.

5.1 Extreme Programming

Da schon bei den Zielen des Projektes das Extreme-Programming-Konzept erwähnt wurde, werden im folgenden Abschnitt die wichtigsten Aspekte sowie die Kernelemente des Extreme Programming aufgezeigt und der Begriff erklärt.

5.1.1 Was ist Extreme Programming?

Wie in [Ju00] zu lesen ist, versteht man unter Extreme Programming einen neuen Weg der Software-Entwicklung, der für kleine Teams von zwei bis zehn Programmierern entwickelt wurde. Laut [Ju00] ist Extreme Programming eine leichtgewichtige Entwicklungsmethode, die effizient, risikoarm, flexibel, vorhersehbar und überdies wissenschaftlich ist.

Einer der bekanntesten Begründer dieses Konzepts ist Kent Beck, der Extreme Programming mit der Veröffentlichung seines Buches „Extreme Programming Explained“ [Be99] erst salonfähig machte. Im Gegensatz zu veralteten und statischen Entwicklungsmethoden spricht man beim Extreme Programming von einer agilen Software-Entwicklungsmethode [Ab03]. Die Agilität entsteht durch die Verwendung von gewissen Praktiken, die sich bis heute in der Praxis bewähren. Anstatt die Zeit mit Dokumentation von Code oder dem strikten Einhalten von Prozessabläufen zu vergeuden, stehen beim Extreme Programming das Vorgehen in kleinen Iterationsschritten, die Kommunikation sowie die Teamarbeit im Vordergrund. Wie das Konzept genau angewandt wird, erklären die folgenden Abschnitte.

5.1.2 Das Konzept

Das Erfolgsrezept des Extreme-Programming-Konzepts ist schnell erklärt: Durch die im Abschnitt 5.1.3 Extreme-Programming-Praktiken erwähnten Maßnahmen wird zum einen die Qualität der Software und zum anderen die Flexibilität enorm gesteigert. Man kann schnell auf Modifikationen im Anforderungsprofil reagieren, wodurch die Änderungskurve über die

gesamte Projektdauer annähernd linear bleibt. Genauer gesagt konvergiert sie gegen einen Kostenwert, der im Gegensatz zu einer konventionellen Software-Entwicklungsmethode fast über die gesamte Projektdauer „günstiger“ ist. Diese Änderungskurve kann in Abbildung 1: Extreme Programming – XP-Entwicklungskurve in Anlehnung an [Ru01] genauer betrachtet werden.

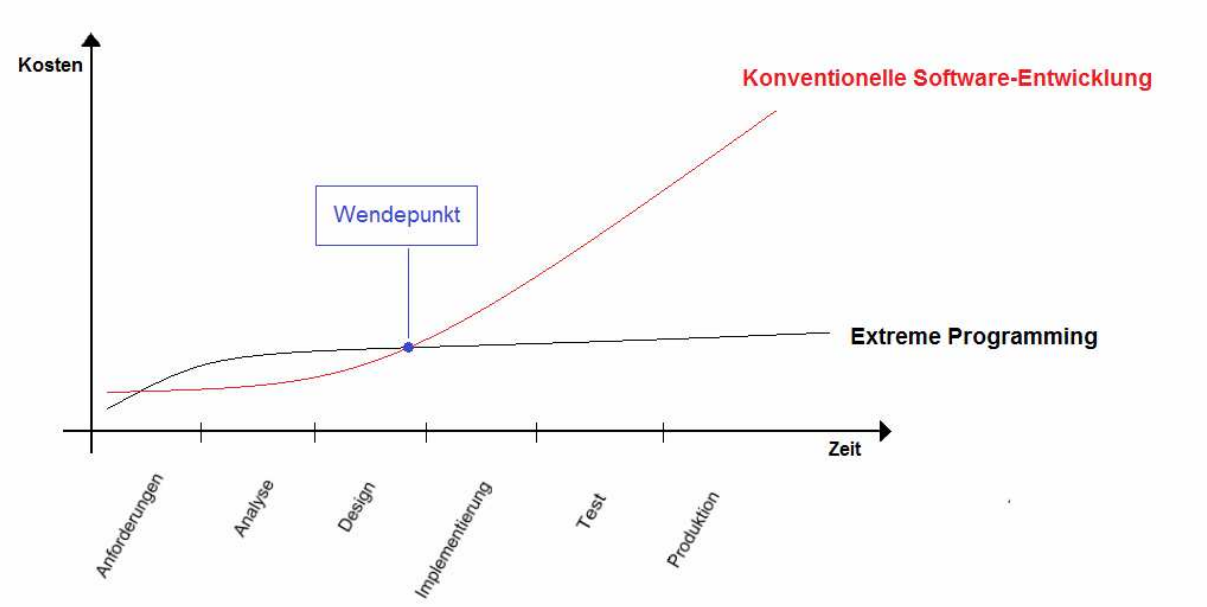

Abbildung 1: Extreme Programming – XP-Entwicklungskurve in Anlehnung an [Ru01]

Zwar zeigt die konventionelle Entwicklungskurve bis zu einem gewissen Wendepunkt (vgl. Abbildung 1: Extreme Programming – XP-Entwicklungskurve in Anlehnung an [Ru01]) bezüglich der Kosten einen vorteilhafteren Verlauf, doch ist die Differenz hinsichtlich der Extreme-Programming-Kurve nahezu vernachlässigbar. Im Gegensatz dazu steigt der Kostenunterschied dieser beiden Kurven ab dem eingezeichneten Wendepunkt exponentiell an. Bei konventionellen Entwicklungsmethoden werden die Probleme oder Fehler meistens erst sehr spät erkannt, womit auch ein enormer Kostenaufwand verbunden ist. Im schlechtesten Fall entspricht das gelieferte Produkt nicht den Vorstellungen des Kunden, und die damit verbundenen Kosten für Änderungen und Adaptionen sind meist enorm.

Dank der zuverlässigen Praktiken des Extreme Programming, die das Einmaleins dieser agilen Softwareentwicklungsmethode bilden, konvergiert die Änderungskurve hinsichtlich der Kosten gegen einen Wert, der über die gesamte Projektdauer nahezu konstant bleibt. Warum das so ist, wird aus der Erklärung der etablierten Praktiken ersichtlich.

5.1.3 Extreme-Programming-Praktiken

Wie bereits dargelegt, basiert das Extreme-Programming-Konzept auf einer Reihe von Methoden, die sich in der Praxis bis heute als robust und zielführend erwiesen haben. Laut [Ju00] und [Be99] gibt es beim Extreme Programming die folgenden traditionellen Praktiken:

- Kurze Iterationen
- The Planning Game
- Pair Programming
- Einfaches Design
- Coding Standards
- Metapher
- 40-Stunden-Woche
- Refactoring
- Kundeneinbeziehung
- Testgetriebene Entwicklung bzw. permanentes Testen
- Permanente Integration
- Kollektives Eigentum

Nachdem nun ein Überblick über die traditionellen Praktiken präsentiert wurde, sollen diese auf den folgenden Seiten im Detail betrachtet werden.

5.1.3.1 Kurze Iterationen

Darunter versteht man eine Methode, die besagt, dass man möglichst kurze „Veröffentlichungszeiträume“ wählt. Zu Beginn eines Projekts beträgt dieser Zeitraum laut [Hu08] rund einen Monat. Dadurch wird schon im frühen Entwicklungsstadium überprüft, ob die Entwicklung den Wünschen des Kunden entspricht. Nach jedem Abschluss einer Iteration werden die entwickelten Features vom Kunden, der im Idealfall zugleich auch Anwender der Software ist, getestet. Werden die Tests bestanden, folgt die nächste Iteration. Durch dieses schrittweise Vorgehen kann man zu jedem Zeitpunkt ein Produkt abliefern, das den Vorstellungen des Kunden entspricht. Der Zeitraum für diese Iterationen wird in weiterer Folge von einem Monat auf rund drei Monate erweitert, um den administrativen Aufwand für die Iterationsplanungsmeetings möglichst gering zu halten.

5.1.3.2 The Planning Game

Prägnant könnte diese Methodik, wie auch in [Mu01] beschrieben wird, wie folgt zusammengefasst werden: Unter dem sogenannten Planning Game versteht man die Festlegung der Anforderungen und die Reihenfolge der Feature-Entwicklungen in inkrementellen Abschätzungen. Oft wird dieser Vorgang auch mit einem Pokerspiel verglichen, bei dem sich der Kunde und das Entwicklerteam auf die gemeinsame Vorgangsweise einigen.

5.1.3.3 Pair Programming

Darunter versteht man, dass jeweils zwei Programmierer auf einem Computer kooperativ bzw. paarweise zusammenarbeiten. Derjenige, der am Computer sitzt und programmiert, wird als Fahrer bezeichnet. Der zweite Programmierer ist der sogenannte „Kopf“ und unterstützt den Fahrer als Kontrolleinheit. Unter dem Motto „Vier Augen sehen mehr als zwei“ soll vor allem der Programmierer im Hintergrund einen Blick auf die Semantik und die Syntax des Codes werfen, um Fehler schnell zu erkennen und diesen vorzubeugen. Weiters hat der zweite Programmierer die Aufgabe, den gerade programmierenden Mitarbeiter vor Ablenkungen zu schützen. Sollte beispielsweise das Telefon läuten, kümmert sich der zweite Mitarbeiter um dieses Telefonat. So bleibt der Fahrer stets konzentriert und fokussiert. Durch die ständige Kontrolle des Kopfes ist die Hemmschwelle des Fahrers auch höher, sich ablenken zu lassen (zum Beispiel durch das Lesen privater E-Mails, Schreiben einer SMS oder Blättern in der Zeitung). Zwar verliert man dabei auf den ersten Blick etwas Produktivität, da statt zwei Programmierern nur einer arbeitet, doch hat sich in der Praxis herausgestellt, dass man durch das Pair Programming in puncto Qualität einen entscheidenden Vorteil hat. Wie auch eine Studie mit Studenten in [Xu05] beweist, lieferten die Studenten, die in Paaren gearbeitet hatten, ein qualitativ höherwertiges Produkt ab als jene, die bei der Problemlösung auf sich allein gestellt gewesen waren.

Durch die ständige Verbesserung des Codes und durch die Kommunikation des Programmierpaares wird ein hochwertiges Ergebnis erzeugt. Zusätzlich kann dadurch beispielsweise ein Anfänger oder ein neuer Mitarbeiter behutsam an das Projekt herangeführt werden. Damit auch beide Programmierer etwas von dieser Vorgangsweise haben und der Fahrer stets voll konzentriert bleibt, wechseln sich die Paare bei der Programmierung sowie bei der Kontrolle ab. Durch diesen Informationstransfer wird auch der sogenannte „Truck

Faktor“ verbessert. Dies ist ein von Kent Beck vorgeschlagener Wert, der die Wahrscheinlichkeit des Scheiterns des Projekts beschreibt, sollte ein Mitarbeiter von einem Truck überfahren werden, also eine unvorhersehbare Situation eintreten, die den Verlust eines Mitarbeiters für das Projekt bedeutet.

Ein kurzer Blick auf ein praktisches Beispiel verdeutlicht die Bedeutung dieses Wertes. Sollte beispielsweise ein einzelner Programmierer das gesamte Know-how des Projektes verinnerlicht haben und „von einem Truck überfahren“ werden, dann ist die Wahrscheinlichkeit, dass das Projekt scheitert, enorm. Wenn jedoch das Wissen über das Projekt auf mehrere Personen verteilt ist (kollektives Eigentum), kann der Ausfall (zum Beispiel auch durch Kündigung) eines Mitarbeiters viel leichter verkraftet werden.

5.1.3.4 Einfaches Design

Wie der Name vermuten lässt, wird beim Extreme Programming angestrebt, einen möglichst leichtgewichtigen Code zu produzieren. Das bedeutet, dass wirklich nur der benötigte Code implementiert wird. Es wird auf keinen Fall Code geschrieben, der eventuell in der Zukunft gebraucht werden könnte. Nur Code, der für die Erfüllung der jeweiligen Iteration benötigt wird, soll auch implementiert werden. Weiters wird auch bei der Strukturierung des Codes auf ein einfaches Design Wert gelegt. Unnütze Klassen, Methoden oder Abfragen werden entfernt, um die Les- und Wartbarkeit zu verbessern. Durch das „Einfachhalten“ des Codes kann man auch viel flexibler auf eventuelle Änderungen des Kunden reagieren.

5.1.3.5 Coding Standards

Jeder Programmierer hat einen eigenen Stil. Das ist an sich nicht negativ, doch wenn viele Programmierer an einem Projekt arbeiten und jeder seinen eigenen Stil mitbringt, kann dies leicht zu Problemen führen. Aus diesem Grund sollte man sich auf gemeinsame Coding Standards einigen. Dann fällt auch der Wechsel der Programmierpaare untereinander viel leichter, da alle dieselben Standards verwenden. Dadurch erzielt man wiederum eine höhere Wart- und Lesbarkeit des Codes.

5.1.3.6 Metapher

Die Metapher dient vor allem zur Beseitigung der Sprachbarriere zwischen den Entwicklern und den Kunden. Beck erklärt die Metapher beim Extreme Programming in [Be99] wie folgt: Die Metapher hilft allen am Projekt Beteiligten, die Basiselemente und ihre Beziehungen zu verstehen. Dafür werden ausschließlich Wörter von der gewählten Metapher verwendet, um die technischen Dinge zu beschreiben.

Da beide Parteien meistens vom Fach sind und aus verschiedenen Geschäftsbereichen stammen, besitzen sie ein eigenes Vokabular. Damit es nicht zu Verständigungsproblemen kommt, wird von besagten Metaphern Gebrauch gemacht. Ein Beispiel für diese Metaphern wären sogenannte User Stories oder Story Cards.

Mit diesen Story Cards wird der Funktionsumfang der Software abgegrenzt und klar definiert. Typischerweise umfasst eine Story Card rund drei bis vier Sätze, die in groben Zügen die Funktionalität für den Endbenutzer skizzieren. Ein Beispiel hierfür wäre: „Ich möchte alle Kunden sehen, die heute Geburtstag und in den letzten beiden Monaten x Produkte bestellt haben.“ Somit ist für den Entwickler die Aufgabe klar definiert, und es kann eine Aufwandschätzung erfolgen. Da der Entwickler den Umfang und die Komplexität der Software abschätzen kann, definiert er nun die Implementierungszeitrahmen für diese Story Card. Im Schnitt kann man sagen, dass die Erfüllung einer Story Card zwischen einer und drei Wochen dauert. Wichtig dabei ist die ständige Kommunikation mit dem Kunden, um zu gewährleisten, dass die entwickelte Software wirklich den Ansprüchen des Kunden gerecht wird. Durch die fortlaufende Kommunikation und das Feedback des Kunden kann schnell auf Änderungen der Prioritäten oder des Funktionsumfangs reagiert werden.

5.1.3.7 40-Stunden-Woche

Wie der Name schon vermuten lässt, spiegelt die 40-Stunden-Woche den idealen Arbeitszeitraum eines Programmierers wider. Dies bedeutet, dass man auf Überstunden so weit wie möglich verzichten sollte, da die Konzentrationsfähigkeit eines Entwicklers bei Überarbeitung stark sinkt. Meistens sind Überstunden auf falsche oder schlechte Planung zurückzuführen, die durch Mehrarbeit im Normalfall nicht verbessert werden kann, da die Produktivität mit jeder Überstunde weiter zurückgeht.

5.1.3.8 Refactoring

Unter Refactoring versteht man das nachträgliche Überarbeiten des Codes. Das bedeutet, dass man in erster Linie einen funktionierenden Code erzeugt, der dann durch einen Reviewprozess hinsichtlich Architektur, Design und Komplexität überarbeitet wird. Ein Grund, warum die Änderungskurve bei Extreme-Programming-Projekten annähernd konstant verläuft, ist das Refactoring, da man den Code stets verbessert.

5.1.3.9 Kundeneinbeziehung

Wie bereits bei der Metapher erwähnt, sollte der Kunde so weit wie möglich ins Projekt miteinbezogen werden. Darunter versteht man, dass der Kunde so viel Feedback wie möglich geben sollte. Anhand von vorher definierten User Stories (oder Story Cards) werden die Ziele für jede Iteration klar abgesteckt. Wenn dem Kunden das Iterationsziel präsentiert wird, hat dieser die Möglichkeit, anhand von sogenannten Akzeptanztests die Funktionalität zu überprüfen. Wenn der Kunde zugleich auch Endnutzer der Software ist, spricht man von einem Idealfall, da er die Software sowohl aus der Sicht des Verkäufers als auch der des Endnutzers betrachten kann.

5.1.3.10 Testgetriebene Entwicklung bzw. permanentes Testen

Die testgetriebene Entwicklung ist besser unter dem englischen Begriff Test-Driven-Development bekannt. Dabei steht das permanente Testen im Vordergrund. Man wird dabei gezwungen, zuerst einen Test (meistens Unit-Tests) zu schreiben, bevor die eigentliche Funktionalität implementiert wird. Bei dieser Vorgangsweise muss sich der Entwickler schon vor der eigentlichen Implementierung des Codes Gedanken über die Architektur und das Design machen. Diese Art der Test-Implementierung wird auch als Grey-Box-Test bezeichnet, da der Testersteller zugleich der Entwickler der Funktionalität ist. Wichtig dabei ist, dass zu Beginn nur so viel Code implementiert wird, dass der Code kompiliert werden kann. Die Folge daraus ist, dass der Test fehlschlägt. Erst danach beginnt die richtige Entwicklung des Funktionscodes. Ist dies geschehen, wird der Test erneut ausgeführt. Wird dieser nun bestanden, widmet sich der Entwickler dem nächsten Iterationsschritt. Durch das andauernde Testen wird gewährleistet, dass das komplette System fortwährend auf die Probe gestellt wird. Durch diese Vorgangsweise ist darüber hinaus sichergestellt, dass fehlerhafter

Code jeweils nur in der letzten Iteration entsteht. Damit fällt die Fehlersuche bedeutend leichter.

5.1.3.11 Permanente Integration

Ähnlich wie bei der testgetriebenen Entwicklung wird laut Extreme Programming auf fortwährende Integration von neuen Features Wert gelegt. Dadurch wird einerseits der Integrationsvorgang verinnerlicht, andererseits können Fehler, die im Zuge der Integration aufgetreten sind, sehr gut eingegrenzt werden. Falls Fehler auftreten, müssen sie während der letzten Einpflegung passiert sein, da die Software erst beim Bestehen aller Tests für die Weiterarbeit wieder freigegeben wird. In der Praxis hat sich herausgestellt, dass oft bei der finalen Integration aller Komponenten extreme Fehler auftreten und dadurch Termin- und vor allem auch Kostenprobleme entstehen. Diesem Phänomen kann man mit permanenter Integration leicht entgegenwirken.

5.1.3.12 Kollektives Eigentum

Ein weiterer wichtiger Punkt ist das Wirgefühl. Alle Mitglieder des Teams müssen sich darüber bewusst sein, dass das Projekt nur als Team bewältigt werden kann. Der gesamte erstellte Code ist Eigentum des gesamten Teams, weshalb es auch keine klare Aufgabenverteilung gibt (zum Beispiel kümmert sich Mitarbeiter A um die Klasse X und Mitarbeiter B um die Klasse Y). Jeder arbeitet an allen Fronten mit, um schließlich ein Topprodukt abzuliefern. Dies wird den Teammitgliedern vor allem durch das Pair Programming sowie das Auswechseln der Paare untereinander klargemacht. Durch diese Vorgangsweise wird, wie bereits erwähnt, auch der gefürchtete „Truck Faktor“ sehr gering gehalten und so auch die Erfolgschance des Projekts enorm gesteigert.

Diese traditionellen Praktiken stammen aus den Anfängen des Extreme Programming. In den letzten Jahren wurden sie stets weiterentwickelt und verbessert, weshalb sie in weiterer Folge als evolutionäre Praktiken bezeichnet werden. Beck selbst überarbeitete die traditionellen Praktiken in der zweiten Auflage seines Werkes „Extreme Programming Explained: Embrace Change“ [Be04].

5.1.3.13 Hauptpraktiken (evolutionäre Praktiken)

Wie in [Ei11] erwähnt, werden die evolutionären Praktiken in Hauptpraktiken und Begleitpraktiken unterteilt, die inhaltlich den traditionellen sehr ähnlich sind. Mit der Metapher und dem Coding Standard wurden zwei Praktiken entfernt, teilweise die Namen verändert und manche Praktiken weiter unterteilt:

- Räumlich Zusammensitzen
- Informativer Arbeitsplatz
- Vollständiges Team
- Pair Programming
- Energievolle Arbeit
- Freiraum
- User Stories
- Wöchentlicher Zyklus
- Quartalsweiser Zyklus
- Zehn-Minuten-Build
- Kontinuierliche Integration
- Test-First-Programming
- Inkrementelles Design

Grob zusammengefasst kann man sagen, dass das Wirgefühl noch weiter in den Vordergrund gerückt wurde. Das Teamgefüge soll vor allem durch das Arbeiten aller Teams in einem Raum und die gemeinsame Verantwortung für den entwickelten Code gestärkt werden.

Darüber hinaus werden die User Stories (oder Story Cards) an einem gut sichtbaren Platz im „War Room“ (Raum, in dem die Programmierung vonstattengeht) positioniert. Dadurch haben alle Mitarbeiter stets einen guten Überblick über den aktuellen Stand des Projekts. Neben den Stories, die schon das inkrementelle Vorgehen mit sich bringen, werden die Code-Erstellungen samt Tests in zehnminütigen Schritten vollzogen. Im Idealfall werden die gesammelten Implementierungen rund alle zwei Stunden in das gemeinsame Repository eingepflegt. Die Vorteile dieser fortlaufenden Implementierung und Integration wurden bereits bei den traditionellen Praktiken aufgezeigt.

5.1.3.14 Begleitpraktiken (evolutionäre Praktiken)

Neben diesen Hauptpraktiken kommen laut [Be04] noch die vorher erwähnten Begleitpraktiken zum Einsatz, die sich wie folgt unterteilen:

- Echte Kundeneinbeziehung
- Inkrementelle Verteilung
- Team-Kontinuität
- Schrumpfende Teams
- Ursprungsursachenanalyse
- Geteilter Code
- Codebasis
- Verhandelbarer, vertraglicher Funktionsumfang
- Zahlen pro Nutzung
- Tägliche Verteilung

Da diese Praktiken in groben Zügen schon bei der traditionellen Vorgangsweise erläutert wurden, wird auf eine weitere Erklärung der bereits bekannten Punkte verzichtet. Einzig die Prinzipien Zahlen pro Nutzung und Team-Kontinuität sollen an dieser Stelle noch kurz erklärt werden.

Unter Zahlen pro Nutzung versteht man, wie der Namen schon vermuten lässt, eine Vertragssituation. Der Kunde zahlt hierbei nur für die implementieren Features (User Stories), wobei die Anzahl der bereitgestellten Versionen vernachlässigt wird.

Wie bereits erwähnt, ist das Wirgefühl ein zentraler Punkt beim Extreme Programming. Dies soll vor allem auch durch die Team-Kontinuität weiter gestärkt werden. Durch die Kontinuität wächst das Team immer weiter zusammen und kann sich so immer schwierigeren Aufgaben stellen. Auch Umstrukturierungen (zum Beispiel Versetzung von Mitarbeitern in andere Projektteams) im Unternehmen können dadurch leichter verkraftet werden.

5.1.4 Die Werte des Extreme Programming

Nach den Praktiken wird nun ein Blick auf die Werte des Extreme Programming geworfen, die sich laut [Be99] wie folgt aufschlüsseln:

- Kommunikation
- Einfachheit
- Feedback
- Mut
- Respekt

Diese Werte finden sich auch zum Großteil in den vorher aufgezählten und ausgeführten Praktiken wieder und benötigen deshalb keiner weiteren Erklärung.

5.1.5 Die Variablen eines XP-Projekts

Neben den Prinzipien und Praktiken sind die Variablen eines XP-Projekts ein essenzieller Bestandteil. Drei dieser vier Faktoren (Umfang, Kosten, Qualität und Zeit) kann der Kunde laut [Ei11] bestimmen, während sich der vierte Wert daraus ableiten lässt. Werden beispielsweise die Eckdaten Umfang, Qualität und Zeit verändert, zieht das unmittelbar Folgen bei den Kosten nach sich. Wird die Entwicklungszeit gekürzt und zugleich der Umfang und die Qualität erhöht, steigen damit auch die Kosten, da unweigerlich mehr Entwickler für die neuen Anforderungen rekrutiert werden müssen.

5.1.6 Die Rollen im Extreme Programming

Nachdem in den letzten Absätzen mehrmals über das Verhältnis vom Entwicklerteam zum Kunden gesprochen wurde, zeigt der folgende Abschnitt die einzelnen Rollen samt Tätigkeitsfelder aller Involvierten auf.

Es gibt laut [Sl10] die folgenden fünf Rollen:

- Auftraggeber
- Kunde
- Entwickler

- Coach
- Manager

Jede dieser Rollen nimmt eine bestimmte Position im Entwicklungsprozess ein, die im Anschluss genauer erklärt werden.

5.1.6.1 Auftraggeber

Den Anfang macht der Auftraggeber, der im Normalfall zugleich der Geldgeber ist und in groben Zügen den Umfang definiert. Er ist weiters in das Planning Game involviert und trifft die zentralen Entscheidungen im Projekt.

5.1.6.2 Kunde

Wie bereits mehrmals erwähnt, gehört der Kunde im Idealfall zum Projektteam. Er soll durch das ständige Vorortsein für aufkommende Fragen sowie Feedback bereitstehen. Vor allem durch seine Rückmeldungen übt er großen Einfluss auf die Entwicklung aus.

5.1.6.3 Entwickler

Der Entwickler ist, wie der Name schon vermuten lässt, für die Programmierarbeit zuständig. Weiters ist er für Feedback, Problemlösungen, Tests, Integrationen sowie Refactoring verantwortlich. Wichtig ist dabei vor allem die Zusammenarbeit mit dem restlichen Team. Je motivierter jeder einzelne Entwickler ist, desto besser wird das Gesamtergebnis.

5.1.6.4 Coach

Der Coach übernimmt im Extreme Programming eine ähnliche Rolle wie im Sport: Er schützt das Team vor Hindernissen und Problemen. Weiters sollte er stets einen guten Überblick über das Voranschreiten des Projektes haben und bei Problemen die richtigen Hebel in Bewegung setzen, um das Projekt wieder auf Kurs zu bringen. Dabei soll allerdings nicht die Selbstverantwortung der Entwickler außer Acht gelassen werden.

5.1.6.5 Manager

Der Manager ist das Sprachrohr zur Außenwelt. Er repräsentiert das Team nach außen und erarbeitet Marketingstrategien. Weiters präsentiert er die Fortschritte des Teams firmenintern sowie -extern und gibt stets Feedback, ob der Projektfortschritt den Einschätzungen entspricht.

5.1.7 Persönliche Eindrücke vom Extreme Programming

Die Ideen und Praktiken bergen gute Ansätze. Auch in der Praxis findet das Extreme Programming immer mehr Einsatz, da auch der Erfolg dieser agilen Entwicklungsmethode erwiesen ist.

Das einzige Manko ist vielleicht, dass kleine Firmen Bedenken wegen der eventuellen Kosten haben könnten – auf den ersten Blick scheint es ein Nachteil zu sein, zwei Leute an der Umsetzung einer Lösung arbeiten zu lassen und damit auch doppelt bezahlen zu müssen. Dass dadurch aber die Zeit für das Lösen des Problems verringert und auch die Fehlerquote erheblich gesenkt wird, wird oft übersehen.

Weiters ist es recht schwierig, das Pair Programming einzusetzen, wenn zu wenige Senior-Programmierer im Projektteam sind, da immer nur ein Neueinsteiger vom Senior-Programmierer lernen kann, um in weiterer Folge sein Wissen auch weitergeben zu können. Der Idealfall vom Kunden, der dem Projektteam angehört, ist in der Praxis auch kaum vorhanden.

Zusammenfassend lässt sich sagen, dass wenn das Unternehmen und die Mitarbeiter offen sind und dem Extreme-Programming-Konzept die nötige Zeit geben, die Qualität der Arbeit definitiv erhöht wird. Das Betriebsklima profitiert davon, und auch die Quote für das Scheitern von Projekten sinkt. Durch all diese Vorteile spart sich das Unternehmen in weitere Folge Kosten, was wohl ein ausschlaggebender Aspekt für die Anwendung dieses Ansatzes ist.

5.2 Testgetriebene Entwicklung

Neben dem Extreme-Programming-Ansatz ist auch die testgetriebene Entwicklung ein Kernelement dieser Masterarbeit. Wie bereits im Abschnitt Extreme Programming gezeigt, ist das permanente Testen ein essenzieller Bestandteil der agilen Software-Entwicklungsmethode. Aus diesem Grund erhält dieses Thema einen eigenen Abschnitt in dieser Masterarbeit.

5.2.1 Definition

Unter testgetriebener Entwicklung (Englisch: „test first development“ oder „test-driven development“) versteht man eine Methode, die häufig bei der agilen Entwicklung von Computerprogrammen Anwendung findet und den Entwickler dazu „zwingt“, sich zuerst mit dem Problem auseinanderzusetzen, in weiterer Folge einen Testfall zu schreiben und erst dann den entsprechenden Funktionscode zu erstellen. Deshalb spricht man hier auch nicht von Black- oder White-, sondern Grey-Box-Tests.

5.2.2 Motivation

Der Zweck der testgetriebenen Entwicklung ist die Reduktion der Kosten, die durch entstandene Fehler und die damit verbundene Fehlersuche und -behebung entstehen. Je weiter der Entwicklungsprozess vorangeschritten ist, desto schwieriger und teurer wird die Fehlersuche im Allgemeinen. Immerhin müssen die Entwickler das Problem erst identifizieren, bevor sie es beheben können. Durch Tests lassen sich diese Fehler minimieren oder sogar ganz ausmerzen.

Im Gegensatz zu traditionellen Entwicklungsmethoden wird die Testphase beim Extreme Programming nicht am Projektende durchgeführt, sondern während des gesamten Entwicklungsprozesses. Das hat den Vorteil, dass man Fehler sofort identifizieren und zuordnen kann, anstatt am Projektende auf eine langwierige und kostenintensive Fehlersuche gehen zu müssen. Dadurch bleibt die Kostenkurve für die Beseitigung der Fehler über den gesamten Projekt- bzw. Entwicklungszeitraum mehr oder weniger konstant, während sie bei traditionellen Methoden mit der Zeit exponentiell ansteigt (vgl. Abbildung 2).

5.2.3 Gründe für testgetriebene Entwicklung

Wie in [Bi10] beschrieben, gibt es zehn gute Gründe, warum die testgetriebene Entwicklung eingesetzt werden sollte:

- Wenn man keinen Testfall schreiben kann, versteht man den zu implementierenden Funktionscode nicht.
- Ohne schnelle Tests ist es schwierig, den Code „aufzuräumen“.
- Schnelle Feedbackzeiten sparen Geld und Zeit.
- Testen ist essenziell – es ist besser, mit der gebotenen Zeit effizient umzugehen.
- Ohne ausführliche Tests benötigt man für die Weiterentwicklung viel Glück oder eine umfassende Dokumentation.
- Evolutionäres Design ist möglich.
- Es muss weniger Code geschrieben werden.
- „Bugfreie“ Entwicklung sichert Aufträge und Arbeitsplätze.
- Erfolgreiche Tests motivieren.
- Testgetriebene Entwicklung kombiniert mit Akzeptanztests liefert ein optimales Ergebnis.

5.2.4 Ablauf

Nachdem die testgetriebene Entwicklung ausführlich erklärt wurde, soll nun der Ablauf der testgetriebenen Entwicklung in einem Projekt aufgezeigt werden.

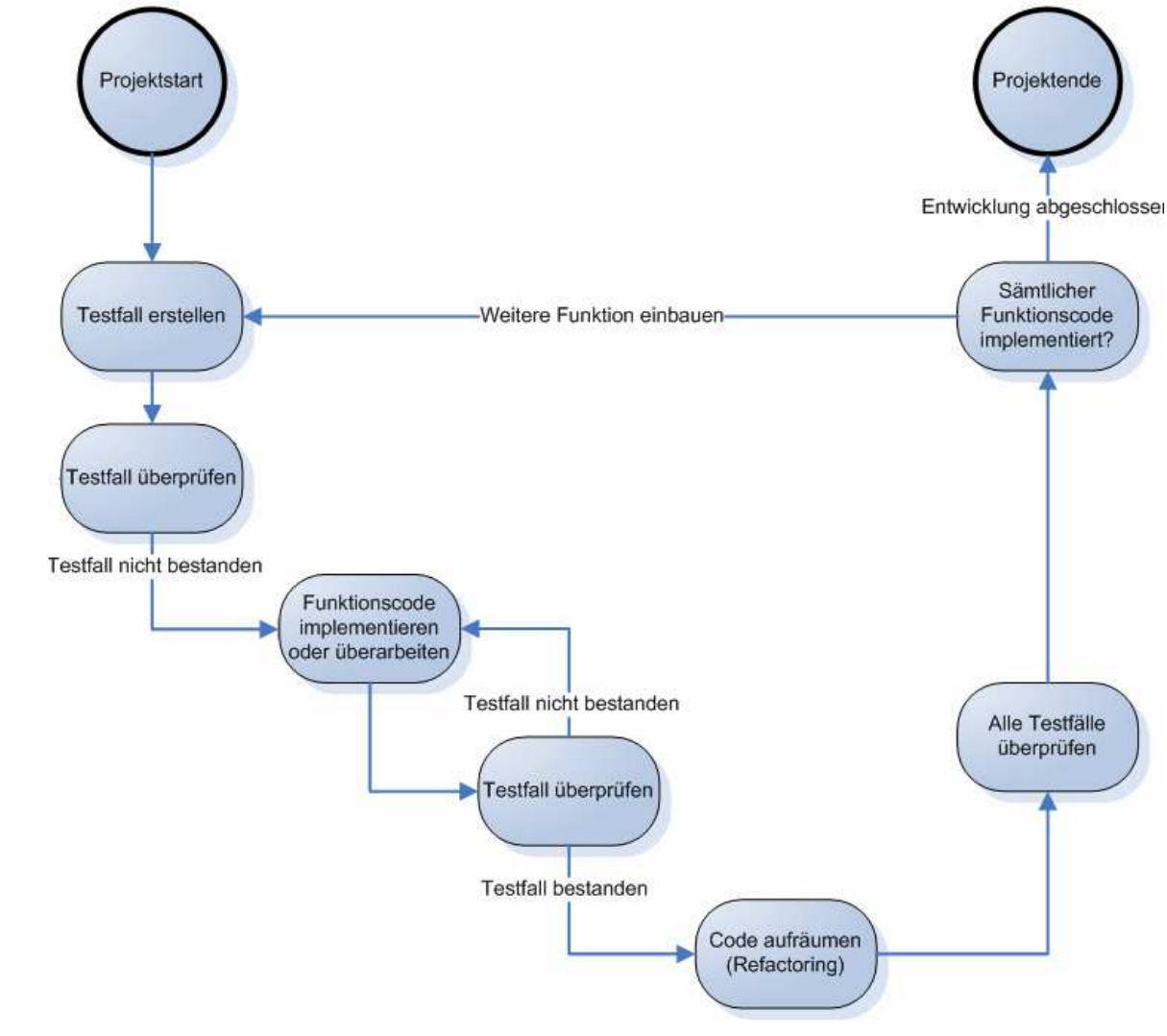

Abbildung 2: Testgetriebene Entwicklung – Workflow (eigene Abbildung)

Wie man in der Abbildung sehen kann, steht der Projektstart am Beginn der Entwicklungsarbeit. Bevor ein Funktionscode implementiert wird, muss ein Testfall geschrieben werden, der die noch nicht implementierte Funktionalität überprüft. Der Testfall kann praktisch nicht bestanden werden, da der entsprechende Code noch nicht implementiert wurde. Aus diesem Grund wurde die Option, dass der Testfall bestanden wird, weggelassen, da in diesem Fall der Testfall nicht richtig geschrieben wurde und der Test definitiv fehlschlägt.

Im nächsten Schritt muss der Funktionscode überarbeitet beziehungsweise erst erstellt werden. Wenn man damit fertig ist, wird der Testfall ein weiteres Mal durchlaufen. Danach folgt das Refactoring, bei dem unnötiger Code entfernt und die Wart- und Lesbarkeit erhöht wird. Sobald der Code aufgeräumt ist, werden zur Sicherheit noch einmal alle Testfälle

durchlaufen, um stets einen funktionierenden und ausgiebig getesteten Prototyp zur Verfügung stellen zu können. Nur wenn alle Testfälle mit Erfolg bestanden werden, kann man den nächsten Iterationsschritt vornehmen. Wie in [Ma07] erwähnt, sollte ein Iterationsschritt rund zwei Minuten dauern. Dieses Intervall lässt sich allerdings nur einhalten, wenn der Code nicht auf ein externes Device (z. B. Android-Smartphone) oder ein eingebettetes System geladen werden muss.

Nach dieser Erklärung ist der Hauptvorteil der testgetriebenen Entwicklung ersichtlich: Man kann eindeutig und sofort (den) Fehler identifizieren. Diese/r können/kann nur während der letzten Iteration entstanden sein. Doch gibt es neben diesem großen Vorteil noch eine Reihe weiterer Benefits, die an dieser Stelle aufgelistet werden sollen.

5.2.5 Vorteile der testgetriebenen Entwicklung

- Ständig verkaufsfähiges und getestetes Produkt
- Wart- und Lesbarkeit wird erhöht
- Stets guter Überblick über den Projektfortschritt
- Entwickler machen sich schon vorab Gedanken zum Funktionscode
- Möglichkeit, das System zu überlisten, wird ausgeschlossen

Durch das ständige Testen wird der Vorgang vom gesamten Projektteam verinnerlicht, zusätzlich hat man zu jedem Zeitpunkt ein verkaufsfähiges und vor allem getestetes Produkt. In Projekten kommt es oft vor, dass die Projektdauer vor der eigentlichen Testphase vorüber ist. Die Folge daraus ist, dass ein nicht vollständiges und vermutlich nicht einwandfrei funktionierendes Produkt abgeliefert wird. Da man bei der testgetriebenen Entwicklung stets ein funktionstüchtiges Ergebnis zur Verfügung hat, kann man auch leicht auf Kundenwünsche oder Änderungen der Spezifikationen eingehen und das Projekt sogar vorzeitig mit einem verwertbaren Output abbrechen.

Ein weiterer wichtiger Aspekt bei der testgetriebenen Entwicklung ist, dass sich der Entwickler, der den Testfall implementiert, auch in weiterer Folge mit dem Funktionscode beschäftigen muss. Insofern muss er sich schon vorab mit etwaigen Problemen auseinandersetzten und die Randbedingungen überlegen. Wenn man zuerst den Funktionscode schreibt, weiß man schon bestens über die Funktionsbereiche beziehungsweise die Schwächen Bescheid und kann somit leicht einen Testfall schreiben, der bestanden wird,

der aber nicht die Randbedingungen und Problemfälle überprüft. Dies entspricht allerdings keiner guten Testphase und ist ein Hauptproblem der White-Box-Tests, das mit der „Test First“-Vorgangsweise verhindert werden kann. Das Produkt wird dadurch flexibler, billiger und funktionstüchtiger sowie wart- und lesbarer.

5.2.6 Die Gesetze der testgetriebenen Entwicklung

Wie in [Ma07] zu lesen ist, stützt sich die testgetriebene Entwicklung auf drei Grundgesetze, die stets befolgt werden sollten. Wie bereits erwähnt, soll kein Funktionscode geschrieben werden, bevor ein fehlschlagender Testfall implementiert wurde. Der geschriebene Testfall soll nicht mehr als nötig enthalten, und schließlich soll kein Code, der nicht essenziell für das Bestehen des Testfalls ist, erstellt werden.

5.2.7 Persönliche Eindrücke von der testgetriebenen Entwicklung

Die Vorteile der testgetriebenen Entwicklung stehen außer Frage; der Erfolg gibt dem Konzept recht. Es ist zwar eine Einarbeitungsphase notwendig, da man sich bei dieser Entwicklungsmethode bereits im Vorfeld zahlreiche Gedanken über den zu implementierenden Code machen muss. Nach wenigen Testfällen hat man das Konzept jedoch bereits verinnerlicht und kann dank der vielen Qualitätsschleifen ständig ein auch an den Randbedingungen getestetes Produkt abliefern.

6 Praxisteil

6.1 Einleitung

Im folgenden Abschnitt wird der praktische Teil der Diplomarbeit eingeleitet. Anfangen möchte ich mit allgemeinen Basics zum Activity Testing und Beispielen, die die Verwendung von Tests illustrieren sollen. Weiters werden hilfreiche Tools, das Android-Test-Framework, Basiswissen für das Testen auf Mobiltelefonen sowie Testbeispiele aufgearbeitet.

Das Ziel von Tests ist es, auch für spätere Entwicklungsschritte einen funktionierenden Programmcode garantieren zu können. Man schreibt also Tests zur Abdeckung der Funktionalität eines gewissen Codefragments. Sollte das gesamte Programm weiterentwickelt werden, kann man sämtliche Tests durchlaufen lassen und somit immer garantierten, dass dieser spezielle Codeabschnitt noch immer funktioniert. Daher ist es wichtig, viele und vor allem durchdachte Tests zu schreiben. Dazu gehört auch das Ausführen der Tests beim Refactoring, denn der beste Test ist nutzlos, wenn er nicht ausgeführt wird. Unter Refactoring versteht man die Überarbeitung des Codes hinsichtlich Einfachheit, Komplexität, Les- und Wartbarkeit. Damit man beim sogenannten Aufräumen des Codes nicht funktionierenden Code eliminiert, sollten die Tests während der Überarbeitung mehrmals ausgeführt werden. Nur so kann gewährleistet werden, dass in jeder Entwicklungsphase ein funktionierendes Endprodukt zur Verfügung steht. Nach allgemeinen Informationen zum Test-First-Konzept im Theorieteil erläutern die folgenden Seiten nun, wie testgetriebene Entwicklung in der Praxis aussieht.

6.2 Das Test-Framework

Folgende Grafik erklärt den Aufbau des Test-Frameworks:

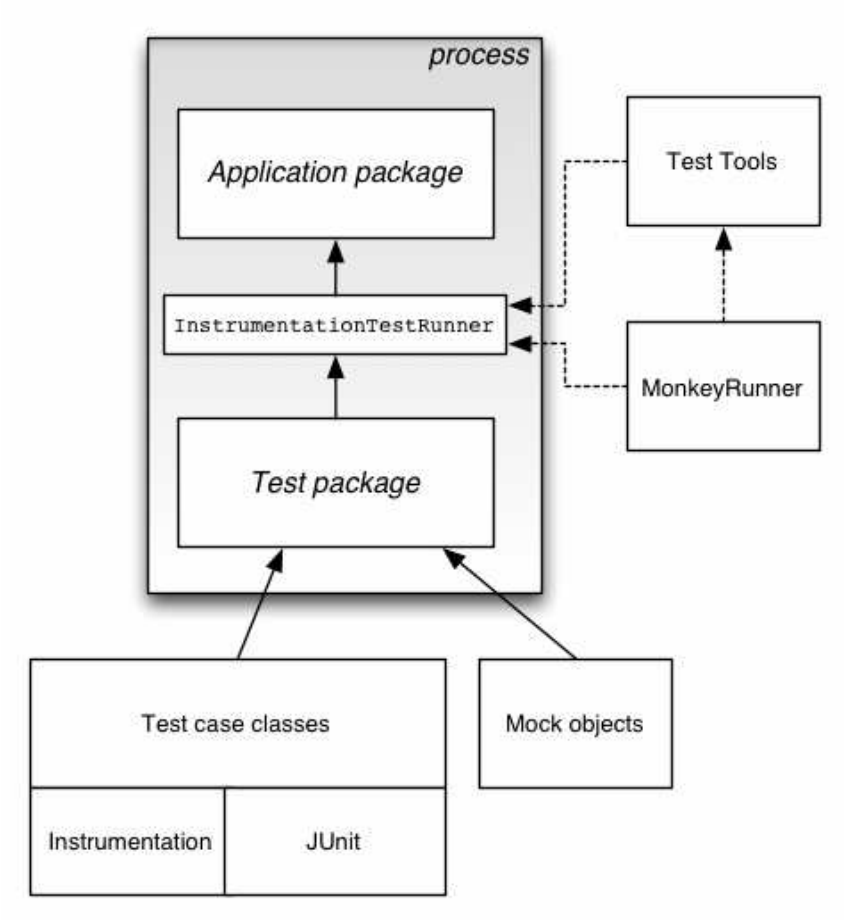

Abbildung 3: Das Test-Framework – Datenmodell-Übersicht [An10]

Hervorzuheben ist hier die Tatsache, dass das Application Package und das Test Package im gleichen Prozess ausgeführt werden.

Wie in [Li12] zu lesen ist, nutzt Android das Java-Unit-Test-Framework JUnit. Dieses Test-Framework hat sich in den vergangenen Jahren erfolgreich in Entwicklerkreisen etabliert und erleichtert das Durchführen von Unit Tests erheblich, wodurch auch die Codequalität sukzessive zunimmt. Neben JUnit erfreuen sich Mocking Frameworks großer Beliebtheit – dabei werden Testkomponenten durch Mocks ersetzt.

Das Android SDK hat zahlreiche Testoberklassen, wie zum Beispiel ActivityUnitTestCase, doch so leicht wie in Java fällt das Unit Testing trotzdem nicht. Der Grund dafür ist laut [Li12] das Deployen der Tests in einer Testapplikation auf einem virtuellen Device (Emulator) wo sie von dem Android JUnit Runner ausgeführt werden können. Das ist bis hierhin eigentlich kein Nachteil, doch die Testbasisklassen erzeugen mehr oder weniger nur eine isolierte Ablaufumgebung, wodurch die Grundidee der Unit Tests verloren geht.

Aufgrund der vielen Abhängigkeiten vom Android-Framework ist eine komplette Isolation einer Klasse oder beispielsweise einer Methode gar nicht so leicht möglich. Etablierte Mocking Frameworks wie EasyMock oder Mockito können hier nicht verwendet werden, weil sie mit Java Reflection und nicht auf der in Android verwendeten Dalvik Virtual Machine funktionieren. Daher muss man auf Android Mock zurückgreifen, das eine Syntaxnähe zu EasyMock besitzt. Der Vorteil ist, dass damit zur Compile-Zeit die Mock-Objekte erzeugt und gemeinsam mit dem Testprojekt auf dem virtuellen Device deployt werden.

Damit bleibt nur noch ein Nachteil des Emulators zu lösen: der Zeitfaktor, der besonders bei der testgetriebenen Entwicklung enorm ist. Durch die vielen Tests verliert man, wie in [Li12] angeführt, wertvolle Entwicklungszeit, weshalb sich die Frage stellt, ob diese nicht umgangen werden können. Die Idee, auf Emulatortests zu verzichten und stattdessen die Tests direkt in JUnit als Java-Projekt auszuführen, liegt zwar nahe, muss allerdings verworfen werden. Die Initialisierung der zu testenden Klassen stellt ein größeres Problem dar: Das mit dem SDK ausgelieferte android.jar enthält keine kompletten Class-Dateien. Erst durch das Zusammenspiel von Android und dem Emulator werden die Klassen ausführbar. Android-spezifischen Code zu kapseln ist auch nicht sinnvoll, da er den Großteil einer Android-Applikation ausmacht. Würde man nur den Android-freien Code testen, wäre die Testabdeckung viel zu gering.

Da weltweit viele Entwickler mit diesem Problem kämpfen, wurde laut [Li12] Robolectric ins Leben gerufen. Dabei handelt es sich um ein reines Unit-Test-Framework, das auf den Emulator verzichtet. In einem gewissen Rahmen können damit sogar UI-Tests durchgeführt werden, dennoch eignet sich Robotium noch besser dafür. Informationen zu den beiden Test-Frameworks folgen in den Kapiteln 6.5.1 Robotium und 6.5.2 Robolectric.

6.3 Activity-Testing-Basics

Laut den Ansätzen des Test-First-Konzeptes soll zuerst der Test und dann der auszuführende Code geschrieben werden. Damit man einen Test erstellen kann, muss man zu Beginn nur das Nötigste für den zu testenden Code implementieren. Das bedeutet, dass man mehr oder weniger nur das Activity-Gerippe erstellt und sich dann dem dafür vorgesehenen Test widmet.

Um die entsprechenden Tests zu erstellen, muss man die zu testende Activity mit der Testklasse verknüpfen. Sobald man diese Verknüpfung (vgl. Abbildung 4: Activity-Testing-Basics – Verknüpfung Testklasse) erstellt hat, wird der Rest von Eclipse automatisch ausgefüllt. Wenn man nun bestätigt, erhält man einen grob vorgefertigten Projektordner, der wie folgt aussieht:

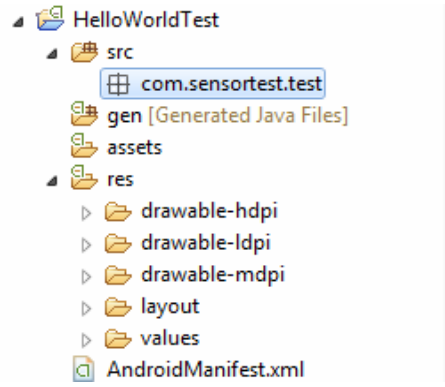

Abbildung 4: Activity-Testing-Basics – Verknüpfung Testklasse

Wenn man nun auf com.sensortest.test einen Rechtsklick macht und dann auf New → Class klickt, erscheint das folgende Fenster:

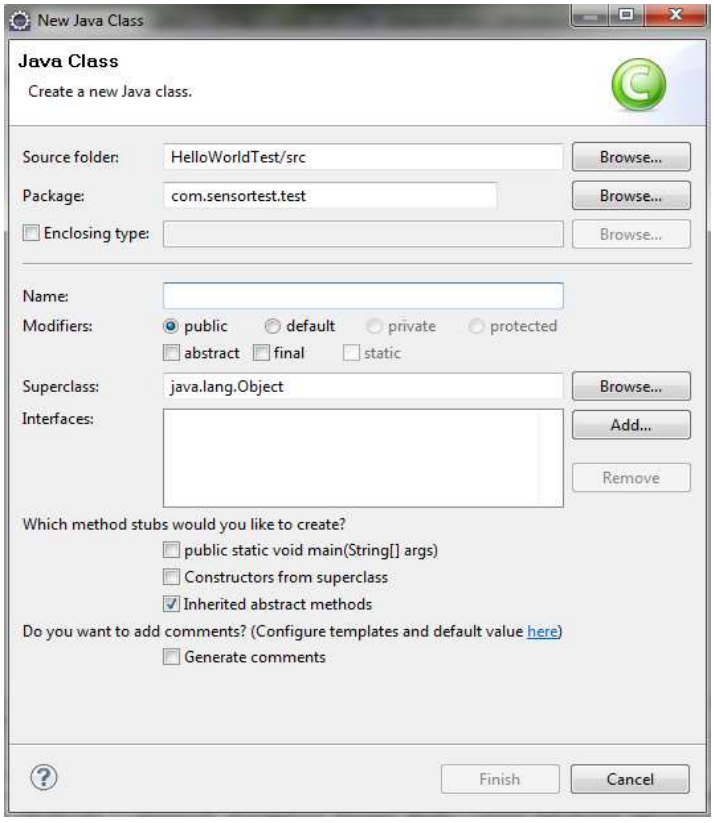

Abbildung 5: Activity-Testing-Basics – Neue Testklasse

Hier sieht man, dass der Source-Folder sowie der Package-Name automatisch von Eclipse eingetragen wurden. Hier muss man lediglich den Namen (HelloWorldTest) sowie die Superclass (android.test.ActivityInstrumentationTestCase2<HelloAndroid>) angeben.

Danach muss man „Finish“ drücken, um folgendes Testgerippe zu erhalten:

```
package com.sensortest.test;

import android.test.ActivityInstrumentationTestCase2;

public class HelloWorldTest extends
        ActivityInstrumentationTestCase2<HelloAndroid> {

}
```

Abbildung 6: Activity-Testing-Basics – Standard-Testgerippe

Kommen wir nun zu einfachen Tests, die in Android mithilfe des Android-JUnit-Framework leicht implementiert und getestet werden können.

Das vorher erstelle Testgerippe (vgl. Abbildung 6: Activity-Testing-Basics – Standard-Testgerippe) wird nun um einige Funktionen erweitert, um beispielsweise schnelle und einfache User-Interface-Tests zu implementieren.

```
@Override
protected void setUp() throws Exception {
    super.setUp();
    mActivity = this.getActivity();
    mView1 = (TextView) mActivity.findViewById(com.sensortest.R.id.textview1);
    mView2 = (TextView) mActivity.findViewById(com.sensortest.R.id.textview2);
    resourceString1 = mActivity.getString(com.sensortest.R.string.hello);
    resourceString2 = mActivity.getString(com.sensortest.R.string.hello2);
}

public void testPreconditions() {
    assertNotNull(mView1);
    assertNotNull(mView2);
  }

public void testText() {
   assertEquals(resourceString1,(String)mView1.getText());
   assertEquals(resourceString2,(String)mView2.getText());
}
```

Abbildung 7: Activity-Testing-Basics – Einfache UI-Tests

Bei diesem Beispiel wird nur ein einfacher Datenabgleich gemacht. Es wird überprüft, ob der Text, der auf dem Display angezeigt wird, mit dem Text, der dort dargestellt werden sollte, übereinstimmt (vgl. testText in Abbildung 7: Activity-Testing-Basics – Einfache UI-Tests).

Beim zweiten Test (vgl. testPreconditions in Abbildung 7: Activity-Testing-Basics – Einfache UI-Tests) wird kontrolliert, ob die zu überprüfenden Views (Bildschirmansichten) überhaupt vorhanden sind.

Wenn beide Tests bestanden werden, kann man im Anschluss mit der Entwicklung fortfahren. Hier ein Bild des JUnit-Reiters, der unter anderem die Anzahl der Tests, die benötigte Dauer sowie die Anzahl der fehlgeschlagenen Tests aufzeigt:

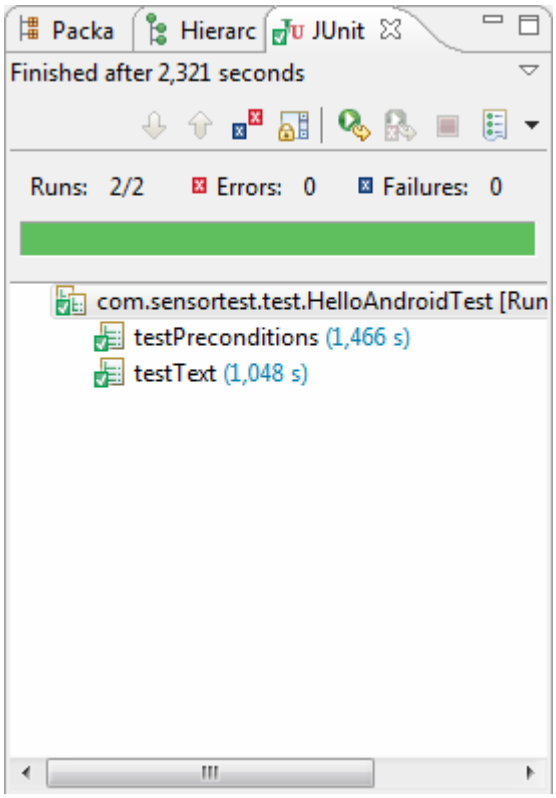

Abbildung 8: Activity-Testing-Basics – JUnit-Reiter

Sollte ein Test nicht erfolgreich sein, wird der grüne Balken in Abbildung 8: Activity-Testing-Basics – JUnit-Reiter rot und auch der fehlgeschlagene Test mit einem blauen X statt eines grünen Hakens dargestellt. Tests, die Errors hervorrufen (z. B. NullPointerExceptions), werden mit einem roten X gekennzeichnet.

Hier ein Bild, das einen fehlgeschlagenen Test zeigt:

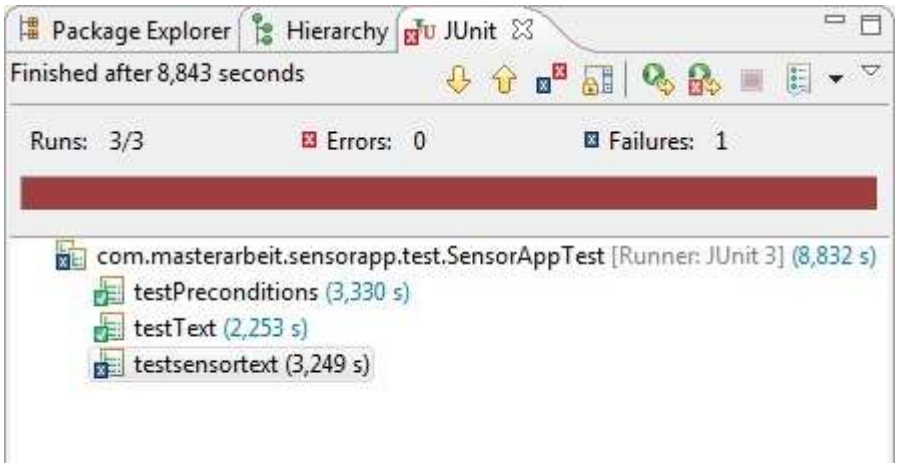

Abbildung 9: Activity-Testing-Basics – JUnit-Reiter mit fehlgeschlagenem Test

Wenn man die Einstellungen in Eclipse nicht verändert, wird unter dem JUnit-Reiter das Fehlerlog-Fenster angezeigt. Anhand dieser Informationen kann man oft schnell und einfach das Problem auffinden und lösen.

Hier ein Bild des Fehlerlogs:

Abbildung 10: Activity-Testing-Basics – Fehlerlog

Sollten die Informationen aus dem Fehlerlog für die Fehlerbehebung nicht ausreichen, kann man ein interessantes Tool zur Hilfe nehmen: DDMS (Abkürzung für Dalvik Debug Monitor Server). Mithilfe dieses Debug-Monitors kann man sich beispielsweise Log Messages ausgeben lassen oder Error Messages analysieren, um Fehler im Code einzugrenzen.

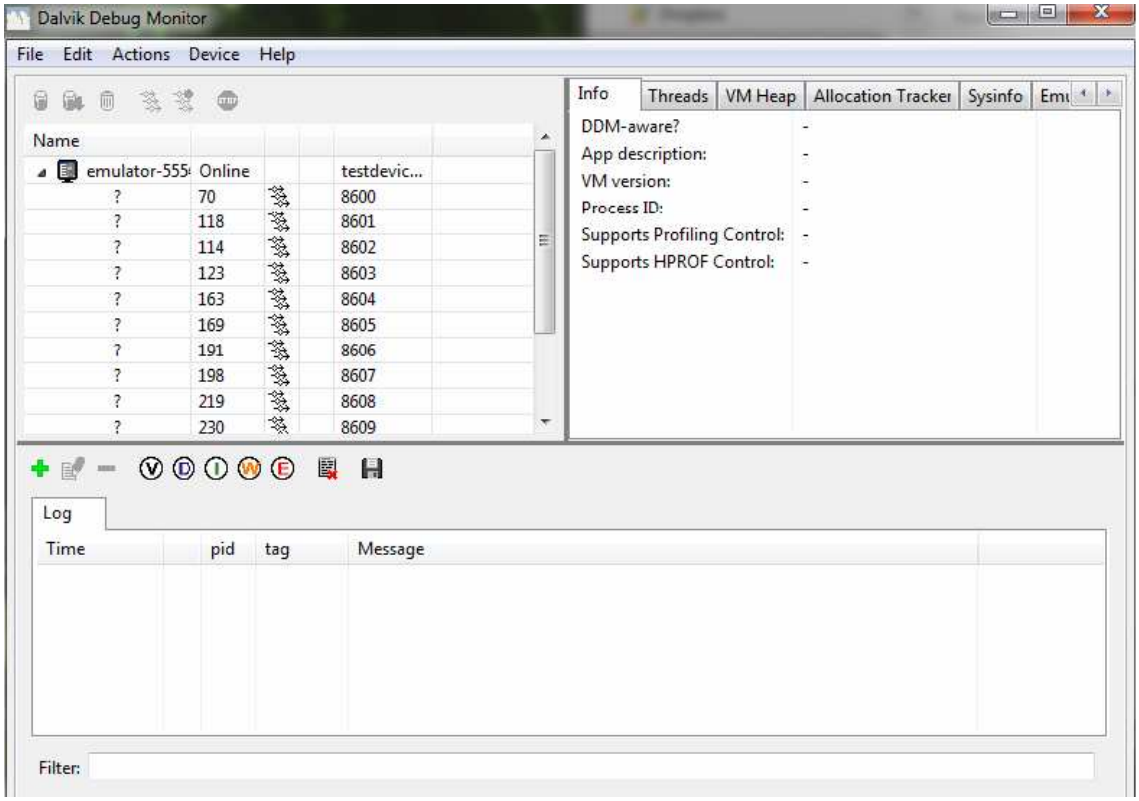

Abbildung 11: Activity-Testing-Basics – DDMS im Standby-Modus

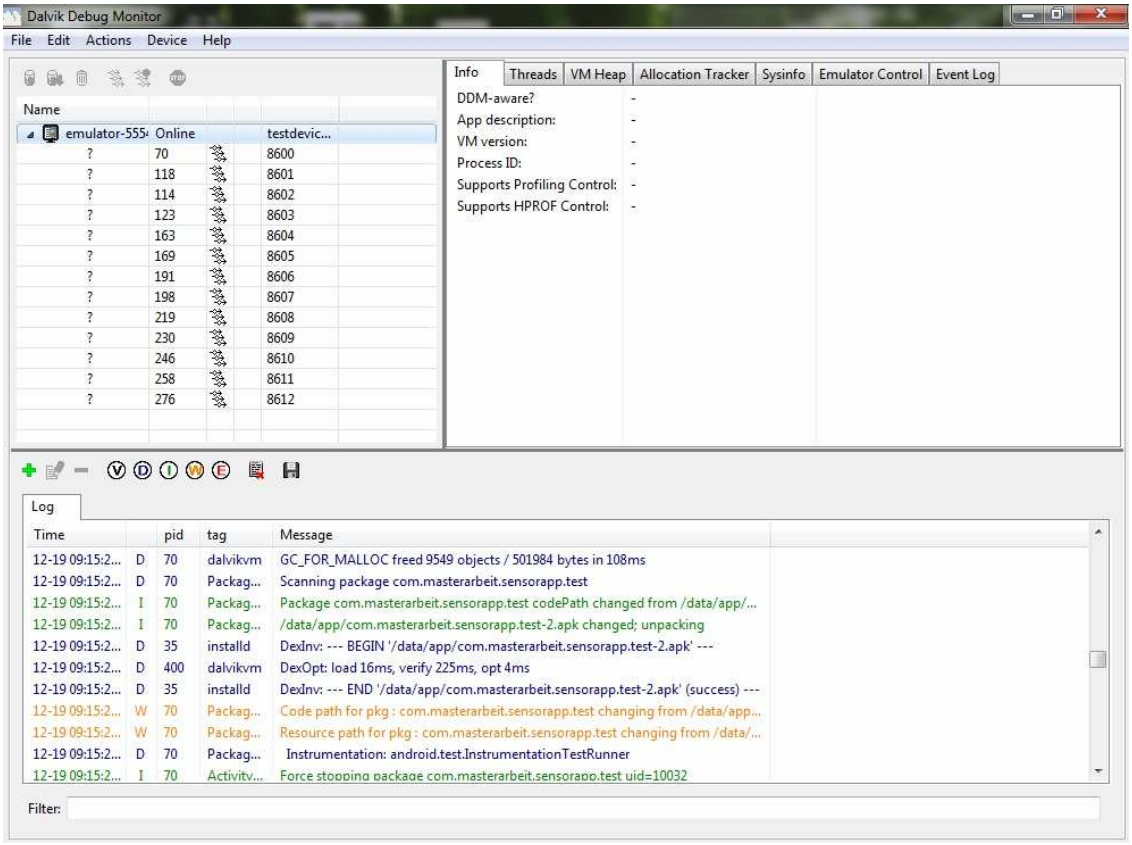

Abbildung 12: Activity-Testing-Basics – DDMS im Working-Modus

Im linken oberen Bereich des Fensters (vgl. Abbildung 12: Activity-Testing-Basics – DDMS im Working-Modus) werden die Devices aufgelistet, die mithilfe des Dalvik-Debug-Monitor-Servers genauer betrachtet werden können. Hier wird beispielsweise der Emulator mit dem erstellten Virtual Device aufgelistet. Sollte man ein Smartphone mit dem PC verbinden, wird auch dieses Mobile Device in diesem Fenster aufgelistet (vgl. Abbildung 13: Activity-Testing-Basics – Device-Auflistung im DDMS).

Hier sieht man die Veränderung, wenn ein Mobile Device angeschlossen ist:

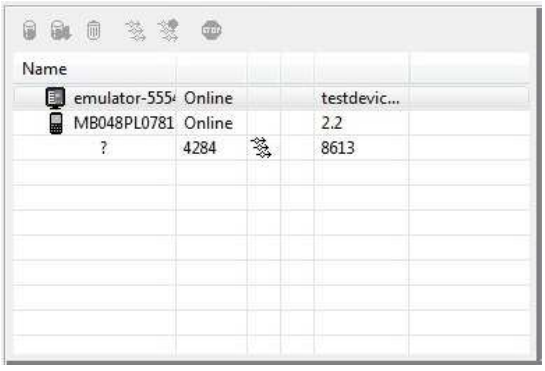

Abbildung 13: Activity-Testing-Basics – Device-Auflistung im DDMS

Die Wichtigkeit des DDMS wird im Folgenden anhand eines einfachen Beispiels erläutert: Bei der Entwicklung wurde festgestellt, dass offenbar ein falscher X-Wert verwendet wird. Um diesen genauer zu untersuchen, erstellt man eine eigene Log Message, die den Wert von X exemplarisch im Log-Fenster ausgibt (vgl. Abbildung 14: Activity-Testing-Basics – Eigene Log-Message im DDMS):

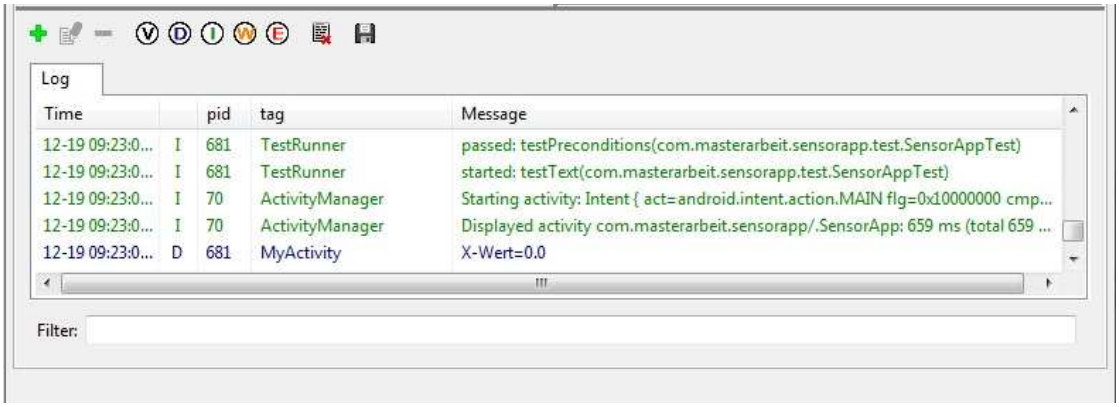

Abbildung 14: Activity-Testing-Basics – Eigene Log-Message im DDMS

In der letzten Zeile des Log-Fensters sieht man die selbst erstellte Log Message, die offenbart, dass der aktuelle Wert der X-Koordinate gleich 0 ist. Dieser Wert entspricht allerdings nicht dem erwarteten, weshalb man das Problem weiter eingrenzen kann und somit einen Anhaltspunkt für die Fehlersuche hat.

Hier noch ein weiteres Beispiel, wie der DDMS die Fehlersuche erleichtern kann:

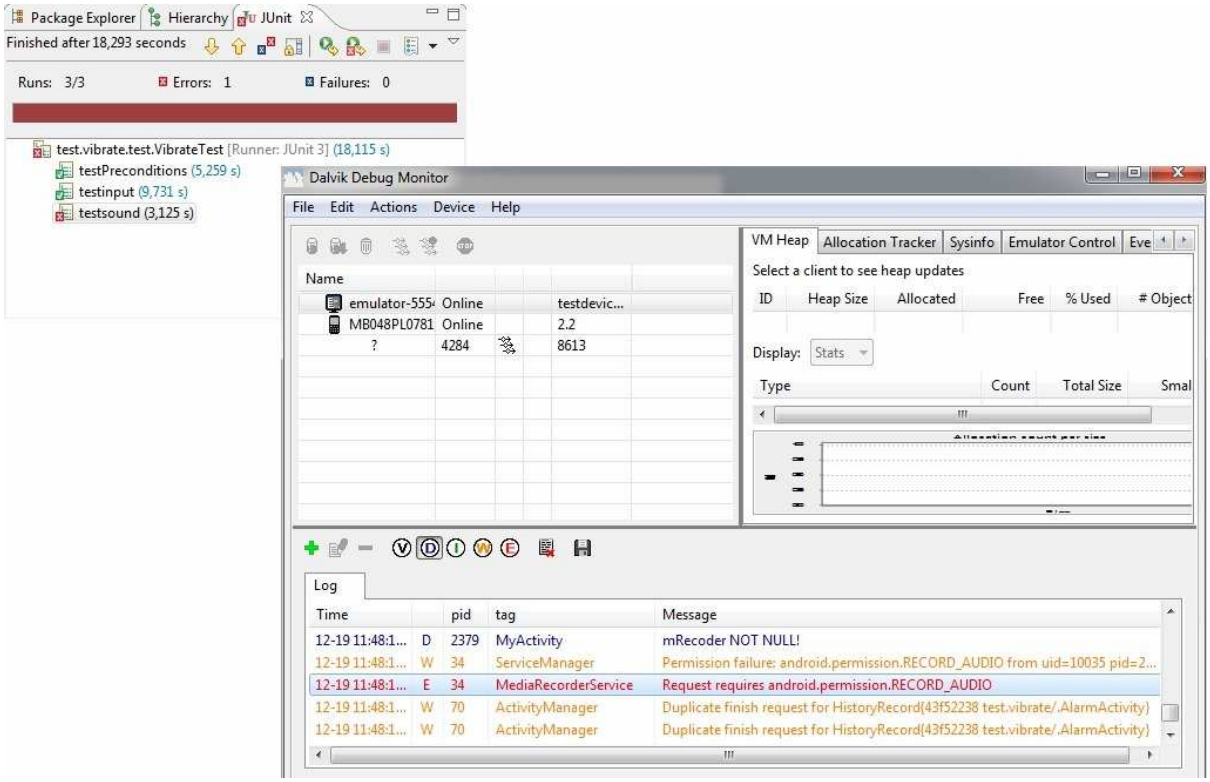

Abbildung 15: Activity-Testing-Basics – Fehlersuche mit DDMS

Hier wurde ein Testfall erstellt, der via Mikrofon versuchen sollte, Sprache oder Musik aufzuzeichnen. Der Test ist fehlgeschlagen, mittels DDMS wurde der Fehler aber schnell identifiziert. Die entsprechende Ausnahmeregel (Record.Audio) wurde im Android-Manifest nicht verankert. Daher konnten keine Samples aufgezeichnet und dementsprechend der Testfall, der dies überprüfen sollte, nicht bestanden werden. Durch das Hinzufügen der Ausnahmegenehmigung im Android-Manifest funktioniert der Test.

Weitere Informationen zur Erstellung von Testfällen und anwendbaren Programmen folgen nach dem Abschnitt Refactoring.

6.4 Refactoring

Der folgende Abschnitt soll den wichtigen Refactoring-Schritt anhand eines praktischen Beispiels illustrieren. Wenn der geschriebene Code funktioniert, ist der nächste Schritt das Code-Aufräumen bzw. Refactoring. Dazu wird der geschriebene Code in Bezug auf Wart- und Lesbarkeit verbessert, bis er schließlich ohne Kommentare auskommt und trotzdem alle Entwickler den Code interpretieren können. Kommentare können zwar verwendet werden, doch sollten sie wirklich nur in äußersten Notfällen Anwendung finden.

Anhand des folgenden Beispiels wird der Refactoring-Vorgang erklärt:

```
package masterarbeit.tugraz.at;

import android.app.Activity;
import android.os.Bundle;
import android.provider.Settings;
import android.text.TextUtils;
import android.widget.TextView;

public class EmulatorDecetion extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {

        // Autor Stefan Hohenwarter
        // if/else statement to proceed different ways for emulator or mobile device
        // for example Vibrate don't work on the emulator

        if(isEmulator()){
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
           }
        else{
            super.onCreate(savedInstanceState);
        setContentView(R.layout.main2);}
    }

    //Autor Stefan Hohenwarter
    //
    // isEmulator ()
    // returns true if the software is running on the emulator
    // returns false if the software is not running on the emulator

    public boolean isEmulator() {
        String id = Settings.Secure.getString(this.getContentResolver(), Settings.Secure.ANDROID_ID);
        boolean emulator = TextUtils.isEmpty(id);
        if (!emulator) {
        emulator = id.toUpperCase().equals("9774D56D682E549C");
        }
        return emulator;
        }
}
```

Abbildung 16: Refactoring – Ausgangscode

Es handelt sich dabei um einen Programmcode, der herausfinden soll, ob es sich bei dem verwendeten Device um ein Handy oder den Emulator handelt. Dies ist besonders für diese Masterarbeit von Bedeutung, da gewisse Funktionen vom Smartphone unterstützt werden,

jedoch im Emulator nicht funktionieren. Beispiele dafür sind die Sensorwerte oder das Vibrieren.

Als Erstes wird unnötiger Code rigoros eliminiert. Ein Beispiel dafür ist das unnötige TextView-Import. Da TextView nicht verwendet wird, kann man den Import-Code getrost entfernen. Wichtig ist jetzt, auch wenn man es als unnötig erachtet, die Testfälle zu durchlaufen. Die Iterationsschritte beim Refactoring können auch sehr klein sein. Es ist nicht sinnvoll, etwas zu überarbeiten und zu vereinfachen, wenn dabei die Funktionalität zerstört wird. Man sollte beides vereinen und sich mit genügend Testfällen vor solchen Problemen absichern. Testen ist niemals vergeudete Zeit.

Als Nächstes werden die Kommentare überarbeitet. Da diese nicht wirklich schwer zu verstehende Codeabschnitte beschreiben, kann man sie auch eliminieren. Somit ist der erste Schritt des Refactorings abgeschlossen, und der Code sieht wie folgt aus:

```
package masterarbeit.tugraz.at;

import android.app.Activity;
import android.os.Bundle;
import android.provider.Settings;
import android.text.TextUtils;


public class EmulatorDecetion extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        if(isEmulator()){
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
           }
        else{
            super.onCreate(savedInstanceState);
        setContentView(R.layout.main2);}
    }
   public boolean isEmulator() {
       String id = Settings.Secure.getString(this.getContentResolver(), Settings.Secure.ANDROID_ID);
       boolean emulator = TextUtils.isEmpty(id);
       if (!emulator) {
       emulator = id.toUpperCase().equals("9774D56D682E549C");
       }
       return emulator;
       }
```

Abbildung 17: Refactoring – Code nach erster Überarbeitung

In weiterer Folge wird der Kommentar „Called when the activity is first created“ gelöscht. Danach fügt man die Variable „emulatorID” ein. Dank dieses selbsterklärenden Namens ist allgemein verständlich, wofür die Zahlenfolge 9774D56D682E549C steht. Ein weiterer wichtiger Punkt des Refactorings ist das Eliminieren von doppelten Codesegmenten. In der

onCreate-Methode sieht man zweimal den identischen Code. Diesen Code zieht man aus dem If-Statement, wobei auch die Namen der xml-files verändert werden. Aus main.xml und main2.xml werden emulator_main.xml und mobile_device_main.xml, wodurch der Code nun wie folgt aussieht:

```
package masterarbeit.tugraz.at;

import android.app.Activity;
import android.os.Bundle;
import android.provider.Settings;
import android.text.TextUtils;

public class EmulatorDecetion extends Activity {

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

        if(isEmulator())
          setContentView(R.layout.emulator_main);
        else
          setContentView(R.layout.mobile_device_main);
    }


    public boolean isEmulator() {
        String id = Settings.Secure.getString(this.getContentResolver(), Settings.Secure.ANDROID_ID);
        boolean emulator = TextUtils.isEmpty(id);
        if (!emulator) {
        String emulatorID = "9774D56D682E549C";
        emulator = id.toUpperCase().equals(emulatorID);
        }
        return emulator;
        }
}
```

Abbildung 18: Refactoring – Code nach zweiter Überarbeitung

Zwischendurch wurde die Funktionalität des Codes stets geprüft. Am Ende dieses Refactoring-Vorgangs sieht man einen deutlich aufgeräumten Code, der die Wart- und Lesbarkeit sowie die weitere Arbeit am Code verbessert beziehungsweise erleichtert.

6.5 Test-Automation-Tools

Ein weiteres wichtiges Werkzeug bei der Erstellung von Testfällen sind sogenannte Test-Frameworks, wie zum Beispiel Robotium. Doch auch abseits von diesen Testumgebungen gibt es Hilfsprogramme, die für den App-Entwickler nützlich sind. Auf den folgenden Seiten wird ein Überblick über diese hilfreichen Tools gegeben.

6.5.1 Robotium

6.5.1.1 Allgemeines

Mit dem Leitspruch „It's like Selenium, but for Android" wirbt Robotium und erklärt damit zugleich, worauf das Framework abzielt. Es ist eine Testumgebung, die das Schreiben von robusten und guten Black-Box-Testfällen für Android-Applikationen erleichtert und das Verhalten von realen Usern simuliert. Mit Robotium kann der Entwickler Funktions-, Akzeptanz- und Systemtestszenarios erstellen. Weiters werden Activities, Dialoge, Toasts, Kontext- und Standardmenüs unterstützt.

Die erste Version von Robotium wurde im Januar 2010 veröffentlicht (die aktuellste Version ist Robotium 4.0, die nun auch Web-Support beinhaltet). Derzeit liegt der Fokus der Entwickler auf dem Support der Robotium-Nutzer, der Fehlerbehebung von auftretenden Problemen sowie der kontinuierlichen Verbesserung des Test-Frameworks. Wenn die Zeit es erlaubt, wird an neuen Features gearbeitet. Beispiele dafür sind eine Remote Control (vergleichbar mit Selenium RC), die automatische Messung der UI-Testabdeckung oder die Erzeugung eines Screenshots bei einem fehlgeschlagenen Test [Ro13].

6.5.1.2 Vorteile von Robotium

Laut den Entwicklern bietet Robotium eine Reihe von Vorteilen, die für die Verwendung des Test-Frameworks sprechen [Ro13]:

- Erstellung aussagekräftiger Testfälle mit minimalem Wissen über die Testapplikation
- Automatische Behandlung mehrerer Android-Applikationen durch das Framework
- Geringer Zeitbedarf für das Schreiben eines soliden Testfalles
- Höhere Lesbarkeit der Testfälle im Vergleich zu Standard-Instrumentation-Tests
- Robustheit der Testfälle durch die Run-Time-Anbindung der GUI-Komponenten
- Schnelle Testfallausführung
- Reibungslose Integration von Maven oder Ant als Teil der Continous Integration

6.5.1.3 Anwendung von Robotium

Um Robotium bei eigenen Android-Applikationen verwenden zu können, müssen die gleichen Schritte wie bei normalen Android-JUnit-Testfällen durchgeführt werden. So erstellt man als ersten Schritt ein Testprojekt im gleichen Package der zu testenden Applikation:

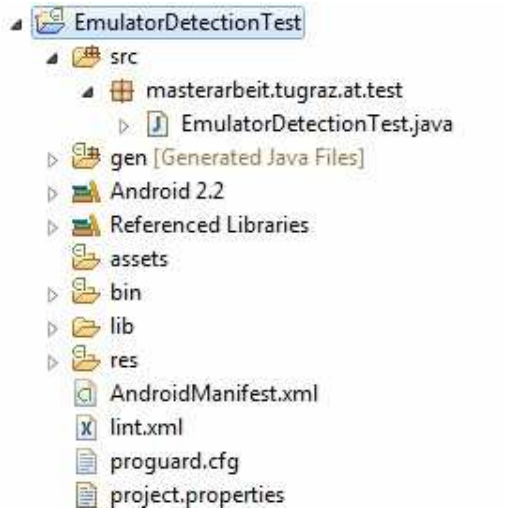

Abbildung 19: Robotium – Testprojekterstellung

Der nächste Schritt ist die Implementierung eines JUnit-Testfalles. Hier kann man auch gleich die Standardmethoden setUp () und tearDown () sowie den Constructor anlegen lassen (vgl. Abbildung 20: Robotium – JUnit-Test-Case-Erstellung).

Abbildung 20: Robotium – JUnit-Test-Case-Erstellung

Bevor man mit dem Schreiben der Testfälle beginnen kann, muss man nur noch eine Referenz des Robotium-jar-Files im Projekt verankern (vgl. Abbildung 21: Robotium – Java-Build-Path-jar-File-Einbindung).

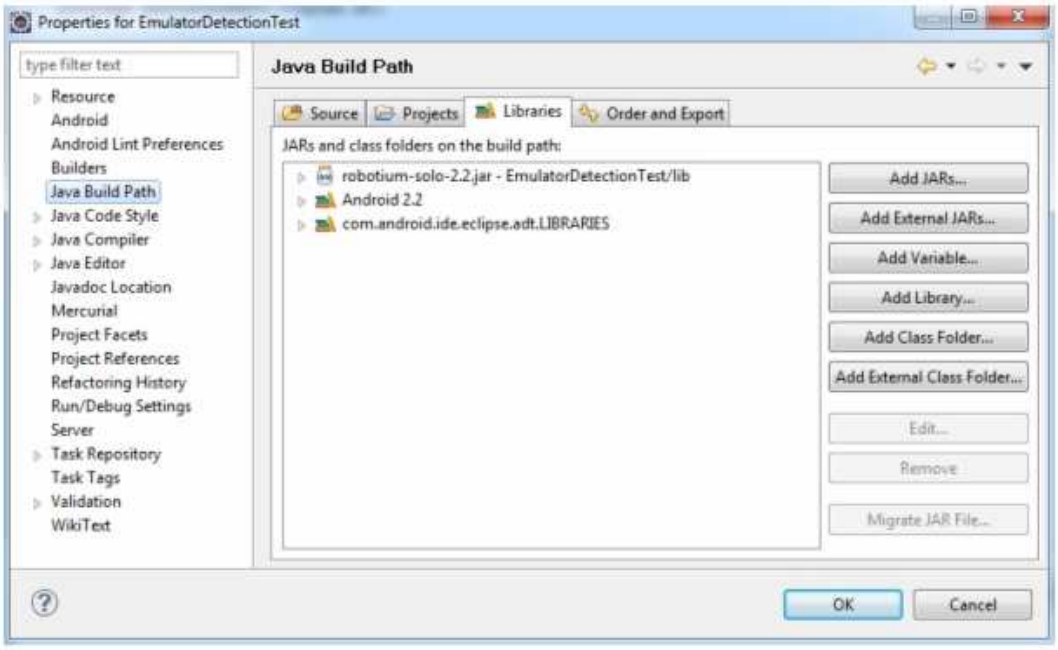

Abbildung 21: Robotium – Java-Build-Path-jar-File-Einbindung

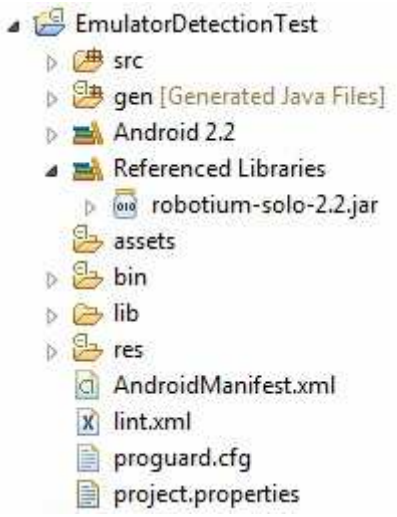

Abbildung 22: Robotium – jar-File im Projektordner

Nachdem nun alles vorbereitet ist, kann man damit beginnen, in der Testklasse die ersten Testfälle zu erstellen.

6.5.1.4 Allgemeine Beispielcodes

Im folgenden Abschnitt werden einige exemplarische Codeschnipsel betrachtet, um einen Eindruck über die Möglichkeiten, die Robotium bietet, zu gewinnen.

```
if(isEmulator())
{
  solo.clickOnButton(0);
  SystemClock.sleep(1000);
  resultsView = (TextView) mActivity.findViewById(masterarbeit.tugraz.at.R.id.resultview);
  assertNotNull(resultsView);
}
```

Abbildung 23: Robotium – Beispielcode clickOnButton

Mit der Anweisung solo.clickOnButton wird überprüft, ob die Applikation die richtigen Ergebnisse liefert, wenn der jeweilige Button angeklickt wird (vgl. Abbildung 23: Robotium – Beispielcode clickOnButton).

```
if(isEmulator())
{
  assertNotSame("", solo.getText("Do you noticed some music?"));
}
```

Abbildung 24: Robotium – Beispielcode assertNotSame

Abbildung 24: Robotium – Beispielcode assertNotSame zeigt ein weiteres Beispiel, in dem überprüft wird, ob ein String auf dem Display angezeigt wird. In weiterer Folge könnte man auch vordefinierte Strings mit den Texten am Display vergleichen (Abbildung 25: Robotium – Beispielcode assertSame).

```
String display = solo.getText(1).getText().toString();
assertSame(resourceString2, display);
```

Abbildung 25: Robotium – Beispielcode assertSame

In Abbildung 26: Robotium – Beispielcode enterText wird die Texteingabe eines Users simuliert. Dank Robotium ist es möglich, solche Eingabefelder schnell und einfach auf ihre Richtigkeit zu überprüfen.

```
double vibrateTime = 2.00;

solo.clickOnEditText(0);
solo.clearEditText(0);
solo.enterText(0, vibrateTime + "");
solo.clickOnButton(0);
```

Abbildung 26: Robotium – Beispielcode enterText

Robotium ist ein mächtiges Testautomationswerkzeug, das auch in der Praxis häufig verwendet wird. Das Framework wird ständig erweitert und ist ein überaus hilfreiches und leicht zu verwendendes Entwicklungstool, das sich aller Voraussicht nach länger auf dem Markt behaupten wird.

6.5.2 Robolectric

Wie auch Robotium ist Robolectric ein Android-Test-Framework. Es gibt allerdings auch einen entscheidenden Unterschied, denn Robolectric ist laut [Pi13] ein reines Unit-Test-Framework, das auf den Emulator verzichtet und dadurch einen enormen Zeitvorteil mit sich bringt. Damit kann die testgetriebene Entwicklung wesentlich schneller und effizienter durchgeführt werden, allerdings auf Kosten der Codesicherheitsabdeckung. Es gibt weiterhin Funktionalitäten, die sich nur mit anderen Test-Frameworks, wie zum Beispiel Robotium, überprüfen lassen. Das bislang letzte große Update zu Robolectric ist die Ankündigung der Alpha-2-Version von Robolectric 2.0. [Pi13]

6.6 SensorSimulator

6.6.1 Allgemeines

Ein weiteres hilfreiches Tool ist der SensorSimulator von Open Intents. Mithilfe dieses Programms kann man laut [Se13] eine Verbindung zwischen einem Java-Programm und dem Emulator herstellen, über die man Sensordaten des Emulators auf einfachste Weise modifizieren kann. Es bleibt dem Anwender selbst überlassen, welche Sensoren er verändern will (z. B. Accelerometer, Orientation oder Magnetic Field). Nach Belieben kann man sich einzelne – oder auch alle – Sensoren aussuchen und diese sehr leicht im gleichnamigen Java-Programm verändern (vgl. Abbildung 28: SensorSimulator – PC-Applikation zur Datenmodifikation). Zusätzlich können auch GPS-Daten oder der Batteriestatus simuliert werden, wodurch man sich auf einige Ernstfälle vorbereiten kann. Abbildung 27: SensorSimulator – Emulator-Applikation zeigt die Applikation im Emulator mit den beiden Reitern Testing und Settings:

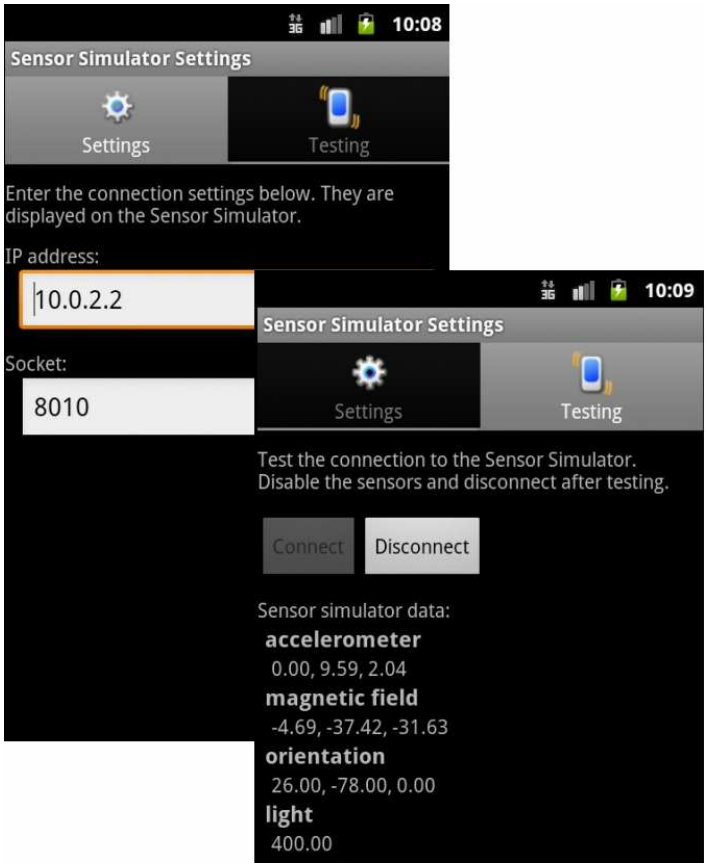

Abbildung 27: SensorSimulator – Emulator-Applikation

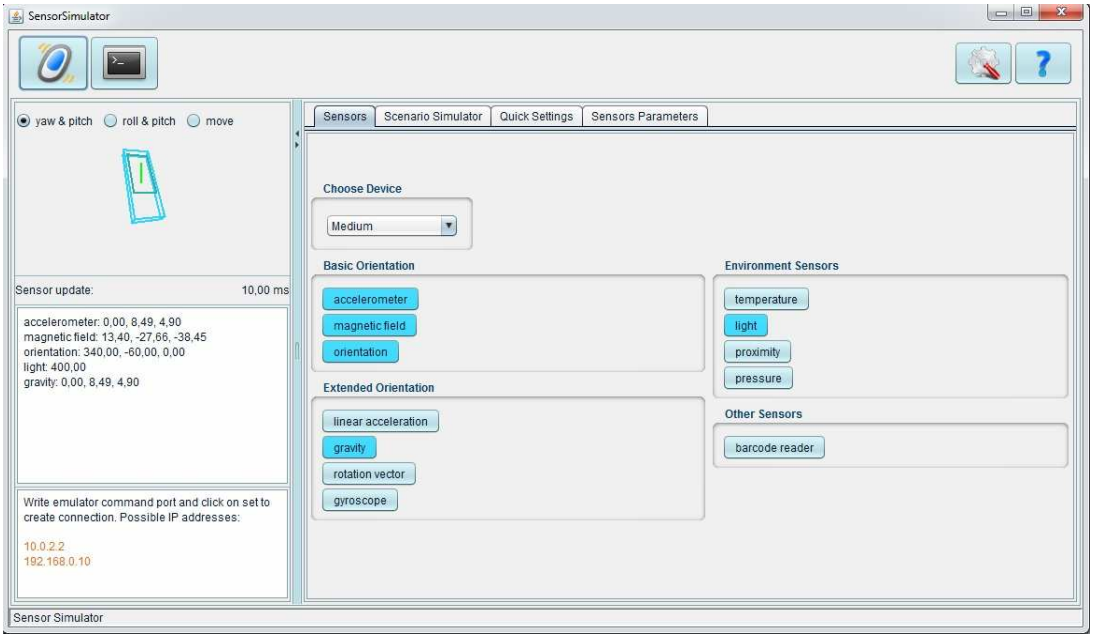

Abbildung 28: SensorSimulator – PC-Applikation zur Datenmodifikation

Laut [Se13] werden die folgenden Sensoren unterstützt:

- Accelerometer
- Orientation
- Magnetic Field
- Light
- Temperature
- Proximity
- Pressure
- Linear Acceleration
- Gravity
- Rotation Vector
- Gyroscope
- GPS

Dank des letzten Updates auf die Version 2.0 und der neu integrierten Features ist der SensorSimulator noch robuster und vielseitiger einsetzbar. Besonders hilfreich ist die Möglichkeit, ein Szenario von einem echten Smartphone aufzunehmen und im Emulator abzuspielen. Diese aufgezeichneten Szenarios können auch gespeichert und geladen werden.

6.6.2 Getting Started

Bevor der SensorSimulator verwendet werden kann, muss das entsprechende apk-File auf dem Emulator installiert werden. Das funktioniert am einfachsten mit der Konsole und dem Befehl **adb install $APK.** $APK ist der Platzhalter für die gewünschte Applikation, die auf dem Emulator installiert werden soll.

Danach muss die Applikation im Emulator gestartet werden. Nachdem der SensorSimulator fertig geladen ist, muss man unter Settings nur noch die IP-Adresse aus dem Java-Programm eintragen, und schon werden die Sensordaten in den Emulator übertragen (vgl. Abbildung 29: SensorSimulator – Verbindungsherstellung).

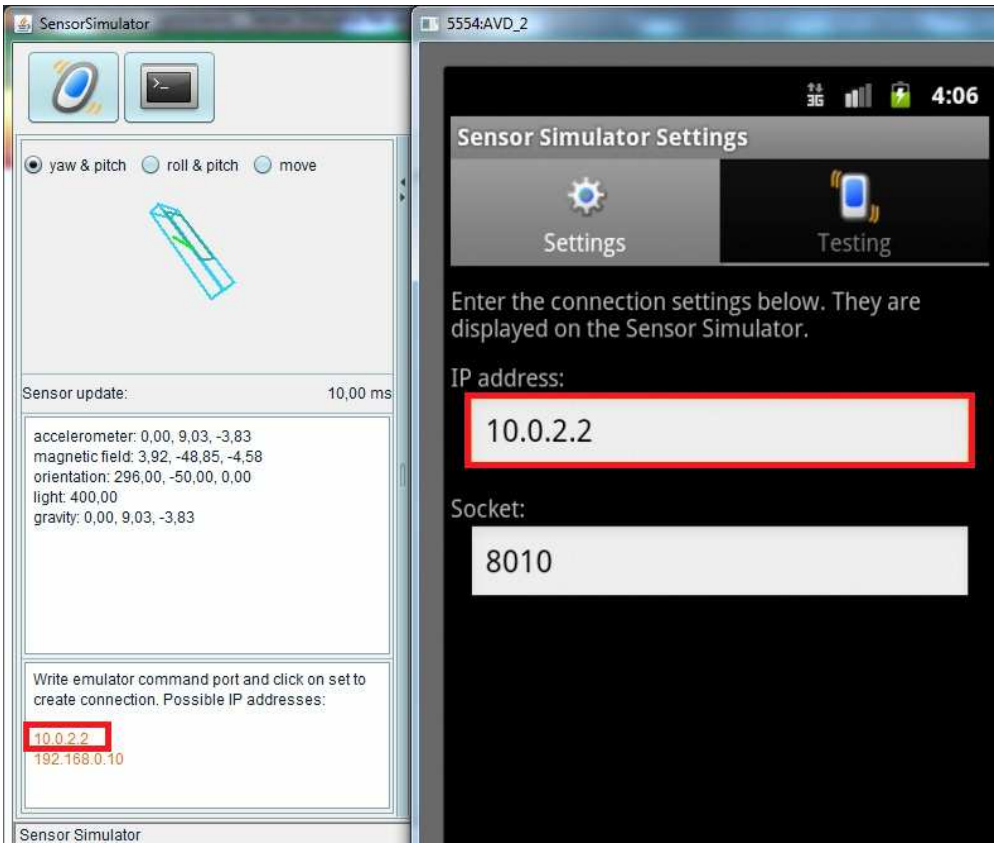

Abbildung 29: SensorSimulator – Verbindungsherstellung

6.6.3 Modifizierung der Sensordaten

Die Sensoren können manuell als Werte im SensorSimulator oder über ein in Echtzeit reagierendes Drag-and-Drop-Fenster eingetragen werden. Letztere Möglichkeit sieht man in Abbildung 30: SensorSimulator – Drag-and-Drop-Wertemodifikation. Man drückt einfach die

Maustaste und zieht sie in die gewünschte Richtung, um beispielsweise eine andere Lage des Smartphones zu simulieren.

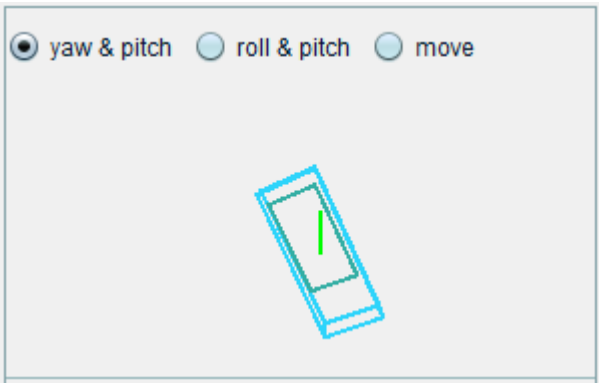

Abbildung 30: SensorSimulator – Drag-and-Drop-Wertemodifikation

Das Quick Setting ist eine alternative Modifikationsmöglichkeit, mit der man die Daten exakt festlegen kann (vgl. Abbildung 31: SensorSimulator – Wertemodifikation via Quick Setting).

Abbildung 31: SensorSimulator – Wertemodifikation via Quick Setting

Abbildung 32: SensorSimulator – Verschiedene Sensordatenbeispiele zeigt einige Beispiele dafür, wie die Sensordaten mittels SensorSimulator verändert werden können.

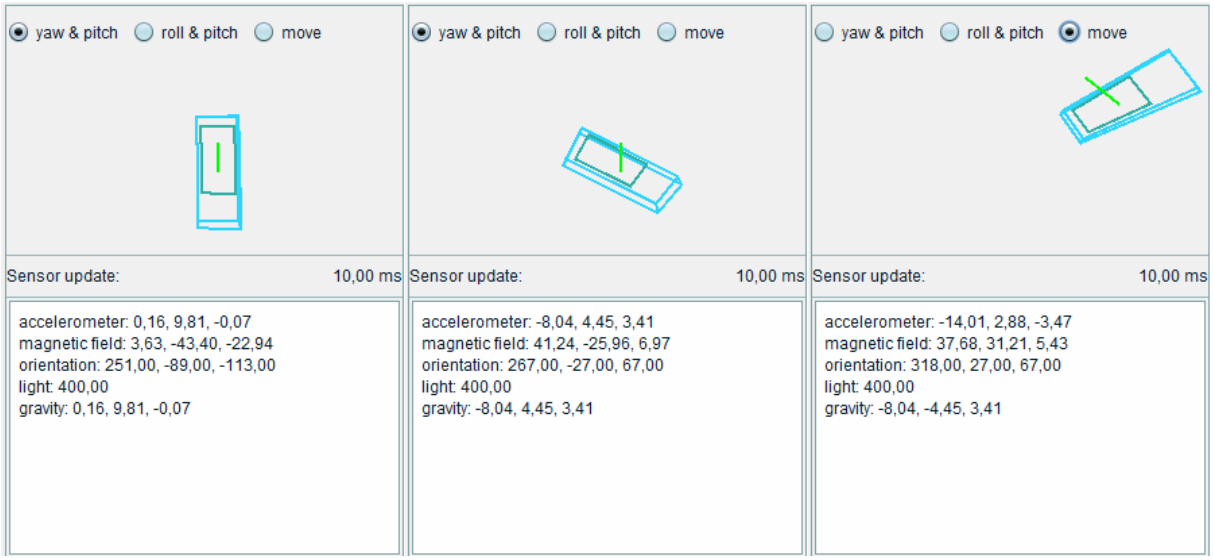

Abbildung 32: SensorSimulator – Verschiedene Sensordatenbeispiele

Mit dem SensorSimulator kann man dem Emulator sehr leicht spezielle Sensorwerte vorspielen. Damit fällt das Testen von Applikationen, die auf Sensordaten angewiesen sind, wesentlich leichter. Man spart sich das zeitintensive Überspielen auf das Smartphone sowie die manuelle Bedienung, um spezielle Sensorwerte zu provozieren. Der Komfort wird folglich erhöht und die benötigte Zeit reduziert. Dazu kommt die einfache Bedienung sowie die Möglichkeit des Aufzeichnens von Smartphone-Bewegungsabläufen, was den Nutzen dieses Tools weiter steigert.

6.7 Einfaches Praxisbeispiel

Nachdem nun sowohl die Theorie als auch die Tools eingehend erklärt wurden, kann das Gelernte in einem realen Praxisfall angewendet werden.

Wie im Theorieteil dargelegt, muss vor dem Erstellen des ersten Codes ein Testfall erstellt werden, der die Funktionalität des Codes überprüft. Nachdem bei der Android-Projekterstellung automatisch ein Hello-World-Funktionscode erzeugt wird, muss man sich eines Tricks bedienen und einfach akzeptieren, dass es hier ausnahmsweise schon einen Funktions- vor einem Testcode gibt.

```
package com.sensortest;

import android.app.Activity;
import android.os.Bundle;

public class HelloWorld extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Abbildung 33: Einfaches Praxisbeispiel – Hello-World-Code

In Abbildung 33: Einfaches Praxisbeispiel – Hello-World-Code sieht man den automatisch erstellten Code der gleichnamigen Applikation. Um diesen zu kontrollieren, müssen Testfälle erstellt werden, die den Funktionscode überprüfen.

Dazu erstellt man, wie im Abschnitt 5.2 Testgetriebene Entwicklung beschrieben, eine Testklasse, stellt die Verbindung zu der zu testenden Klasse her und schreibt dann die notwendigen Variablen und Funktionen.

```
import com.sensortest.HelloWorld;

public class HelloAndroidTest extends ActivityInstrumentationTestCase2<HelloWorld> {

      private HelloWorld mActivity;
      private TextView mView1;
      private String resourceString1;
  public HelloAndroidTest() {
        super("com.sensortest.HelloWorld", HelloWorld.class);
      }
  @Override
  protected void setUp() throws Exception {
      super.setUp();
      mActivity = this.getActivity();
      mView1 = (TextView) mActivity.findViewById(com.sensortest.R.id.textview);
      resourceString1 = mActivity.getString(com.sensortest.R.string.hello);
  }
```

Abbildung 34: Einfaches Praxisbeispiel – Standard-Testfälle

Bevor man mit den Testfällen anfängt, sieht die Testklasse wie in Abbildung 34: Einfaches Praxisbeispiel – Standard-Testfälle aus. Ein interessanter Testfall ist die Überprüfung, ob die einzelnen Views – in diesem Beispiel ist nur eine View vorhanden – existieren. Diese führt

man mit der Funktion testPreconditions (vgl. Abbildung 35: Einfaches Praxisbeispiel – testPreconditions) durch.

```
public void testPreconditions() {
    assertNotNull(mView1);
  }
```

Abbildung 35: Einfaches Praxisbeispiel – testPreconditions

Weiters erstellt man nun die Funktion testText (), die einen Variablenvergleich macht und so die Richtigkeit des Funktionscodes überprüft. Hier wird kontrolliert, ob der dargestellte Text dem erwarteten Text entspricht (vgl. Abbildung 36: Einfaches Praxisbeispiel – testText).

```
public void testText() {
   assertEquals(resourceString1,(String)mView1.getText());
}
```

Abbildung 36: Einfaches Praxisbeispiel – testText

Bevor man nun weitermacht, lässt man die Testfälle einmal durchlaufen und sieht sich das Ergebnis an. Normalerweise müssten die Tests fehlschlagen, weil der Funktionscode noch nicht existiert, da aber in diesem speziellen Fall der Code schon vor den Tests vorhanden war, werden alle Testfälle bestanden, wie Abbildung 37: Einfaches Praxisbeispiel – JUnit-Reiter (Tests bestanden) beweist.

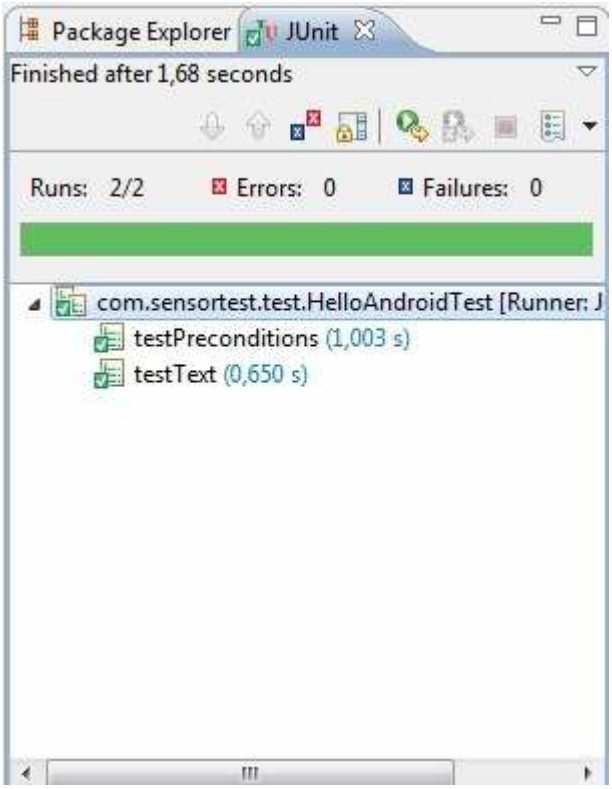

Abbildung 37: Einfaches Praxisbeispiel – JUnit-Reiter (Tests bestanden)

Man erweitert nun die Funktionalität der Hello-Android-Applikation, indem man mit der Implementierung eines komplexeren Testfalls startet.

6.8 Erweitertes Praxisbeispiel

Mit einem erweiterten Praxisbeispiel soll nun ein Schritt weiter gegangen und das Gelernte vertieft werden:

```
    private HelloWorld mActivity;
    private TextView mView1;
    private String resourceString1;
    private TextView mView2;
    private String resourceString2;
    private Solo solo;

public HelloAndroidTest() {
      super("com.sensortest.HelloWorld", HelloWorld.class);
     }
@Override
protected void setUp() throws Exception {
     super.setUp();
     mActivity = this.getActivity();
     solo = new Solo(getInstrumentation(), mActivity);
     mView1 = (TextView) mActivity.findViewById(com.sensortest.R.id.textview);
     mView2 = (TextView) mActivity.findViewById(com.sensortest.R.id.textview2);
     resourceString1 = mActivity.getString(com.sensortest.R.string.hello);
     resourceString2 = mActivity.getString(com.sensortest.R.string.hello2);
}
public void testPreconditions() {
     assertNotNull(mView1);
   }
public void testText() {
    assertEquals(resourceString1,(String)mView1.getText());

    //Press Button and check Text
    solo.clickOnButton(0);
    SystemClock.sleep(1000);
    String display = solo.getText(1).getText().toString();

    assertSame(resourceString2, display);
}
}
```

Abbildung 38: Erweitertes Praxisbeispiel – Testklasse

Aufgabe ist es nun, eine Möglichkeit zu verankern, um per Button auf eine neue Bildschirmansicht (View) zu wechseln. Als Basis für den neuen Testfall dient der Einzeiler aus dem Beispiel zuvor (vgl. Abbildung 36: Einfaches Praxisbeispiel – testText). Dank des Vergleiches in der Funktion testText () (vgl. Abbildung 38: Erweitertes Praxisbeispiel – Testklasse) kann man schnell die Korrektheit des angezeigten Textes beim Start der Applikation überprüfen. Durch die Integration des Robotium-Framework ist es nun möglich, den entsprechenden Code für das Drücken des Buttons schnell und leicht zu implementieren. Zusätzlich baut man noch eine kurze Wartezeit ein, damit die Applikation genügend Zeit zum Umschalten auf die neue View hat. Mit Robotium kann man auch einfach den Text vom Display auslesen und das Ergebnis mit dem erwarteten String vergleichen. Noch bevor man den notwendigen Code zum Bestehen des Tests schreibt, startet man einen Testdurchlauf.

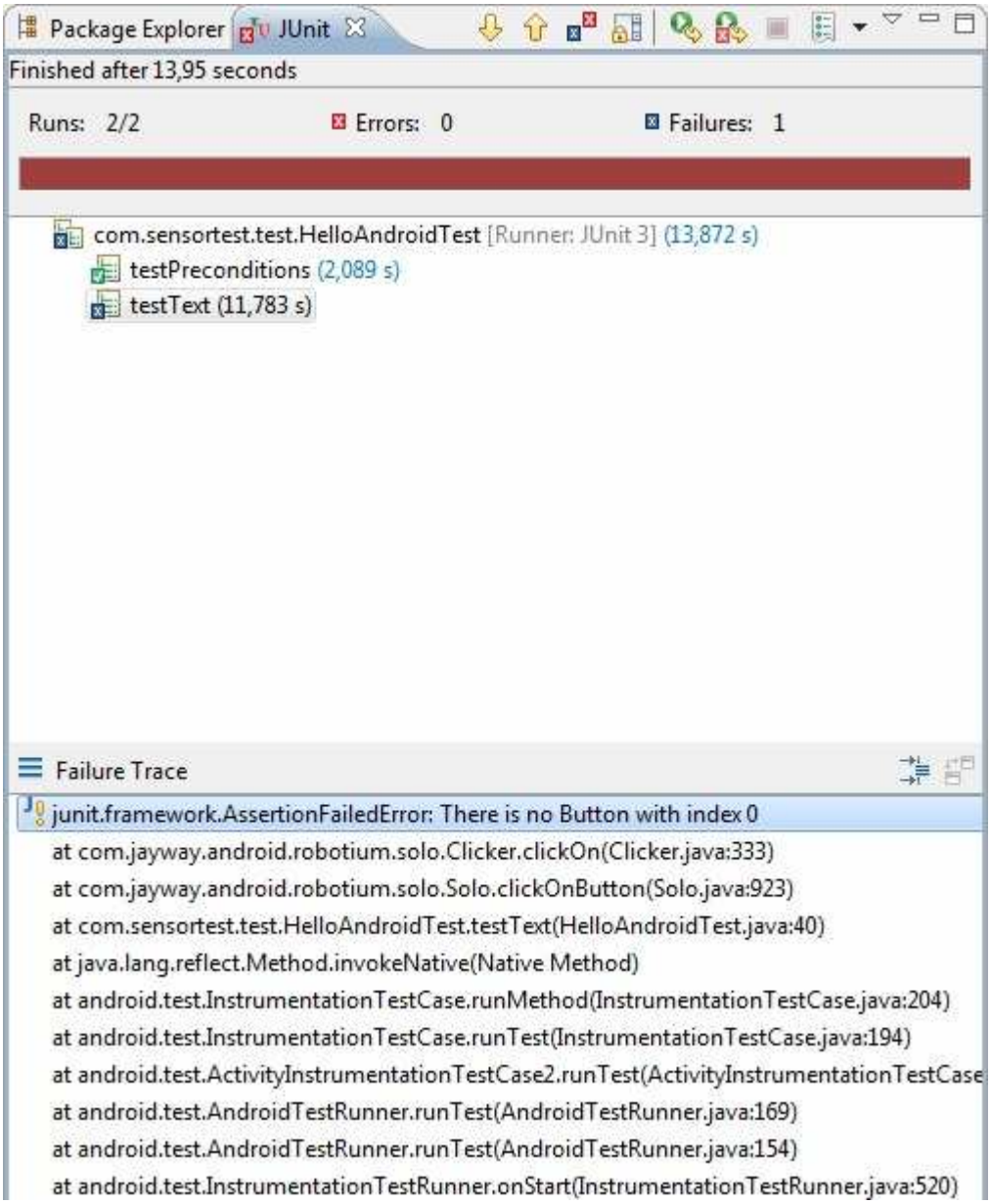

Abbildung 39: Erweitertes Praxisbeispiel – Button fehlt, Test nicht bestanden

Wie nicht anders zu erwarten, schlägt der Test fehl. Im Error Log (vgl. Abbildung 39: Erweitertes Praxisbeispiel – Button fehlt, Test nicht bestanden) sieht man auch gleich den Grund dafür: Es wurde kein Button mit dem Index 0 gefunden. Nun widmet man sich der Erstellung des Funktionscodes, um zukünftig das Scheitern dieses Testfalls zu verhindern. Man beginnt mit der Adaption der Views: Den Anfang macht die Main.xml, die um einen Button erweitert wird.

```
<?xml version="1.0" encoding="utf-8"?>
 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:layout_weight="1">

     <TextView
         android:id="@+id/textview"
         android:layout_width="fill_parent"
         android:layout_height="133dp"
         android:text="@string/hello" />

<Button
         android:id="@+id/button2"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="Button" />

</LinearLayout>
```

Abbildung 40: : Erweitertes Praxisbeispiel – Start-View mit Button

Nachdem man nun die Start-View (main.xml) verändert und um einen Button erweitert hat, widmet man sich der zweiten View, auf die mittels Knopfdruck umgeschaltet werden soll. Die in weiterer Folge erscheinende View kann in Abbildung 41: Erweitertes Praxisbeispiel – Ziel-View betrachtet werden.

```
<?xml version="1.0" encoding="utf-8"?>
 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:layout_weight="1">

  <TextView
  android:id="@+id/textview2"
  android:layout_width="fill_parent"
  android:layout_height="133dp"
  android:text="@string/hello2"/>
</LinearLayout>
```

Abbildung 41: Erweitertes Praxisbeispiel – Ziel-View

Es ist zwar vorauszusehen, dass der Test noch immer fehlschlagen wird, da der Button zwar im System ist, aber noch kein Funktionscode zum View-Wechsel erstellt wurde, dennoch startet man einen Testlauf. Das Ergebnis kann in Abbildung 42: Erweitertes Praxisbeispiel – View-Vergleich fehlgeschlagen betrachtet werden.

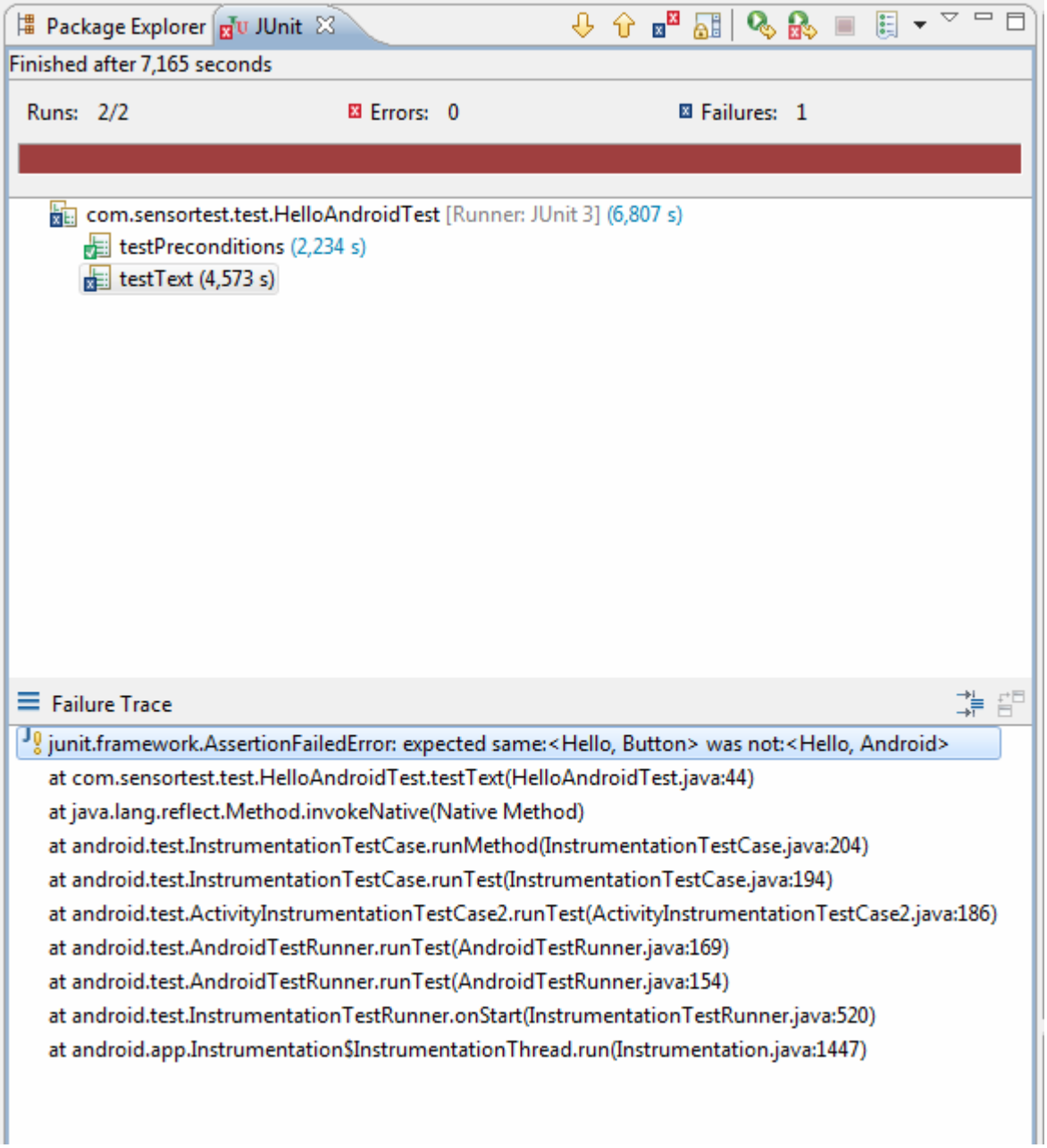

Abbildung 42: Erweitertes Praxisbeispiel – View-Vergleich fehlgeschlagen

Der Test wird wie erwartet nicht bestanden, wenngleich der Test nun an einer anderen Stelle fehlschlägt. Nachdem vorher noch das Fehlen eines Buttons als Grund für das Nichtbestehen des Tests angegeben wurde, steht im aktuellen Error Log (vgl. Abbildung 42: Erweitertes Praxisbeispiel – View-Vergleich fehlgeschlagen), dass der Fehler nun beim Ergebnisvergleich auftritt. Es wird der String „Hello Button“ erwartet, der in der zweiten View implementiert wurde. Da jedoch der View-Wechsel noch nicht eingebaut wurde, bleibt der Text der Start-View (main.xml) auf dem Bildschirm, wodurch es zum Fehlschlagen des Testfalles kommt.

Um diese Information reicher, widmet man sich nun der Implementierung des View-Wechsels.

```
package com.sensortest;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;


public class HelloWorld extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //Button zum Wechsel der Views
        final Button buttonMain2 = (Button) findViewById(R.id.button2);

        buttonMain2.setOnClickListener(new View.OnClickListener()
        {
            public void onClick(View v)
            {
                setContentView(R.layout.main2);
            }
        });

    }

}
```

Abbildung 43: Erweitertes Praxisbeispiel – View-Wechsel implementiert

Der komplette Funktionscode wurde nun implementiert (vgl. Abbildung 43: Erweitertes Praxisbeispiel – View-Wechsel implementiert). Ein weiteres Mal lässt man die Testfälle durchlaufen, um zu eruieren, ob der neue Testfall nun bestanden wird.

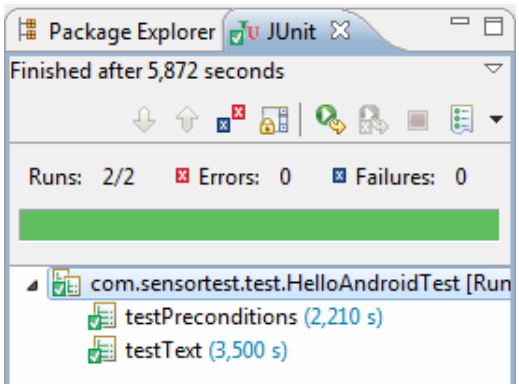

Abbildung 44: Erweitertes Praxisbeispiel – View-Wechsel-Testfall bestanden

Da der Testfall nun bestanden wird (vgl. Abbildung 44: Erweitertes Praxisbeispiel – View-Wechsel-Testfall bestanden) und der neu geschriebene Funktionscode abgesichert ist, ist der nächste Schritt das Refactoring. Dadurch wird die Les- und Wartbarkeit erhöht und gleichzeitig überflüssiger Code – ganz den Vorgaben des Extreme-Programming-Konzepts entsprechend – eliminiert (vgl. Abbildung 45: Erweitertes Praxisbeispiel – Refactoring Funktionscode).

Den Anfang macht man in der Applikation, in der man die Kommentare und die nicht gebrauchten Imports löscht. Weiters nimmt man sich der Button-ID an, da buttonMain2 nicht selbsterklärend ist (vgl. Abbildung 46: Erweitertes Praxisbeispiel – Refactoring String Ressources).

```
package com.sensortest;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class HelloWorld extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.startview);

        final Button switchView = (Button) findViewById(R.id.nextview);

        switchView.setOnClickListener(new View.OnClickListener()
        {
            public void onClick(View v)
            {
                setContentView(R.layout.newview);
            }
        });
    }
}
```

Abbildung 45: Erweitertes Praxisbeispiel – Refactoring Funktionscode

Zusätzlich müssen auch die Ressourcen (Layout & Strings) überarbeitet werden.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="startview">Hello, Android</string>
    <string name="app_name">SensorTest</string>
    <string name="newview">Hello, Button</string>
    <string name="button">Switch</string>
</resources>
```

Abbildung 46: Erweitertes Praxisbeispiel – Refactoring String Ressources

```xml
<?xml version="1.0" encoding="utf-8"?>
 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:layout_weight="1">

     <TextView
         android:id="@+id/startview"
         android:layout_width="fill_parent"
         android:layout_height="133dp"
         android:text="@string/startview" />

<Button
         android:id="@+id/nextview"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@string/button" />

</LinearLayout>
```

Abbildung 47: Erweitertes Praxisbeispiel – Refactoring Start-View-Layout

```xml
<?xml version="1.0" encoding="utf-8"?>
 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:layout_weight="1">

  <TextView
  android:layout_width="fill_parent"
  android:layout_height="133dp"
  android:text="@string/newview"/>
</LinearLayout>
```

Abbildung 48: Erweitertes Praxisbeispiel – Refactoring Ziel-View-Layout

Zwischen und nach dem Refactoring lässt man mehrmals die Testfälle durchlaufen, damit entstandene Fehler schnell eingegrenzt und beseitigt werden können. Wenn der Code bereinigt wurde, kann man sich der Erstellung weiterer Testfälle inklusive Funktionscode widmen.

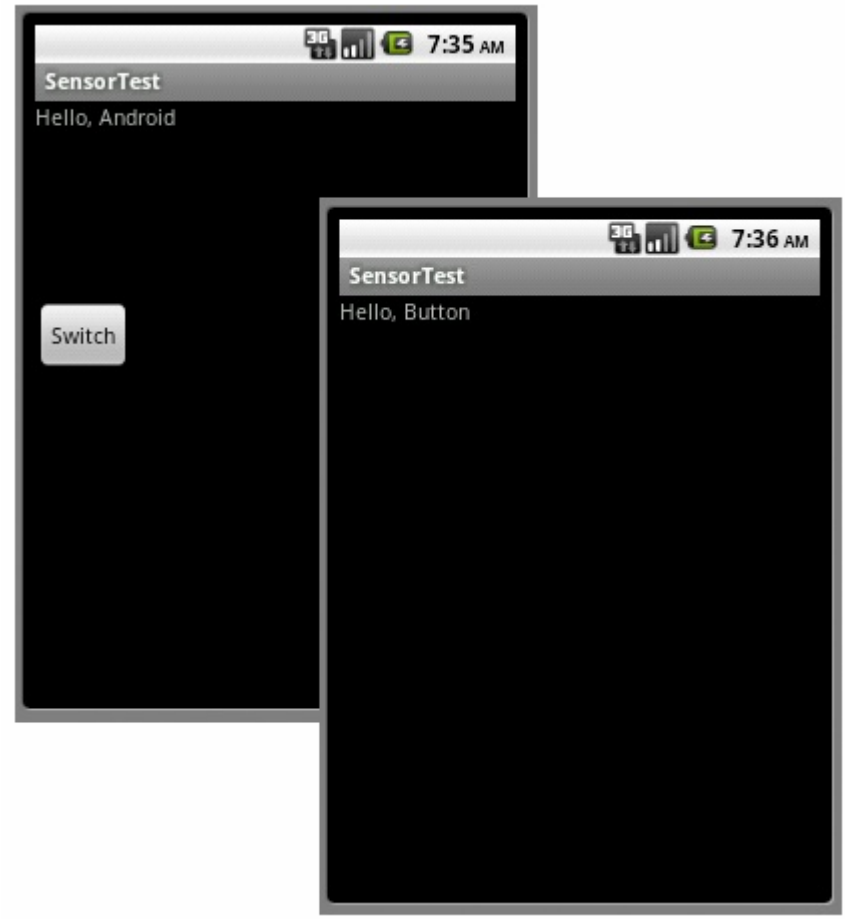

Abbildung 49: Erweitertes Praxisbeispiel – Start- und Ziel-View auf dem virtuellen Device

Auf Abbildung 49: Erweitertes Praxisbeispiel – Start- und Ziel-View auf dem virtuellen Device sieht man die Applikation mit ihren beiden Views, zwischen denen mithilfe des Buttons Switch umgeschaltet werden kann. Nachdem man die Applikation überarbeitet hat, geht man zu den Testfällen über. Auch hier gibt es Refactoring-Potenzial.

```
package com.sensortest.test;

import android.os.SystemClock;
import android.test.ActivityInstrumentationTestCase2;
import android.widget.TextView;
import com.sensortest.HelloWorld;
import com.jayway.android.robotium.solo.Solo;

public class HelloAndroidTest extends ActivityInstrumentationTestCase2<HelloWorld> {

       private HelloWorld mActivity;
       private TextView mView1;
       private String resourceString1, resourceString2;
       private Solo solo;

    public HelloAndroidTest() {
          super("com.sensortest.HelloWorld", HelloWorld.class);
        }
    @Override
    protected void setUp() throws Exception {
        super.setUp();
        mActivity = this.getActivity();
        solo = new Solo(getInstrumentation(), mActivity);
        mView1 = (TextView) mActivity.findViewById(com.sensortest.R.id.startview);
        resourceString1 = mActivity.getString(com.sensortest.R.string.startview);
        resourceString2 = mActivity.getString(com.sensortest.R.string.newview);
    }
    public void testPreconditions() {
        assertNotNull(mView1);
      }
    public void testText() {
       assertEquals(resourceString1,(String)mView1.getText());
       solo.clickOnButton(0);
       SystemClock.sleep(1000);
       String display = solo.getText(1).getText().toString();
       assertSame(resourceString2, display);
    }
}
```

Abbildung 50: Erweitertes Praxisbeispiel – Refactoring Testklasse

Abbildung 50: Erweitertes Praxisbeispiel – Refactoring Testklasse zeigt die überarbeiteten Testfälle (entfernte Kommentare, Code-Redundanzen usw.), die man abermals durchlaufen lässt, um sicher zu sein, dass beim Refactoring keine Fehler eingebaut wurden. Dieser Vorgang wird fortgesetzt, bis alle Funktionalitäten in der Applikation verankert sind.

Nun wurde die Erstellung einer Applikation mithilfe der testgetriebenen Entwicklung dargelegt. Dabei wurde auch auf das Test-Framework Robotium zurückgegriffen und damit beispielsweise der Button zum Wechseln zwischen Bildschirmansichten benutzt, um die Korrektheit der angezeigten Texte zu überprüfen.

Im Folgenden soll ein Schritt weiter gegangen und Testfälle entwickelt werden, die auf die Sensordaten des Testgerätes zugreifen. Damit erhält man eine neue Möglichkeit für die Implementierung von Testfällen, wodurch sie noch robuster und vielschichtiger werden.

7 Einsatz von Smartphone-Sensoren

Der letzte Abschnitt der Arbeit beschäftigt sich mit dem Einsatz von Sensoren für positive und negative Zwecke (z. B. Testen auf Smartphones oder Attacken auf die Privatsphäre der Anwender) sowie mit einem Ausblick darauf, wie man Smartphone-Sensoren einsetzen könnte, um die Lebensqualität zu verbessern.

7.1 Testen mithilfe von Sensordaten

Smartphones der aktuellen Generation verfügen über eine Vielzahl von Sensoren, wie beispielsweise einen Temperaturfühler und Bewegungs- oder Beschleunigungssensoren. Diese Daten könnte man für Testfälle heranziehen, um diese noch effektiver zu gestalten. Doch dafür ist es in einem ersten Schritt nötig, sich Gedanken über die Sensoren zu machen und Experimente durchzuführen, um verwendbare Parameter und Sensordaten heranziehen zu können. Möglichkeiten zur Datengewinnung bieten der Beschleunigungssensor und das Mikrofon, die man beispielsweise zur Erkennung von Vibrationen oder Sound verwenden kann.

7.1.1 Vibrationen detektieren

Im Zuge meiner Mitarbeit am Catroid-Projekt entwickelte ich einen Funktionsblock, der die Vibrationsfunktion leicht nutzbar macht. Da alle im Catroid-Projekt entwickelten Codesegmente eingehend getestet werden müssen, untersuchte ich, ob und vor allem wie man Vibrieren über Sensordaten erkennen kann. Zwar müssen einige Rahmenbedingungen geschaffen werden, aber dafür kann man Vibrieren ohne User-Interaktion oder externe Hardware effektiv detektieren.

Untersucht wurden unter anderem die Werte des Accelerometer sowie die Orientation-Sensorwerte. Man könnte vermuten, dass die Daten des Beschleunigungssensors naheliegender wären, doch hier konnten keine zuverlässigen Daten für die Erkennung des Vibrierens abgeleitet werden. Insofern wurden die Orientation-Sensorwerte, die die Lage des Smartphones widerspiegeln, genauer untersucht. Hier fiel auf, dass an der X-Achse des Sensors eine erkennbare und reproduzierbare Abweichung detektiert werden kann, wenn das Smartphone auf eine glatte Oberfläche, wie zum Beispiel einen Tisch, gelegt wird.

Um die Zahlenwerte des Sensors besser deuten zu können, werden sie mit dem Faktor 1.000 multipliziert und dann mit einem Threshold verglichen. Praxistests zeigten, dass der Grenzwert für die Erkennung von Vibrieren bei einem x-Wert von 3.000 liegt. Wenn der x-Wert der Orientation-Sensordaten diesen Threshold überschreitet, vibriert das Smartphone. Durch diese Beobachtung kann man einen Testfall erstellen, der für die Detektion von Vibrieren nicht nur Flags oder Variablen, sondern auch Sensordaten ausliest. Dadurch wird der Test zuverlässiger und vor allem vielschichtiger. Die Verwendung der Orientation-Sensordaten hat einen weiteren Vorteil: Bei der Installation einer Applikation auf dem Smartphone müssen vom User Freigaben erteilt werden, damit die Applikation beispielsweise auf die Sensordaten wie Accelerometer oder Orientation zugreifen kann. Die Orientation-Daten sind für den User vermutlich weniger sensible Daten als beispielsweise die GPS-Daten.

Bevor man einen Testfall zum Detektieren von Vibrieren erstellen kann, muss man eine Unterscheidungsmöglichkeit verankern, um die Testumgebung zu erkennen. Der Android-Emulator bzw. das Virtual Device sind nur Softwarelösungen und können nicht vibrieren, weshalb hierfür ein entsprechender Code implementiert werden muss (vgl. Abbildung 51: Testen mithilfe von Sensoren – Testumgebung erkennen).

```
public boolean isEmulator()
{
  String id = Settings.Secure.getString(mActivity.getContentResolver(), Settings.Secure.ANDROID_ID);
  boolean emulator = TextUtils.isEmpty(id);
  if (!emulator)
  {
    String emulatorID = "9774D56D682E549C";
    emulator = id.toUpperCase().equals(emulatorID);
  }
  return emulator;
}
```

Abbildung 51: Testen mithilfe von Sensoren – Testumgebung erkennen

Mit diesem Funktionscode kann man erkennen, ob die Applikation vom Emulator oder einem Smartphone aufgerufen wird. Sollte Letzteres der Fall sein, kann mithilfe des folgenden Testfalls das Vibrieren des Handys via Sensordaten detektiert werden (vgl. Abbildung 52: Testen mithilfe von Sensoren – Vibrationsdetektion (Testfall)).

```
public void testVibrate()
{
  if(isEmulator())
  {
    assertTrue(true);
  }
  else
  {
    if(mActivity.vibratedetected)
    {
      assertTrue(true);
    }
    else
    {
      assertTrue(false);
    }
  }
}
```

Abbildung 52: Testen mithilfe von Sensoren – Vibrationsdetektion (Testfall)

Wie man im Code des Testfalls sieht, wird die eigentliche Vibrationserkennung nicht im Testfall, sondern direkt in der Applikation verankert. Im Testfall wird nur noch die entsprechende Variable ausgelesen und für die Detektion des Vibrierens herangezogen. Interessant ist nun der Funktionscode in der Applikation, in der das Vibrieren erkannt werden soll. Das Wichtigste dabei ist der SensorEventListener, der unumgänglich ist, wenn man Sensordaten in der Applikation verwenden möchte, denn nur über den SensorEventListener kann man auf die Orientation-Daten zugreifen. Abbildung 53: Testen mithilfe von Sensoren – Vibrationsdetektion (Funktionscode) zeigt den entsprechenden Funktionscode.

```
@Override
public void onSensorChanged(SensorEvent event) {
    synchronized (this)
     {

        if (event.sensor.getType() == Sensor.TYPE_ORIENTATION)
        {
          orient_x = event.values[1] * 1000;

          if(temp_x < orient_x)
          {
            temp_x = orient_x;
          }
        }
        if(temp_x >= vibratetreshold)
        {
          vibratedetected = true;
        }
    }
}
```

Abbildung 53: Testen mithilfe von Sensoren – Vibrationsdetektion (Funktionscode)

Wie eingangs erwähnt, hat sich ein Threshold von 3.000 als zielführend herausgestellt. Über einen vordefinierten Zeitraum wird der höchste x-Wert des Orientation-Sensors gespeichert. Übersteigt dieser den Threshold vibratetreshold, wird die Variable vibratedetected auf „true“ gesetzt. Diese Variable wird im vorher erstellten Testfall testVibrate ausgelesen und überprüft.

Wichtig bei diesem Test ist, dass das Handy ruhig auf einem Tisch liegt, um stets dieselben Voraussetzungen zu schaffen. Dies ist aber nur ein Beispiel dafür, wie Sensordaten zur Unterstützung bei Tests dienen können. Ein weiteres Beispiel ist die Erkennung von Sound über das Mikrofon.

7.1.2 Sound detektieren

Neben den Sensoren können auch Input-Geräte, wie beispielsweise das Mikrofon, für Testfälle genutzt werden. Im Zuge meiner Entwicklungsarbeit am Catroid-Projekt kam mir die Idee, dass man das Mikrofon zur Erkennung von Sound verwenden könnte. Die ursprüngliche Idee bezog sich auf einen Anwendungsfall im Programmiertool Catroid. Ich dachte mir, man könnte mittels lauten Kommandos die Applikation steuern (z. B. wenn der Mikrofon-Input eine vordefinierte Schranke überschreitet, könnte die Applikation eine selbst erstellte Routine starten). Diese Idee fand aber auch bei der Integration von robusteren Testfällen Anwendung. So kann man relativ einfach Sound über das eingebaute Mikrofon aufnehmen und damit leicht erkennen, ob die Sound-Ausgabe eines Smartphones richtig funktioniert.

Wie auch beim Erkennen von Vibration muss man eine Unterscheidung implementieren, da nur mit dem Smartphone Sounds aufgezeichnet werden können. Der Emulator unterstützt diese Funktion derzeit noch nicht, womit wieder der Fallunterscheidungscode von der Vibrationsdetektion verwendet wird.

Im Folgenden sieht man den interessanteren Codeabschnitt: den Testfall für die Erkennung von Sound.

```
public void testSound()
{

    if(isEmulator())
    {
        //expected behavior
        assertFalse(soundtest);
    }
    else
    {
      solo.clickOnButton(1);
      SystemClock.sleep(3000);
      soundtest = mActivity.sounddetected;
      assertTrue(soundtest);
    }
}
```

Abbildung 54: Testen mithilfe von Sensoren – Sound-Detektion (Testfall)

Der Testfall ist auf das Wesentliche reduziert. Hier wird nur der Bool-Wert der Variable soundtest überprüft. Der eigentliche Funktionscode des Testfalls wurde, wie auch bei der Detektion von Vibrieren, direkt in der Applikation verankert. Während in der Funktion detectSound der Mediaplayer gestartet wird, erfolgt die Sound-Detektion in der Funktion testsound (vgl. Abbildung 55: Testen mithilfe von Sensoren – Sound abspielen (Funktionscode)).

```
void detectSound()
{
    mp = MediaPlayer.create(getBaseContext(), R.raw.kalimba);
    mp.start();
    setContentView(R.layout.mobile_sound);
    testsound();
}
```

Abbildung 55: Testen mithilfe von Sensoren – Sound abspielen (Funktionscode)

Abbildung 56: Testen mithilfe von Sensoren – Sound-Detektion (Funktionscode) zeigt den Funktionscode, der für die Erkennung des Sounds zuständig ist. Für einen kurzen Zeitraum werden sämtliche Geräusche via Mikrofon aufgezeichnet (insgesamt 18.000 Samples). Dabei wird immer das lauteste Geräusch gespeichert und am Ende der Aufzeichnungsphase mit einem Threshold verglichen. Wenn der entsprechende Grenzpegel überschritten wird, setzt man die Variable sounddetected auf den Bool-Wert „true“. Diese Variable wird im Testfall überprüft. Bei diesem Beispiel wird allerdings nur kontrolliert, ob überhaupt Geräusche zu hören sind. Es reicht auch, einfach nur in das Mikrofon zu reden. In weitere Folge könnte man

beispielsweise spezielle Geräuschmuster aufnehmen und vergleichen, um die Testfälle noch robuster zu machen.

```
public boolean testsound()
{
    int audiotemp = 0;
    buffersize = AudioRecord.getMinBufferSize(15000, AudioFormat.CHANNEL_CONFIGURATION_MONO, AudioFormat.ENCODING_PCM_16BIT);
    AudioRecord record = new AudioRecord(AudioSource.MIC, 15000,
    AudioFormat.CHANNEL_CONFIGURATION_MONO,
    AudioFormat.ENCODING_PCM_16BIT, 50*buffersize);
    record.startRecording();
    short[] buff = new short[15000];
    record.read(buff, 0, buff.length);
    record.stop();
    mp.stop();
    for(int count=0; count < 15000; count++)
    {
        if(buff[count] != 0)
        {
            audiotemp = (buff [count]*buff [count])/buff [count];
            if(maxAmplitude < audiotemp)
                {
                    maxAmplitude = audiotemp;
                }
        }
    }
    if(maxAmplitude > soundtreshold)
    {
        sounddetected = true;
    }
    else
    {
        sounddetected = false;
    }
    return sounddetected;
}
```

Abbildung 56: Testen mithilfe von Sensoren – Sound-Detektion (Funktionscode)

Die Verwendung bei Tests ist allerdings nur ein Beispiel, wie Sensordaten für positive Zwecke eingesetzt werden können. Auf den folgenden Seiten möchte ich weitere Beispiele dafür anführen, ehe ich zum Einsatz von Sensordaten für Angriffe auf die Privatsphäre komme.

7.2 Andere positive Anwendungsbeispiele von Sensoren

7.2.1 Erkennung von Bewegungen mittels Accelerometer-Daten

7.2.1.1 Allgemeines

Ein interessantes Anwendungsszenario für Sensoren wird in [La10] beschrieben. Hier wurde mit Sensoren experimentiert und ergründet, ob man auch den Accelerometer von Smartphones für eine verlässliche Bewegungsdetektion verwenden kann, denn tragbare Sensoren sind ein gutes Werkzeug, um Kontexte zu erkennen. Tragbare Sensoren liefern auch relativ gute Werte, doch viele der Tests finden unter Laborbedingungen statt. Dazu kommt das unangenehme Gefühl, überwacht zu werden und oft unhandliche Sensoren am Körper tragen zu müssen. Hier schaffen Smartphones Abhilfe, denn diese sind mit leistungsfähigen

Sensoren ausgestattet und wesentlich „vertrauter“ als diverse Messgeräte. Wenn auch die Erkennung gute Ergebnisse liefert, wären Smartphones ein probates Mittel, um die Bewegung von Menschen durch den Alltag zu verfolgen und Rückschlüsse auf User-Verhalten zu ziehen. Ein weiteres Anwendungsszenario wäre laut [La10] das Monitoring von älteren oder allein lebenden Menschen.

7.2.1.2 Gründe für die Verwendung von Smartphones als Sensorplattformen

- Es sind keine externen Sensoren erforderlich.
- Smartphones weisen eine hohe Rechenleistung und genügend Speicher auf, um die gesammelten Daten zu verarbeiten.
- Das Gefühl der Anwender, überwacht zu werden, ist bei der Verwendung von Smartphones wesentlich geringer als bei externen Sensoren.
- Dank einer relativ langen Akkuzeit können mit einem guten Sensordaten-Polling-Management die Daten eines ganzen Tages aufgezeichnet werden.

7.2.1.3 Wichtige Erkenntnisse

- Es sind keine High-Sampling-Raten wie bei reinen Accelerometern möglich – Abhilfe schafft eine gute Wahl von Sample-Raten und Window Size für die Feature-Extraktion.
- Die Position des Smartphones beeinflusst die Erkennung (fixe vs. lose Position).

7.2.1.4 Experiment

Für das Experiment in [La10] wurde ein Nokia N95 8 GB zur Aufzeichnung der Daten und ein Nokia N800 Internet Tablet zur manuellen Kommentierung der Bewegungen (Gehen, Stehen, Sitzen, Stiegen hinauf, Stiegen hinunter) verwendet. Die Daten vom Accelerometer wurden zusammen mit den Kommentaren des Anwenders, der jede Bewegung auf dem Tablet dokumentierte, aufgezeichnet. Der Vorgang wurde zudem in zwei Settings durchgeführt. Beide Male wurde das Smartphone in der Hosentasche der Probanden aufbewahrt, mit dem Unterschied, dass es einmal fixiert wurde und einmal lose war.

Die gesammelten Daten wurden für die Klassifizierung herangezogen, um zu sehen, wie genau die Erkennung ist. Weiters wurden die Daten beider Settings kombiniert, um zu untersuchen, ob die verschiedenen Settings eine Auswirkung auf die Erkennung haben. Neben den Settings war auch die Feature-Selektion entscheidend.

Die Feature-Selektion wurde auf den Durchschnitt und die Standardabweichung der Accelerometer-Rohdaten sowie den Durchschnitt und die Standardabweichung der Fast-Fournier-Transform-Komponenten in der Frequenzdomäne eingeschränkt. Um eine klare Aussage treffen zu können, wurde das Experiment mit verschiedenen Sample-Rate-Windows-Size-Kombinationen wiederholt.

7.2.1.5 Ergebnis und Konklusion

Für das Setting 1 (fixe Position in der Hosentasche) konnte eine fast hundertprozentige Genauigkeit erzielt werden, während bei der Kombination beider Settings „nur“ eine Trefferquote von 96,52 % erreicht wurde. Interessant ist hierbei, dass die Versuche zeigen, dass bereits mit einer Sample-Rate von 10 und 20 Hz Ergebnisse über 90 % möglich sind. Dies wurde durch die Kombination der Durchschnitts- und Standardabweichung der Accelerometer-Daten erreicht. Für die Klassifikation wurden verschiedene Verfahren wie Bayesian Network, K Nearest Neighbour, Decision Tree und der Rule Based Learner Jrip verwendet. Schließlich wurde festgestellt, dass man mit gängigen Smartphones durch eine gute Wahl von Sample-Raten, Window Size und Klassifikationsverfahren einfache Bewegungen (Gehen, Stehen, Sitzen, Stiegen hinauf, Stiegen hinunter) sehr gut detektieren kann. Aufbauend auf diesen Ergebnissen sind laut [La10] Echtzeiterkennungen und weitere Untersuchungen im Alltag möglich.

7.2.2 Bewegungsmustererkennung mittels Smartphone

7.2.2.1 Allgemeines

Wie in [La10] wird auch in [Bu12] den Smartphones mit Sensoren ein enormes Potenzial für die Entwicklung neuer Applikationen und die Analyse von User-Verhalten zugesprochen. Smartphones mit eingebauten Sensoren sind laut [Bu12] zu einer Mainstreamplattform für Forscher geworden, da sie nicht nur verbreitet sind, sondern auch große Datenmengen verarbeiten können.

Die Verfasser von [Bu12] haben sich zum Ziel gesetzt, die Verwendung von Smartphone-Sensoren in alltäglichen Situationen in urbanen Umgebungen genauer zu beleuchten, und dabei ein interessantes Anwendungsgebiet entdeckt. Der wichtigste Aspekt beim Mobile-Computing ist die Context Awareness. Darunter versteht man laut [Bu12] die Fähigkeit, Daten aufzunehmen, und basierend auf diesen Beobachtungen mit der Umwelt zu interagieren. Besonders beliebt sind Bewegungsmuster, die schon seit geraumer Zeit im Fokus von Forschern stehen. Detektiert werden diese Muster mithilfe von Accelerometer, Gyrosensor und Kompass, die in gängigen Smartphones zu finden sind.

7.2.2.2 Ablauf

Beispiele für solche Bewegungsmuster sind Tippen, Sprechen, Gehen, Klettern und sogar die Ausführung von Reha-Übungen. Während viele Experimente auf die Erkennung von einzelnen Aktionen (Sitzen, Stehen, Gehen usw.) abzielen, wird in [Bu12] versucht, komplexe und kombinierte Bewegungsabläufe zu detektieren. Das Ziel dieser Fallstudie war es, herauszufinden, ob solche komplexen Bewegungsabläufe in einem urbanen Setting erkannt und aus den extrahierten Daten nächste Schritte abgeleitet werden können.

Ein Beispiel dafür wäre laut [Bu12] das Stehenbleiben bei einer roten Ampel und das Losgehen, wenn die Ampel auf Grün schaltet. Für die Navigation könnte man Google Maps und den GPS-Sensor von Smartphones verwenden, doch für die Aufzeichnung aller Ampeln in einer Stadt müsste man eine riesige Datenbank befüllen, was schlichtweg zu teuer und langwierig ist. Die Idee, die in dieser Fallstudie vorgestellt wird, ist einfach und treffend: Man nutzt die Daten (Verkehrslärm und GPS-Koordinaten) von Smartphone-Accelerometern. Immerhin haben in urbanen Gebieten unzählige Menschen ein solches Gerät, das via Internet die gesammelten Daten an einen Remote Server übertragen kann. Dieser legt die entsprechenden Daten in einer Datenbank ab, wodurch keine „menschliche“ Arbeit nötig ist.

Doch wie soll man mit Accelerometer-Daten eine Ampel detektieren? Die Verfasser von [Bu12] überlegten sich einen einfachen Ablauf, der für die Erkennung verwendet werden kann. Normalerweise geht man mit einer durchschnittlichen Geschwindigkeit durch die Stadt. Kommt man nun zu einer roten Ampel, bleibt man stehen. Sobald die Ampel auf Grün schaltet, erhöht man die Geschwindigkeit, um die Straße schnell zu überqueren. Auf der anderen Seite angekommen, setzt man die Reise mit der normalen Geschwindigkeit fort. Unabhängig vom verwendeten Algorithmus wird gleich klar, dass diese Detektionsmethode

nicht die sicherste ist. Das verwendete Erkennungsmuster kann auch bei anderen Situationen im Alltag vorkommen: Beispielsweise könnte man bei einem Schaufenster stehen bleiben, in die Auslage schauen und dann die verlorene Zeit wieder aufzuholen versuchen. Man kann auch nicht annehmen, dass man immer zu einer roten Ampel kommt und man derart dazu gezwungen wird, stehen zu bleiben. Weiters kann man auch nicht davon ausgehen, dass der Großteil der Menschen die Straße mit einer höheren Geschwindigkeit überquert.

7.2.2.3 Konklusion

Die Idee, Kartenmaterial von urbanen Umgebungen mittels Sensordaten aus der Bevölkerung anzureichern, ist definitiv gut, doch die Umsetzung ist in der Fallstudie, die in [Bu12] beschrieben wird, schwierig. Das Anwendungsszenario, Ampeln durch Accelerometer-Daten zu erkennen, ist mit dem vorgestellten Algorithmus problematisch, da zu viele Unsicherheitsfaktoren existieren. Als Pilotprojekt, das weitere Untersuchungen auf diesem Bereich ermöglicht, ist es aber auf alle Fälle geeignet. Diese Fallstudie zeigt auf, wie kreativ die Sensordaten von Smartphones genutzt werden können.

7.2.3 User-Authentifizierung über Fotos vom Ohr

7.2.3.1 Allgemeines

Studien zufolge wünschen sich die Verwender von Smartphones sicherere und vor allem einfachere User-Authentifizierungsmethoden als beispielsweise das Zeichnen von Mustern auf dem Display oder die Eingabe von Passwörtern und PIN-Codes. Dieser Wunsch war die Initialzündung für den Versuch, die biometrische Ohrform für die Erkennung des Anwenders zu verwenden. Wie in [Fa12] zu lesen ist, stellte Dr. Imhofer fest, dass sich das biometrische Ohr im Gegensatz zur Gesichtsform oder zu anderen Teilen des menschlichen Körpers im Laufe des Lebens nicht verändert. Daher ist das Ohr oder genauer gesagt die Ohrform ein interessanter Ansatzpunkt für die User-Erkennung.

7.2.3.2 Ablauf

Im Zuge eines Telefonats macht die Frontkamera des Smartphones ein Bild vom Ohr, ohne dass der Anwender Notiz davon nimmt. In weiterer Folge wird die Erkennung der Ohrform mittels Local-Binary-Pattern-Operator (kurz LBP), der laut [Fa12] erstmals in [Oj96] vorgestellt wurde, und geometrischer Analyse durchgeführt. Der Operator vergleicht die

Grauwerte der acht Nachbarpixel mit dem Wert des Mittelpixels als Threshold. Wird der Grenzwert überschritten, wird das dem Pixel zugeordneten Gewicht notiert und beim Resultat, das im letzten Schritt vom binären in das dezimale Zahlensystem transformiert wird, berücksichtigt (vgl. Abbildung 57: Positive Nützung von Smartphone-Sensoren – Local Binary Pattern [Fa12]).

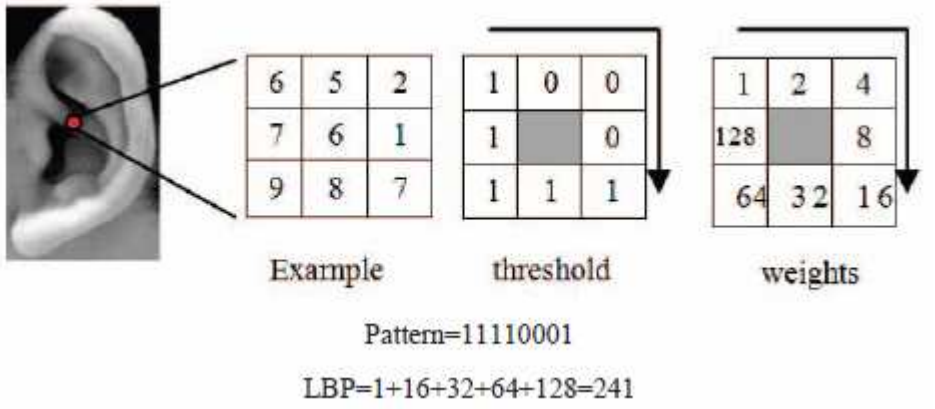

Abbildung 57: Positive Nützung von Smartphone-Sensoren – Local Binary Pattern [Fa12]

In weiterer Folge wird eine geometrische Analyse vorgenommen. Dabei werden die Ohrbeschaffenheiten Antihelix, Tragus, Antitragus und Crus of Helix mittels eines Fotos vom Ohr, das von der Frontkamera des Smartphones gemacht wird, untersucht. Die Vorteile davon sind, dass man das Smartphone nicht umständlich vor das Gesicht halten muss, sondern instinktiv das Gerät zum Ohr hält und selbst Rotationen des Smartphones nur minimale Auswirkungen auf das Ergebnis haben.

Nach einer Normalisierung können die Resultate der beiden Analysen vereint werden, und so erhalten die Verfasser von [Fa12] insgesamt 61 Features, wovon 59 Werte vom LBP-Operator und zwei Werte von der geometrischen Analyse stammen.

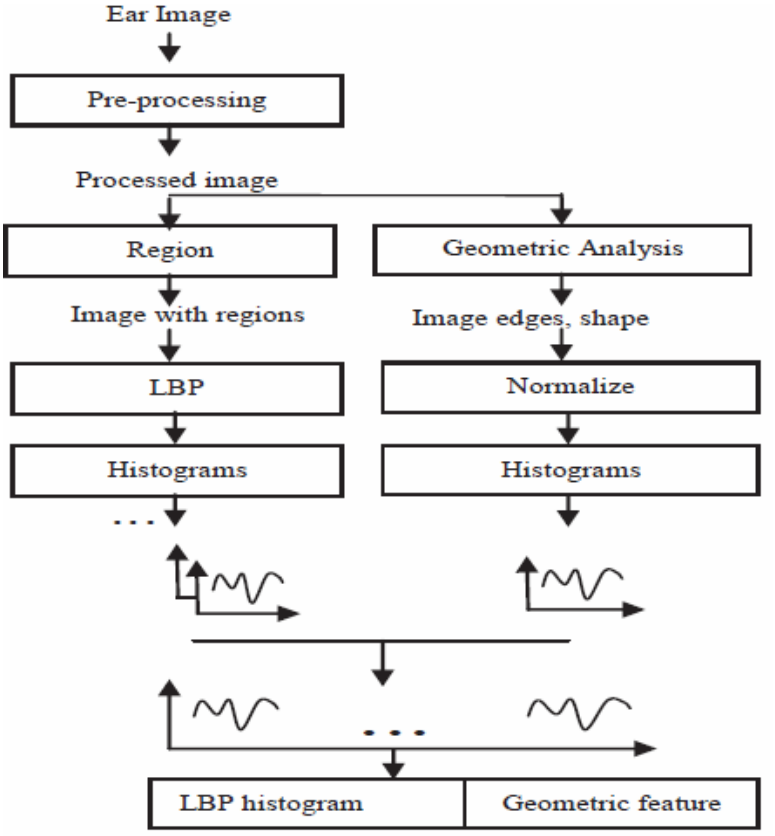

Abbildung 58: Positive Nützung von Smartphone-Sensoren – Ohr-Erkennungsalgorithmus [Fa12]

7.2.3.3 Ergebnisse

Das Ergebnis dieser Schritte ist eine verwendbare Repräsentation des Ohrs, die abschließend mittels Nearest-Neighbour-Algorithmus klassifiziert wird. Das Resultat überzeugt, denn in 92,5 % der Fälle wurden die Anwender richtig erkannt.

7.2.4 Analyse des Fahrverhaltens

7.2.4.1 Allgemeines

Im Straßenverkehr unterscheidet man zwei Fahrweisen: aggressive und nicht aggressive. Studien zufolge ist eine aggressive Fahrweise der häufigste Grund für Verkehrsunfälle. Es gab deshalb einige Forschungen in diesem Bereich, um die Straßen für alle Verkehrsteilnehmer sicherer zu machen. Die Verfasser von [De11] griffen diese Forschungen auf und überlegten sich eine kostengünstige und zugleich effiziente Methode, um aggressive Fahrstile zu erkennen: MIROAD. Dieser Begriff ist die Abkürzung für A Mobile-Sensor-Platform for Intelligent Recognition Of Aggressive Driving. Wie man aus dem Namen ableiten kann, wird für dieses System eine Mobile-Sensor-Plattform verwendet. Smartphones bieten sich dafür

regelrecht an: Mit eingebauter Kamera, Gyrosensor, Accelerometer und enormer Rechenkapazität wird in einem Stück Technik alles vereint, was für die Erkennung des Fahrstils nötig ist.

Das Smartphone wurde beim Experiment in [De11] mit Halterung zentral am Armaturenbrett bzw. an der Windschutzscheibe fixiert, sodass die Kamera immer in Fahrtrichtung ausgerichtet war (vgl. Abbildung 59: Positive Nützung von Smartphone-Sensoren – MIROAD-Positionierung [De11]). Im Gegensatz zu vielen anderen Arbeiten in diesem Bereich entschieden sich die Verfasser von [De11] nicht nur für die Auswertung der Beschleunigungsdaten des Accelerometers, sondern dafür, eine Fusion aus den folgenden Sensoren für die Erkennung zu verwenden:

- Magnetometer
- Gyroscope
- Accelerometer

Abbildung 59: Positive Nützung von Smartphone-Sensoren – MIROAD-Positionierung [De11]

7.2.4.2 Ablauf

Mithilfe der aufgelisteten Sensordaten kann MIROAD unter anderem aggressive Brems-, Kurven- und Umkehrmanöver sowie extreme Geschwindigkeit detektieren, während normales Fahrverhalten laut [De11] nicht vom Sensorrauschen unterschieden werden kann. Da das Überschreiten von Tempolimits und extreme Bremsmanöver leicht über Thresholds erkannt

werden können, konzentrierten sich die Verfasser von [De11] auf die Detektion von aggressiven Kurvenmanövern.

MIROAD überwacht ständig das Fahrverhalten und teilt die aufgezeichneten Daten (Videomaterial, aber auch die Sensordaten) in fünfminütige Timeslots. Sollte nach den fünf Minuten keine aggressive Fahrweise detektiert worden sein, wird die Aufzeichnung sofort gelöscht. Somit wird die Festplatte nicht überladen, sondern nur mit wirklich verwertbaren Daten bespielt. Optional könnte man mittels 3G-Internetverbindung ein Alarmsignal zu einem externen Server schicken und die geografische Position des Fahrzeugs sowie das erkannte Event übertragen. So könnten im Falle eines Unfalls sofort die Rettungskräfte vor Ort sein oder die Daten bei der Aufklärung eines Unfallhergangs herangezogen werden.

7.2.4.3 Ergebnisse

Um die Daten auszuwerten, entschieden sich die Verfasser von [De11] für die Anwendung des Classic-DTW-Algorithmus, der erstmals in [Sa78] als Spracherkennungstechnik vorgestellt wurde, und für eine Endpoint Detection, um eine aggressive von einer normalen Fahrweise zu unterscheiden. Für den Versuch wurden über 200 verschiedene Fahrmanöver in urbanen und ländlichen Gebieten sowie auf Autobahnen aufgenommen, wovon 50 als aggressiv zu werten sind. Am auffälligsten sind die Ergebnisse bei der Erkennung von Umkehrmanövern. Wenn man nur die Daten des Accelerometers verwendet, werden nur rund 23 % der aggressiven Umkehrvorgänge erkannt, während mithilfe der Gyrosensor-Daten 46 % detektiert werden. Kommen beide in Kombination zur Anwendung, steigt die Erkennungsrate auf 77 %.

Insgesamt wurden 97 % aller aggressiven Fahrmanöver detektiert, wodurch gezeigt wurde, dass MIROAD ein mobiles, effektives und vor allem kostengünstiges System ist, das den Straßenverkehr sicherer machen kann.

7.2.5 Sturzdetektion

7.2.5.1 Allgemeines

Die Erkennung verschiedener Aktivitäten ist aber erst der Anfang, denn wenn man diese richtig detektiert, kann man zahlreiche sinnvolle und vor allem nützliche Erkenntnisse ableiten: zum Beispiel den Sturz einer Person. Besonders für allein lebende oder ältere

Personen wäre die Erkennung von solchen Umständen nicht nur wichtig, sondern unter Umständen auch lebensrettend. Laut den Verfassern von [Vi12], die sich auf [Lu04] beziehen, können Stürze nach aktuellem Stand der Technik über drei Wege detektiert werden:

- Videodaten
- Akustische Daten
- Am Körper getragene Sensoren

Durch den Umstand, dass Smartphones mittlerweile mit unzähligen leistungsfähigen Sensoren ausgestattet sind, können diese statt unbequemer Sensoren am Körper für die Erkennung von Bewegungen verwendet werden. Die sehr verbreiteten Smartphones wurden auch von den Verfassern von [Vi12] für ein Experiment verwendet, die das „Sturz-Event“ in vier Kategorien unterteilen, da man im Vorfeld nicht klar sagen kann, ob der Anwender das Smartphone während des Sturzes in der Hosentasche, in der Hand oder beim Ohr hat. Ein Sturz kann stets passieren, weshalb das Fallen in Kategorien [Vi12] unterteilt wurde. Damit werden zwar noch nicht alle Fälle abgedeckt, aber wesentlich mehr als in vielen anderen Studien oder Versuchen.

7.2.5.2 Kategorien inklusive Beschreibung

- Kategorie 1: Sturz, während der Anwender SMS tippt
- Kategorie 2: Sturz, während der Anwender das Smartphone beim Ohr hält (z. B. beim Telefonieren)
- Kategorie 3: Sturz, während das Smartphone in der Brusttasche ist
- Kategorie 4: Sturz, während das Smartphone in der Hosentasche ist

In [Vi12] kam ein Google Nexus Smartphone inklusive eingebautem Accelerometer und Orientation-Sensor zum Einsatz. Die Werte dieser beiden Sensoren wurden für die Erkennung eines Sturzes herangezogen und mit ausgetesteten Grenzwerten verglichen. Wie sich herausstellt, liefert der zugrunde liegende Algorithmus sehr gute Ergebnisse.

7.2.5.3 Ablauf

Das „Fall-Event“ wird in [Vi12] in drei Stufen unterteilt. In der ersten Phase fällt der Wert der Amplitude des Accelerometers extrem, während er in der zweiten Phase – beim Aufprall –

wieder enorm ansteigt. In der dritten Phase normalisiert sich der Wert wieder, da der Körper bzw. das Smartphone keine Beschleunigungen mehr erfährt, sondern auf dem Boden liegt (vgl. Abbildung 60: Positive Nützung von Smartphone-Sensoren – Accelerometer-Daten bei einem Sturz [Vi12]).

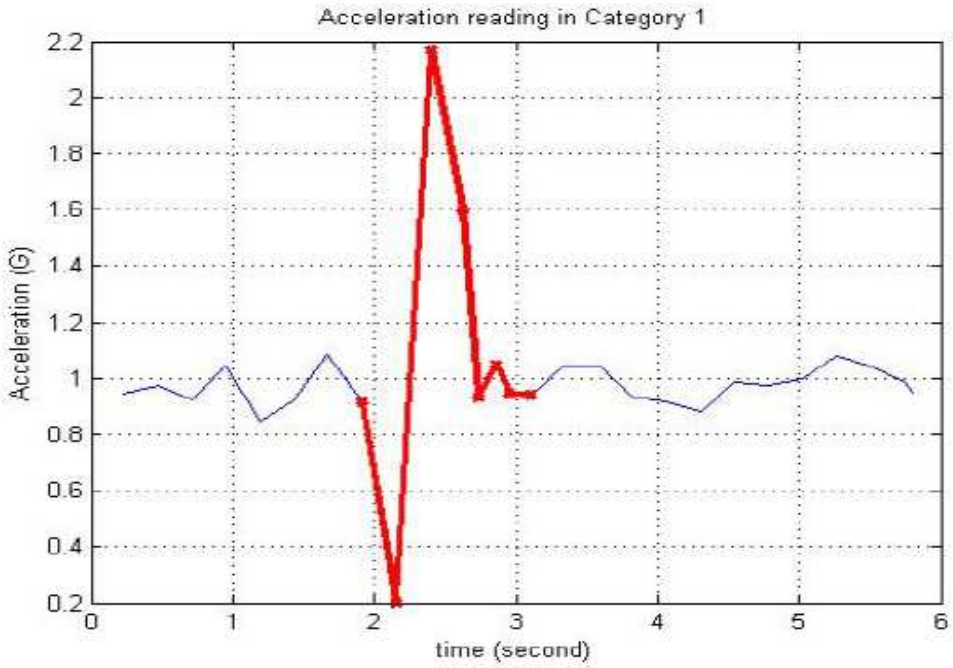

Abbildung 60: Positive Nützung von Smartphone-Sensoren – Accelerometer-Daten bei einem Sturz [Vi12]

7.2.5.4 Ergebnisse

Durch eine gute Wahl der Schranken gelang es den Verfassern von [Vi12], sehr gute Ergebnisse zu erzielen. Fünf Menschen über 23 Jahre nahmen an dem Experiment teil, und jeder Versuch wurde 15-mal wiederholt, damit man aussagekräftige Resultate ableiten kann. In [Vi12] wurde auf den Einsatz von älteren Personen verzichtet, da selbst bei Laborbedingungen immer ein Gesundheitsrisiko beim Sturz besteht. Von insgesamt 260 Testläufen wurde in 221 Fällen der Sturz erfolgreich detektiert, während der Algorithmus nur 39-mal falsche Ergebnisse lieferte. Dies entspricht einer Erkennungsrate von genau 85 %, doch wie in [Vi12] zu lesen ist, könnte das Ergebnis mit einem in Echtzeit arbeitenden Klassifizieralgorithmus noch besser ausfallen.

7.3 Verwendung von Smartphone-Sensoren für Angriffe

Die Sensoren in den Smartphones können, wie im letzten Kapitel dargelegt, verschiedentlich sinnvoll eingesetzt werden. Sie können jedoch ebenso für die Erkennung von eingegebenen Passwörtern, für Rückschlüsse auf Anwendercharakteristika und den aktuellen Aufenthaltsort, und damit auch für Angriffe verwendet werden.

7.3.1 Touchlogger

7.3.1.1 Allgemeines

Wie in [Ca11] zu lesen ist, sind Keyboards die meistgenutzte Input-Möglichkeit auf Smartphones. Es ist daher nicht verwunderlich, dass das Aufzeichnen von Tasteneingaben ein beliebtes Szenario für Angreifer ist. In der Vergangenheit konnten sich Angreifer beispielsweise die Akustik oder das Timing zwischen den Tastenanschlägen zunutze machen, um Informationen über eingegebene Passwörter oder Ähnliches zu erlangen. Die virtuellen Tastaturen erfordern bzw. ermöglichen neue Wege, um sensible Daten zu gewinnen – der in [Ca11] beschriebene Touchlogger bietet einen interessanten Ansatz dafür. Dieses Programm verwendet für die Erkennung von sensiblen Input-Daten Motion-Sensordaten. Je nachdem, welche man auf dem Display drückt, ändert sich die Ausrichtung des Smartphones. Zusätzlich verursacht das Drücken an unterschiedlichen Stellen verschiedene Vibrationen. Beide Motion-Daten verwendet der Touchlogger, um Informationen über die eingegebenen Daten zu erlangen.

7.3.1.2 Beobachtungen

Die vier wichtigsten Aspekte bei der Erkennung von Eingaben sind:

- Die Kraft des auf das Display drückenden Fingers
- Die Widerstandskraft der Hand, die das Smartphone hält
- Die Position auf dem Display, auf die gedrückt wird
- Die Position der Hand, die das Smartphone hält

Während die ersten beiden Faktoren eine Positionsverschiebung mit sich bringen, beeinflussen die anderen beiden die Rotation des Smartphones. Laut [Ca11] haben Untersuchungen ergeben, dass die ersten beiden Faktoren User-abhängig sind und die anderen

beiden vernachlässigt werden können, da die Position der Tasten auf dem Smartphone überall gleich und die Tastatur im Normalfall auch gleich aufgebaut ist. Auch die Position der Hand, die das Smartphone stützt, ist normalerweise ähnlich und daher nicht von Relevanz. [Ca11] beschreibt, wie man diese Beobachtungen nutzen kann, um aussagekräftige Werte zu extrahieren.

In [Ca11] werden die Rotationskomponenten genauer betrachtet, während die Positionsverschiebung durch das Drücken auf das Display gefiltert wird. Aus [An12] geht hervor, dass die meisten Smartphone-Betriebssysteme mindestens zwei Typen von Events abfeuern, wenn eine Interaktion mit dem Display erfolgt: Accelerometer- und Orientation-Events. Wie in [Ca11] beschrieben wird, versuchten die Verfasser zuerst die Accelerometer-Daten zu verwenden, da diese öfter ausgelöst werden als Orientation-Events und die Daten des Accelerometers Positionsverschiebungen als auch Rotationen beinhalten. Schon nach wenigen Versuchen merkte man jedoch, dass sich die Orientation-Daten wesentlich besser eignen, da sie nur Rotationsdaten enthalten.

7.3.1.3 Ablauf

Die Ausrichtung des Smartphones wird über drei Winkel und mit dem Event-Zeitpunkt beschrieben. Durch die Verknüpfung all dieser Winkel wird die Orientierung des Smartphones zu einem bestimmten Zeitpunkt abgebildet. Diese vier Werte ermöglichen Rückschlüsse auf die Position, auf die der Anwender auf dem Display gedrückt hat. Da es hier aber auch anwenderabhängige Werte, wie die Kraft des auf das Display drückenden Fingers oder die Widerstandskraft der Hand, die das Smartphone hält, gibt, muss die Touchlogger-Applikation sich erst mit dem Anwender „vertraut“ machen.

Wie nicht anders zu erwarten, geschieht das in einer Trainingsphase. Der Anwender wird aufgefordert, Tasten auf dem Display zu drücken, und die aufgezeichneten Orientation-Events werden an den Touchlogger übergeben. Beobachtungen zufolge sind die extrahierten Werte für die gleichen Eingaben Gauß-verteilt, weshalb die Verfasser von [Ca11] den Mittelwert und die Standardabweichung für jede Taste errechneten. So hat die Applikation Vergleichspattern für jede einzelne Taste und kann in späterer Folge eingegebene Passwörter oder PIN-Codes im Hintergrund klassifizieren und abspeichern.

7.3.1.4 Erfolgschancen

Für die Evaluierung der Erfolgschancen von Touchlogger wurde das in Abbildung 61: Verwendung von Smartphone-Sensoren für Angriffe – Touchlogger Applikation [Ca11] zu sehende 16-Tasten-Numpad und ein HTC Evo 4 G Smartphone als Hardware werwendet. Aufgezeichnet wurden drei Datensets im Landscape-Modus. Jedes Datenset umfasst mehrere Sessions mit vier bis 25 Tastenanschlägen, wobei alle 16 Tasten verwendet wurden.

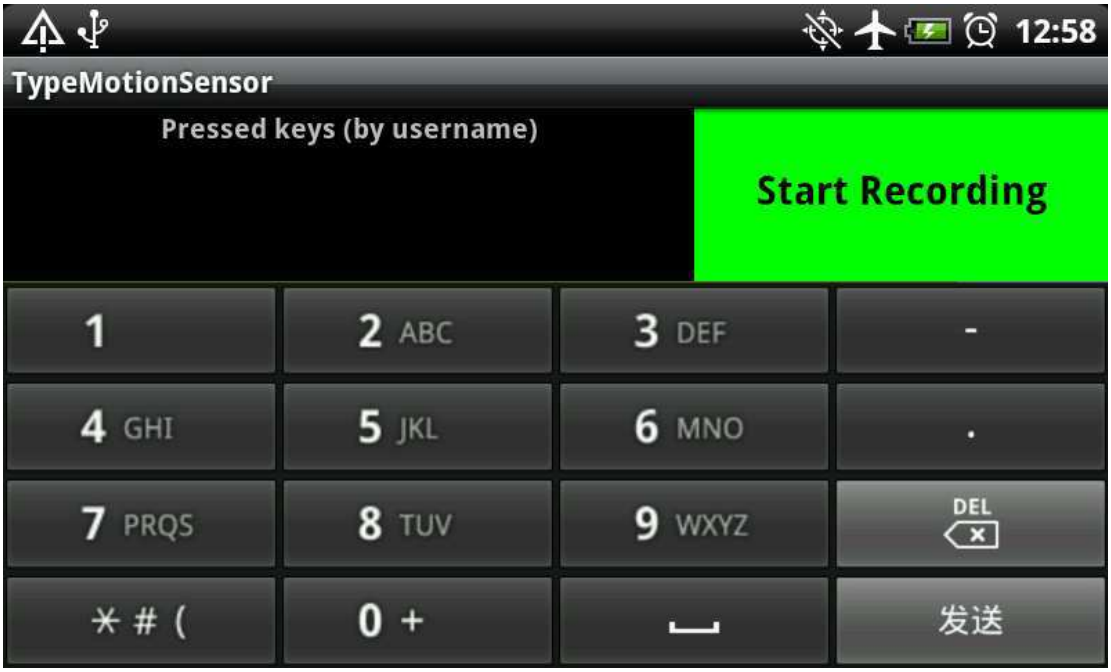

Abbildung 61: Verwendung von Smartphone-Sensoren für Angriffe – Touchlogger Applikation [Ca11]

7.3.1.5 Ergebnisse

Bei dem Versuch wurde ein beachtliches Ergebnis erzielt. Wie in [Ca11] zu lesen ist, konnte eine Trefferquote von über 70 % erreicht werden. Am besten wurden die Tasten 1 (86,3 % Trefferquote) und 9 (80,8 % Trefferquote) erkannt, die sich in den gegenüberliegenden Ecken befinden.

7.3.2 Taplogger

7.3.2.1 Allgemeines

Wie in [Xu12] beschrieben, bergen Smartphones mit all ihren Sensoren auch Risiken und Gefahren für den Bereich der Privatsphäre. Über die Motion-Sensordaten können Third-Party-Applikationen beispielsweise private Informationen auslesen. Wie das funktionieren kann, wird in den folgenden Absätzen genauer beschrieben.

[Xu12] experimentierte mit einer Trojanerapplikation, die im Hintergrund Motion-Daten aufzeichnet und dadurch Rückschlüsse auf die Eingaben ermöglicht. Ähnlich wie in [Ca11] ist das Drücken auf den Bildschirm entscheidend. Das Smartphone neigt sich komplett anders, wenn man beispielsweise auf die linke obere oder rechte untere Ecke drückt (vgl. Abbildung 62: Verwendung von Smartphone-Sensoren für Angriffe – Taplogger-Beobachtung [Xu12]).

Wenn der Angreifer nun auch etwas über die auf dem Display dargestellte View in Erfahrung bringt, kann er daraus Erkenntnisse gewinnen. Weiß man beispielsweise, wo der Anwender auf dem Display hingeklickt hat und was sich an dieser Stelle befindet (z. B. eine Zahlentastatur), weiß man folglich auch, welche Daten wie und wo eingetragen wurden (z. B. PIN-Code beim E-Banking).

Das Ziel in [Xu12] war es, sinnvolle User-Inputs, wie zum Beispiel den gedrückten Button während eines Tap-Events, zu eruieren, anstatt die exakten Koordinaten des Tap-Events auszulesen. Die genauen Daten zu ermitteln ist wegen des Hintergrundrauschens relativ schwierig. Außerdem ist es für den Angreifer viel interessanter, was auf dem Softkeyboard oder dem Display gedrückt wurde, und nicht die genaue Tap-Position.

7.3.2.2 Ablauf

Ähnlich wie beim Touchlogger aus [Ca11] gibt es auch beim Taplogger, der in [Xu12] näher vorgestellt wird, zwei Phasen beziehungsweise Modi: Training Mode und Logging Mode.

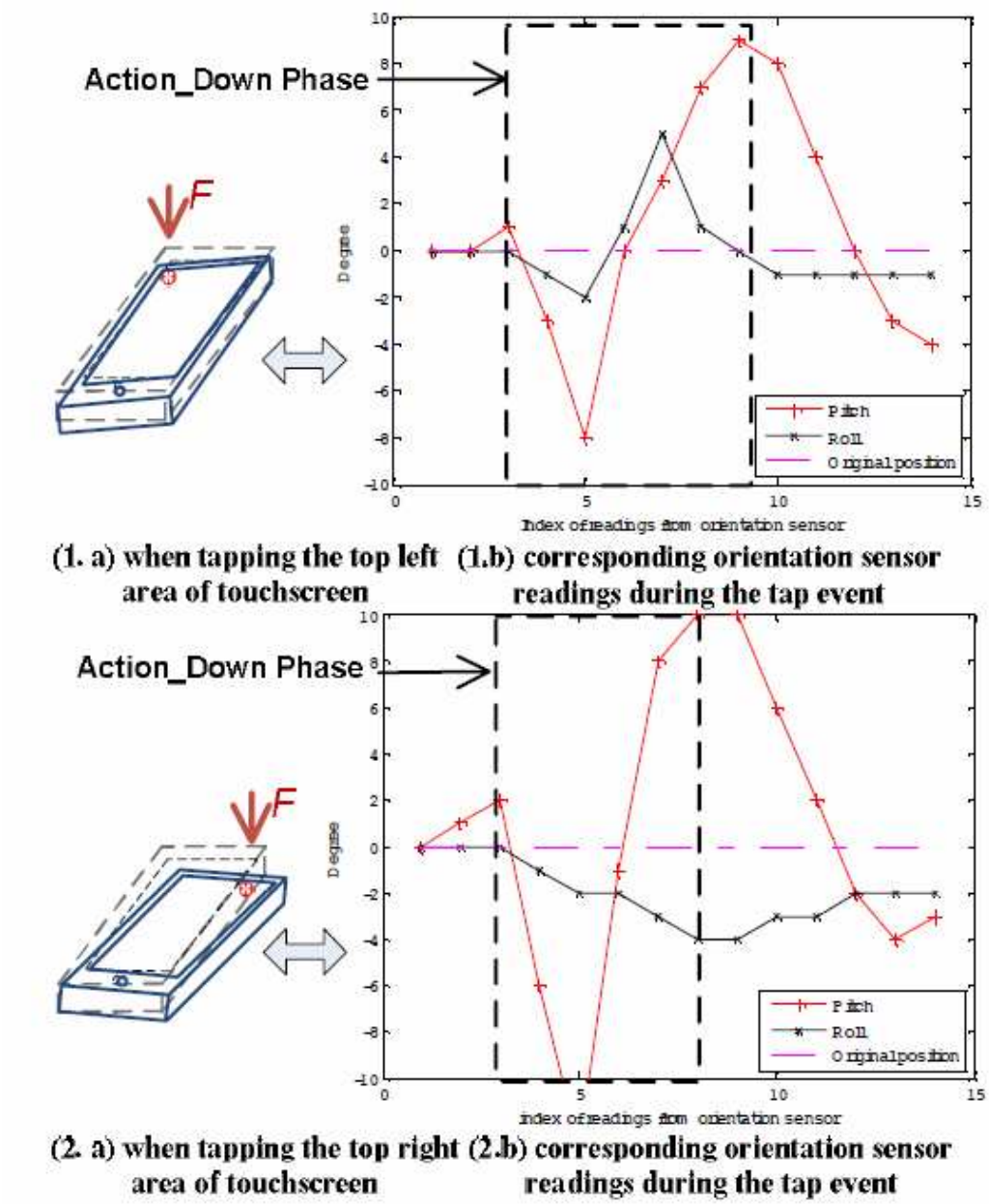

Abbildung 62: Verwendung von Smartphone-Sensoren für Angriffe – Taplogger-Beobachtung [Xu12]

Wie nicht anders zu erwarten, lernt die Applikation im Training Mode die User-Inputs zu verarbeiten. Das erfolgt durch die Interaktion des Users mit der Host-App. Über die gesammelten Orientation- und Accelerometer-Daten werden User-Interaktionspatterns erstellt, mit denen die Eingaben im Logging Mode verglichen werden. Im Gegensatz zum Training Mode läuft im Logging Mode der SensorListener verdeckt im Hintergrund und übermittelt die aufgezeichneten Daten.

Abbildung 63: Verwendung von Smartphone-Sensoren für Angriffe – Taplogger-Modi

7.3.2.3 Ergebnisse

Für das Experiment in [Xu12] wurden jeweils fünf zufällig generierte Passwörter mit der Länge von 4,6 und acht Zeichen erstellt. Vor der Attacke wurden die Anwender aufgefordert, 60-mal das Paarspiel mit der Host-App zu absolvieren, um genügend Trainingsdaten zu sammeln. In weiterer Folge mussten die Probanden die zufällig erstellten Passwörter jeweils 30-mal im Logging Mode eingeben. Die aufgezeichneten Daten wurden mit einem Klassifizierer, der die LibSVM aus [Ch11] verwendet, verarbeitet. Das Ergebnis dieses Versuchs ist beeindruckend.

Abhängig von den akzeptierten Labels wurden Erkennungsraten von bis zu 100 % erreicht. Dieses gute Ergebnis wurde allerdings nur mit der Verwendung von den drei Top-Labels erzielt. Das bedeutet, dass bei jeder Eingabe einer Zahl die drei am häufigsten erkannten Zahlen berücksichtigt wurden. Interessant ist auch, dass die Erkennung von Sechs-Zahlen-Passwörtern im Experiment schlechtere Ergebnisse lieferte als die von Acht-Zahlen-

Passwörtern. Dies rührt laut den Verfassern von [Xu12] daher, dass die Zufallspasswörter, die sechs Zahlen umfassten, viele Zahlen in der Tastaturmitte enthielten.

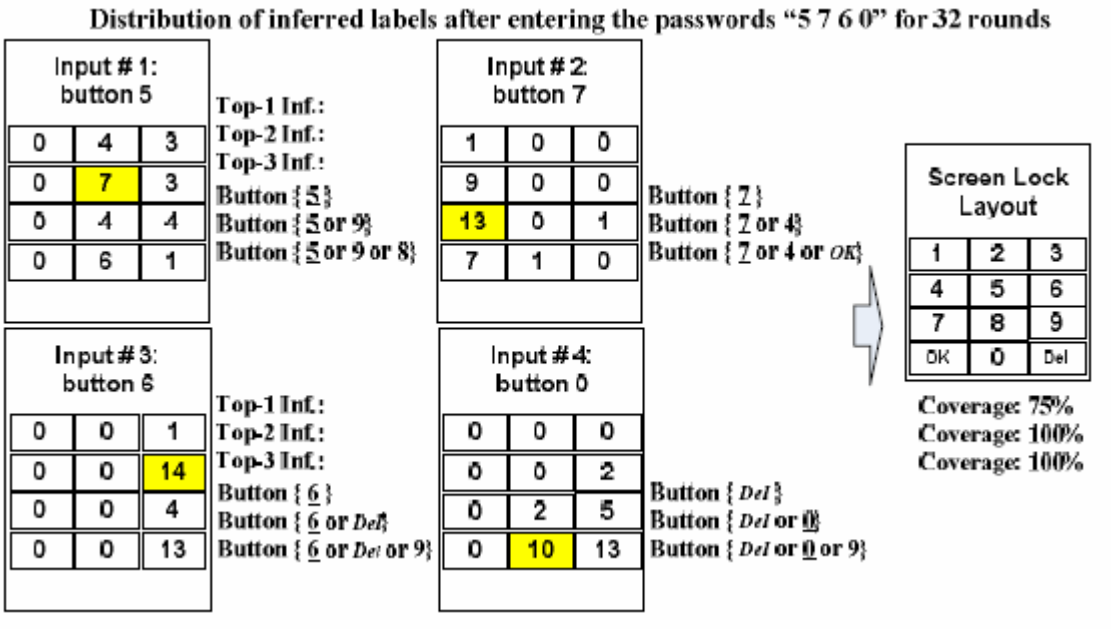

Abbildung 64: Verwendung von Smartphone-Sensoren für Angriffe – Taplogger-Labels [Xu12]

7.3.2.4 Erfolgschancen

Da die Motion-Sensordaten als unproblematisch gesehen werden, benötigt der Taplogger laut [Xu12] keine Security Permissions, um Zugriff auf den Accelerometer oder die Orientation-Sensordaten zu erhalten. Dies gilt für Android-, iOS und Blackberry-Geräte. Offensichtlich muss man die „Spielapplikation" im Logging-Modus sehr oft mit Daten versorgen – also spielen – und die Passwörter mehrmals eingeben, um konkrete Rückschlüsse auf die Eingaben zu ziehen. Die Verfasser von [Xu12] haben mit diesem Experiment aber aufgezeigt, dass man ohne Zugriff auf sensible Daten Passwörter, PINs oder Ähnliches nur durch Verwendung der Motion-Sensordaten entschlüsseln kann.

7.3.3 Rückschlüsse auf Orte und soziales Verhalten auf Basis von Accelerometer-Daten

7.3.3.1 Allgemeines

Die Verfasser von [Ha12] haben in einer wissenschaftlichen Arbeit einen weiteren die Privatsphäre bedrohenden Aspekt beleuchtet. Sie haben die auf den ersten Blick harmlos

scheinenden Accelerometer-Daten verwendet, um damit Rückschlüsse auf den Aufenthaltsort des Smartphones zu ziehen. Damit eine Location Interference wirklich funktioniert, muss sich das Smartphone in Bewegung befinden und das Areal abgesteckt sein. Im Versuch, der in [Ha12] dokumentiert ist, wurden zwei Testgebiete mit einmal 11 x 10 und einmal 12 x 10 Kilometern verwendet.

7.3.3.2 Wie funktioniert die Location Interference via Accelerometer-Daten?

Der Clou bei der Location Interference ist die Erstellung von Trajektorien auf Basis von Accelerometer-Daten, die für alle Smartphone-Besitzer an diesem Ort gleich sind. Fährt man beispielsweise über die Großglockner-Hochalpenstraße, werden dieselben Erschütterungen und Kurvenbeschleunigungen beim gleichen Straßenkilometer ausgelöst. Hat User A nun seine GPS-Daten und die Internetverbindung freigegeben, dann kann zu dieser Koordinate ein Referenzpunkt in der Datenbank abgelegt werden. Kommt nun User B zu diesem Punkt und liefert der Accelerometer die gleichen Trajektorien, kann man die Position von User B zu diesem in der Datenbank abgelegten Referenzpunkt zuordnen. Zwar gibt es mit Sicherheit eine Vielzahl von Orten, die auf diesen Trajektorienwert zutreffen können, aber wenn man beispielsweise private Daten des Anwenders wie seine E-Mail-Adresse, bevorzugte Aufenthaltsorte oder seinen Namen kennt, lässt sich das Gebiet noch weiter eingrenzen.

Dieses Thema wird auch in [Ke13] behandelt. Darin werden von gewonnenen Ortsdaten Erkenntnisse über demografische und soziale Gegebenheiten abgeleitet. So konnten erfolgreich Charakteristiken wie Alter, Geschlecht, reguläre Verkehrsmittel oder Treffen mit nahestehenden Personen (Eltern, Großeltern oder Arbeitskollegen etc.) detektiert werden. Vereint man beide Themen, wird ein extremes Sicherheits- und Privatsphärenproblem aufgezeigt.

Zwar muss zuerst eine App installiert werden, die die Daten sammelt, aber wenn bei den Zugriffsrechten der App nur auf „Accelerometer-Daten“ hingewiesen wird, werden durchschnittliche User keinen Verdacht schöpfen, zumal eine Vielzahl von erfolgreichen und verbreiteten Mobile Games wesentlich umfangreichere Zugriffsrechte verlangen.

8 Ausblick in die Zukunft

In [Ki11] werden einige interessante Ansatzpunkte für weitere Forschungen im Bereich Einsatz von Sensoren für den guten Zweck aufgelistet. Angefangen von unterstützenden bis hin zu lebensrettenden Applikationen werden einige Einsatzszenarien für Smartphone-Sensoren kurz vorgestellt. Abgesehen von den in [Ki11] vorgestellten Bereichen wurde die Liste um einige eigene Ideen und potenzielle Einsatzgebiete erweitert.

8.1 Unterstützung bei Ernährung und Flüssigkeitszufuhr

Seit Jahren ist bekannt, dass fettes und würziges Essen schlecht für die Gesundheit ist, und man öfter zu Früchten, Obst und weißem Fleisch greifen sollte. Generell ist Adipositas ein großes gesellschaftliches Problem – wie könnte diesem ausgerechnet ein bequemes Gerät wie ein simples Smartphone entgegenwirken? Der Verfasser von [Ki11] schlägt vor, mithilfe von Smartphones zum Beispiel zur Ausübung von Sport anzuregen, den Kalorienhaushalt zu kontrollieren sowie auf eine ausreichende Flüssigkeitszufuhr zu achten. Vor allem die beiden letztgenannten Punkte könnten relativ einfach umgesetzt werden.

Durch einen Routine-Arztbesuch könnte man den durchschnittlichen Flüssigkeits- und Kalorienbedarf abschätzen lassen. In weiterer Folge könnte man in einer App die Art des Getränks oder der Nahrung sowie die Menge eintragen und so einen Überblick über die tägliche Flüssigkeits- bzw. Kalorienzufuhr gewinnen. Hier könnte auch der im Anschluss skizzierte Fotodatenbankabgleich zum Einsatz kommen.

Wie genau die Unterstützung beim Kalorienhaushalt funktionieren soll, wird in [Ki11] nur kurz angerissen. Demnach soll man beim Lebensmitteleinkauf die Kalorien einfach zusammenzählen, während man sich in einem Restaurant einfach via Rezept über die Bestandteile des Menüs informieren soll. Die Kalorien sind bekannt (z. B. Datenbank), womit man die gesamte Anzahl bestimmen kann.

In [Ki11] wird weiters die Möglichkeit der Kamera-Integration skizziert. Demnach sollte ein Foto des Gerichts (beispielsweise Makkaroni mit Käse) ausreichen, um Rückschlüsse auf die Mahlzeit zu ziehen. Ein Datenbankabgleich mit dem Foto soll ausreichen, um die Speise zu erkennen und die Kalorien dieses Gerichts abzuschätzen. In Anbetracht der optischen

Ähnlichkeit von Mahlzeiten und der nicht erkennbaren Menge, die eingenommen wird, ist dieser errechnete Wert wohl sehr ungenau – zudem ist diese Vorgehensweise nicht besonders praktisch.

8.2 Sport/Bewegung

Aufgrund von Körpergröße und Muskelstruktur kann der durchschnittliche Kalorienbedarf pro Tag errechnet werden. Wenn man auch weiß, wie viele Kalorien man im Tagesverlauf zu sich nimmt, kann man auch berechnen, wie viel Bewegung nötig ist, um den Kalorienhaushalt auszugleichen. Die durchgeführte Bewegung könnte laut [Ki11] mit einem einfachen Interface auf dem Smartphone vermerkt werden. Man könnte zwar auch die GPS-Daten verwenden, doch diese sind zu ungenau und zu wenig aussagekräftig, weshalb hier das Smartphone nicht ohne Interaktion auskommt. Weiters könnte man vermerken, welche Arbeit oder Tätigkeit man verrichtet: ob man sich beispielsweise mit dem Fahrrad durch die Stadt bewegt oder mit einem Linienbus unterwegs ist. Aus der Tätigkeit und der Dauer lässt sich der Energiebedarf leicht errechnen, wie zum Beispiel aus der Tätigkeit „Joggen“: Mittels GPS-Signal könnte das Smartphone die Geschwindigkeit und Distanz ableiten, dennoch müsste man dem Smartphone mitteilen, dass man joggt und nicht beispielsweise mit einem Fahrrad fährt, da der Kalorienverbrauch in diesen beiden Fällen Unterschiede aufweist.

8.3 Verkehrssicherheit

Im Straßenverkehr könnten Smartphones eine entscheidende Rolle einnehmen. So könnten sie beispielsweise im Fall von Betrunkenheit das menschliche Interagieren mit dem Auto, wie das Verwenden des Lenkrads oder der Pedale, unterbinden. Weiters könnte das Smartphone eine SMS an eine Taxi-Gesellschaft schicken, damit die nicht fahrfähige Person abgeholt und nach Hause gebracht wird.

Ein weiteres Anwendungsszenario wird in [Ki11] beschrieben: Demnach könnte das Smartphone ein Signal an das Auto schicken, wenn die Person am Steuer nicht mehr in der Lage ist, das Auto sicher zu fahren (z. B. durch plötzliches Unwohlsein). Das könnte bewirken, dass das Auto die Kontrolle übernimmt, die Warnblinkanlage aktiviert und unter Berücksichtigung des Verkehrs an den Straßenrand fährt. In der Zwischenzeit könnte das

Smartphone die Rettung verständigen und die gemessenen Werte direkt an die Helfer übermitteln.

An dieser Stelle möchte ich noch einmal auf das in dieser Arbeit näher vorgestellte MIROAD-System verweisen, das aggressive Fahrweise detektieren kann. Sollte dies der Fall sein, könnte das Smartphone auch ein Signal an das Fahrzeug schicken, das ebenfalls unter Berücksichtigung des Verkehrs das Auto am Straßenrand zum Stillstand bringt.

8.4 Biometrische User-Authentifizierung

Ein weiterer interessanter Ansatzpunkt wäre eine verbesserte User-Authentifizierung. Dafür könnte man biometrische Daten wie die Handgeometrie, den Fingerabdruck, die Stimme (Klangfarbe) oder beispielsweise das Tippverhalten verwenden. Aber auch die Ohrform ist ein biometrischer Wert, der dafür herangezogen werden könnte (vgl. Abschnitt User-Authentifizierung über Fotos vom Ohr). Doch wie die Verfasser in [Fa12] auch klar dokumentieren, wäre die Erkennung der Ohrform wohl nur ein Teilaspekt einer biometrischen User-Authentifizierung.

Zusammen mit anderen Parametern, wie zum Beispiel der Klangfarbe der Stimme, könnten Smartphone-User schnell, unkompliziert und eindeutig erkannt werden. Beispielsweise könnte das Smartphone das Telefonieren verweigern, wenn die Biodaten nicht mit denen des Standard-Users übereinstimmen. Ein interessanter Nebenaspekt ist, dass man sich keine komplizierten Passwörter mehr merken müsste, sondern sich einfach auf seine biometrischen Daten verlassen könnte. Überdies ist es wesentlich schwerer, diese Daten zu fälschen, als ein Passwort zu hacken.

8.5 Unterstützung in der Altenversorgung

Im Alter verbringen Menschen oft Zeit alleine oder ohne Aufsicht. Sollte es in diesem Fall zu einem medizinischen Notfall, wie zum Beispiel zu einem Herzstillstand, kommen, zählen Minuten oder Sekunden. Hier könnte laut [Ki11] erneut das Smartphone unterstützen und die veränderten oder schwächer werdenden Lebenszeichen an den Notarzt schicken. Vonstattengehen könnte dies durch die Messung von Vitalwerten wie der Körpertemperatur, des Sauerstofflevels im Blut und der Herzschlagfrequenz.

Sollte Bewusstlosigkeit erkannt werden, könnte ebenfalls ein Signal an den Notarzt geschickt werden. Weiters könnte das Smartphone auch als Erinnerungsgerät dienen, das den Patienten an die Einnahme von Tabletten oder Ähnliches erinnert (z. B. bei Alzheimer).

8.6 Testautomatisierung

Die im Praxisteil erläuterten Möglichkeiten der Sensordatenintegration für das Testen von Apps sind nur einfache Beispiele. Wegen der großen Genauigkeit der Sensoren und der Verbreitung von Smartphones steht den Entwicklern eine Fülle an Möglichkeiten zur Verfügung, um Apps in Zukunft noch robuster und zuverlässiger zu machen.

So könnten der Gyrosensor und der Accelerometer für die korrekte Erkennung von Neigungswinkeln des Smartphones verwendet werden. Die Verwendung des SensorSimulators von Open Intents oder des Test-Automation-Framework Robotium könnte die Möglichkeiten noch erweitern.

8.7 Makros via Sprachbefehle

Weiters könnte man beispielsweise Sprachbefehle mittels Mikrofon realisieren. Man könnte vorgefertigte Makros erstellen, die nach der Aufnahme der Befehle und einem Datenbankabgleich von allein ausgeführt werden. Zum Beispiel: Man diktiert eine Einkaufsliste, die per E-Mail an den Nahversorger übermittelt wird, der wiederum die gewünschten Waren zustellt. Eine solche Möglichkeit könnte vor allem gebrechlichen und älteren Menschen das Leben entscheidend erleichtern.

8.8 Bezahlen via Dockingstation

Via Dockingstation könnte man die Bankomatkarte in Zukunft ablösen. Durch die Eingabe der PIN auf dem Display oder durch biometrische Daten (z. B. Handform und Fingerabdruck) könnte die Transaktion gestartet werden. Auch die Kundendaten könnten im Smartphone gespeichert werden, womit in einem nächsten Schritt die Brieftasche mit Geld und Kundenkarten aus dem Alltag verschwinden könnte.

8.9 Zusatzinformationen beim Betreten eines Geschäfts

Beim Betreten eines Supermarkts oder Fachgeschäfts könnte das Smartphone eine gesicherte Verbindung mit dem Infopoint des gerade betretenen Geschäfts herstellen und die aktuellen Angebote abrufen. So würde man bestens informiert durch das Geschäft gehen. Weiters könnte man einen Lageplan direkt auf das Handy spielen, was beispielsweise bei weitläufigen Geschäften wie Möbelhäusern oder Baumärkten sehr hilfreich wäre. Das Smartphone könnte den Anwender in weiterer Folge direkt zum entsprechenden Angebot oder zur jeweiligen Abteilung leiten.

8.10 Navigation für Blinde

Dank der eingebauten Kamera, Google Street und dem GPS-Senor könnten Smartphones die Blindenführer der Zukunft werden. Per Sprachausgabe könnte das Smartphone den Anwender durch die Stadt führen. Man brauchte nur vorab das Ziel mittels Sprache eingeben und dann das Smartphone mit der Kamera in die eigene Blickrichtung halten. Durch einen Datenbankabgleich mit Google Street könnte das Smartphone den Anwender zu seinem Bestimmungsort führen, indem es ihm einfach die Richtung, die Entfernung, Kreuzungen und Ähnliches ansagt.

9 Das Testgerät

Abschließend folgt noch ein Überblick über das Testgerät, das vor allem für den praktischen Teil dieser Masterarbeit verwendet wurde. Besonderes Augenmerk wurde auf die technischen Spezifikationen, die im Projekt Verwendung finden, gelegt.

Abbildung 65: Das Testgerät (HTC Desire) [Ht10]

9.1 Technische Details im Überblick [Ht10]:

- **Abmessungen:** 119 mm x 60 mm x 11,9 mm (l x b x h)
- **Displayauflösung**: 480 x 800 Pixel
- **Prozessor**: 1 GHz
- **Speicher**:
  - 512 MB ROM
  - 576 MB RAM
- **Akku**: Kapazität: 1.400 mAh
- **Anschlüsse**:
  - 3,5-mm-Audioanschluss
  - Standard-Mikro-USB-Anschluss
- **OS:** Android 2.1
- **Internet**:
  - 3 G (max 7,2 Mbit/s)

    - GPRS (max 114 bit/s Download)
- **Kamera:** 5 Megapixel
- **Bluetooth** 2.1 mit Enhanced Data Rate
- **Multimedia:**
    - Musik (.aac, .amr, .ogg, .m4a, .mid, .mp3, .wav, .wma)
    - FM-Radio
    - Video (.3gp, .3g2, .mp4, .wmv). Aufnahme in .3gp
- **Sensoren:**
    - G-Sensor
    - Digitaler Kompass
    - Näherungssensor
    - Umgebungslichtsensor
    - Interne GPS-Antenne

10 Zusammenfassung

In dieser Arbeit wurden mit Extreme Programming und testgetriebener Entwicklung zwei Konzepte theoretisch aufgearbeitet, die ebenso zukunftsträchtig sind wie die umfangreiche Einbindung und Verwendung der leistungsstarken Sensoren von Smartphones. Im Praxisteil wurden die Prozesse der testgetriebenen Entwicklung vertieft und gängige Entwicklungstools sowie -techniken inklusive Beispiele vorgestellt. Um die Bedeutung der in Smartphones eingebauten Sensoren zu verdeutlichen, wurden zwei Möglichkeiten für die Verbesserung von Funktionstests vorgestellt. Es wurde skizziert, dass mithilfe des Mikrofons und einer Geräuschschranke die korrekte Funktion des Smartphone-Lautsprechers und mithilfe der Orientation-Sensordaten das Vibrieren des Smartphones detektiert werden kann. Abseits der Anwendung zur Verbesserung von Testfällen wurde eine Reihe von anderen positiven Einsatzgebieten, wie zum Beispiel die Detektion von Bewegungsmustern, die User-Erkennung über Fotos vom Ohr oder die Analyse von Fahrverhalten, aufgezeigt. Die potenziellen Smartphone-Sensoren können allerdings auch im negativen Sinn, wie zum Beispiel für Eingriffe in die Privatsphäre, verwendet werden. Durch diesen Abschnitt soll die Gefahr, die von den Sensoren ausgeht, ins Bewusstsein gerufen werden.

Diese Arbeit spiegelt den Status quo in der Sensorverwendung auf Smartphones wider. Es gibt bereits einige interessante Anwendungsszenarien, allerdings ist das Potenzial noch lange nicht voll ausgeschöpft. In Kapitel 8 Ausblick in die Zukunft werden nur ein paar der vielschichtigen Möglichkeiten genannt, wie Smartphone-Sensoren das Leben schon in naher Zukunft erleichtern können. Weiters könnten in Zukunft auch der Power-Aware- sowie der Context-Aware-Computing-Aspekt, die Speicherung der Daten für die Analyse von Userverhalten oder die Einbindung von externen Sensoren, wie einer Pulsuhr oder Ähnliches, in Angriff genommen werden, wodurch die Vielschichtigkeit der Anwendungsgebiete noch größer wird. Die Forschung und Integration von Smartphone-Sensoren ist nicht mehr am Anfang – aber auch noch lange nicht abgeschlossen.

11 Literatur- und Quellenverzeichnis

11.1 Monografien

[Be99] Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley: Boston 1999

[Be04] Beck, K.: Extreme Programming Explained: Embrace Change. 2. Auflage. Addison-Wesley: Boston 2004

11.2 Wissenschaftliche Artikel

[Ab03] Abrahamsson, Pekka: Extreme Programming: First Results from a Controlled Case Study. In: Proceedings of the 29th EUROMICRO Conference. New Waves in System Architecture (EUROMICRO'03). Publication Year: 2003, pp. 259–266

[Bu12] Bujari, A.; Licar, B.; Palazzi, C.E.: Movement pattern recognition through smartphone's accelerometer. In: 2012 IEEE Consumer Communications and Networking Conference (CCNC). Publication Year: 2012, pp. 502–506

[Ca11] Cai, Liang; Chen, Hao: Touchlogger: Inferring Keystrokes on touch screen from smartphone motion. In: HotSec '11 Proceedings of the 6th USENIX conference on Hot topics in security. Publication Year: 2011, pp. 9–9

[Ch11] Chang, C.C.; Lin, C.J.: LIBSVM: A library for support vector machines. In: ACM Trans. on Intell. Syst. Technol (TIST). Vol. 2, Issue 3. Publication Year: 2011, Article 27

[Fa12] Fahmi, P.N.A.; Kodirov, Elyor; Choi, Deok-Jai; Lee, Guee-Sang; Mohd Fikri Azli, A.; Sayeed, S.: Implicit authentication based on ear shape biometrics using smartphone camera during a call. In: 2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC). Publication Year: 2012, pp. 2272–2276

[Ha12] Han, Jun; Owusu, Emmanuel; Nguyen, L.T.; Perrig, Adrian; Zhang: Joy: ACComplice: Location inference using accelerometers on Smartphones. In: 2012 Fourth International Conference on Communication Systems and Networks (COMSNETS 2012). Publication Year: 2012, pp. 1–9

[Hu08] Hussain, Zahid; Lechner, Martin; Milchrahm, Harald; Shahzad Sara; Slany, Wolfgang; Umgeher, Martin; Vlk, Thomas: Optimizing Extreme Programming. In: Proceedings of the International Conference on Computer and Communication Engineering 2008 (ICCCE08). Publication Year: 2008, pp. 1052–1056

[Ju00] Juric, Radmila: Extreme Programming and its Development Practices. In: Proceedings of the 22nd Int. Conf. lnformation Technology lnterfaces (ITI 2000). Pubilcation Year: 2000, pp. 97–104

[Ke13] Kelly, D.; Smyth, B.; Caulfield, B.: Uncovering Measurements of Social and Demographic Behavior From Smartphone Location Data. In: IEEE Transactions on Human-Machine Systems. Vol. 43, Issue 2. Publication Year: 2013; pp. 188–198

[Ki11] Kiss, G.: Using Smartphones in Healthcare and to Save Lives. In: Internet of Things (iThings/CPSCom). 2011 IEEE International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing. Publication Year: 2011, pp. 614–619

[La10] Lau, Sian Lun; David, K.: Movement Recognition using the Accelerometer in Smartphones. In: Conference: Future Network and Mobile Summit. Publication Year: 2010, pp. 1–9

[Lu04] Luo, S., Hu, Q.: A Dynamic Motion Pattern Analysis Approach to Fall Detection. In: 2004 IEEE International Workshop on Biomedical Circuits & Systems. Publication Year: 2004, pp. 1–5-8a

[Ma07] Martin, Robert C.: Professionalism and Test-Driven Development. In: IEEE Software. Vol. 24, Issue 3. Publication Year: 2007, pp. 32–36

[Mu01] Muller, Matthias M.; Tichy, Walter F.: Case Study: Extreme Programming in a University Environment. In: ICSE 2001. Proceedings of the 23rd International Conference on Software Engineering. Publication Year: 2001, pp. 537–544

[Oj96] Ojala, T.; Pietikäinen, M.; Harwood, D.: A comparative study of texture measures with classification based on feature distributions. In: Pattern Recognition. Vol. 29, Issue 1. Publication Year: 1996, pp. 51–59

[Sa78] Sakoe, Hiroaki; Chiba, Seibi: Dynamic programming algorithm optimization for spoken word recognition. In: IEEE Transactions on Acoustics, Speech and Signal Processing. Vol. 26, Issue 1. Publication Year: 1978, pp. 43–49

[Vi12] Viet, Vo Quang; Lee, Guee-Sang; Choi, Deok-Jai: Fall Detection Based on Movement and Smart Phone Technology. In: 2012 IEEE RIVF International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF). Publication Year: 2012, pp. 1–4

[Xu05] Xu, Shaochun; Rajlich Václav: Pair Programming in Graduate Software Engineering Course Projects. In: Proceedings. FIE ’05 Frontiers in Education. 35th Annual Conference. Pedagogies and Technologies for the Emerging Global Economy. Vol. I. Publication Year: 2005. pp. F1G

[Xu12] Xu, Zhi; Bai, Kun; Zhu, Sencun: TapLogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In: WISEC ’12 Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. Publication Year: 2012, pp. 113–124

11.3 Internetquellen

[An10] Android Developer Website. Thema: Testing Fundamentals. Seitenaufruf: 12.12.2010. http://developer.android.com/guide/topics/testing/testing_android.html

[An12] Android Developer Website. Thema: Sensor Event. Seitenaufruf: 23.11.2012. http://developer.android.com/reference/android/hardware/SensorEvent.html

[Bi10] Billups, Toan: How test first development changed my life. 22.4.2010. Webblog. Seitenaufruf: 26.05.2013. http://toranbillups.com/blog/archive/2010/04/22/How-test-first-development-changed-my-life/

[Ca12] Catroid Website. Seitenaufruf: 22.1.2012. http://www.catroid.org/catroid/index/1

[Ht10] HTC Desire Produktwebsite. Seitenaufruf: 11.5.2010. http://www.htc.com/de/product/desire/specification.html

[Li12] Limburg, Arne; Röwekamp, Lars: Was man über Unit Tests in Android wissen sollte. 26.7.2012. Seitenaufruf: 13.6.2013. http://jaxenter.de/artikel/JUnit-fuer-Android

[Pi13] Robolectric Release Notes. Seitenaufruf: 29.3.2013. http://pivotal.github.com/robolectric/release-notes.html

[Ro13] Robotium Google Code Website. Seitenaufruf: 13.6.2013. http://code.google.com/p/robotium/

[Ru01] Rumpe, Bernhard: Extreme Programming – Überblick, Hintergründe, Perspektiven. Software & Systems Engineering. Technische Universität München. 2001. Seitenaufruf: 13.6.2013. http://www4.in.tum.de/misc/perlen/perlen-folien/rumpe-xp-folien.pdf

[Sc13] Scratch Website. Seitenaufruf 7.4.2013. http://scratch.mit.edu/

[Se13] SensorSimulator Google Code Website. Seitenaufruf: 29.3.2013. http://code.google.com/p/openintents/wiki/SensorSimulator