Fachhochschule Lübeck

Fachbereich Elektrotechnik und Informatik

BACHELORARBEIT

Systematisches Testen von Mobile Applikationen

von

Stefanie Grimm

vorgelegt von: Stefanie Grimm

Erstbetreuer: Prof. Dr. rer. nat. Nane Kratzke

Zweitbetreuer: Prof. Dr. Stefan Krause

Aufgabenbeschreibung

In allen Bereichen, wo Software ihren Einsatz findet, ist die Qualität der Software zu einem entscheidenden Faktor für den Erfolg eines Unternehmens oder deren Produkten geworden. Um Qualität von Softwaresystemen zu erreichen, wird Software geprüft und getestet. Hierzu durchläuft Software Testphasen. Die zugehörigen Tests sind bekannt als Komponententest, Integrationstest, Systemtest und Abnahmetest. Innerhalb der Testphasen gibt es verschiedene Testarten, z.B. Black-Box-Test, White-Box-Test, funktionaler Test oder Codeüberdeckung.

Mit zunehmender Anzahl von Smartphones und Tablets steigt heutzutage auch die Bedeutung mobiler Anwendungen. Die Apps können immer mehr Aufgaben übernehmen und durch den Einsatz auf mobilen Geräten gleichzeitig flexibel eingesetzt werden. Bei Softwaretests von zunehmend komplexer werdenden mobilen Anwendungen kommen zu den Anforderungen der Tests herkömmlicher Anwendungen, weitere hinzu. Es treten bei mobilen Anwendungen weitere Problemfelder auf, wie zum Beispiel die flexible Umgebung, in der sich die Geräte befinden können. So gibt es Situationen, dass das Gerät zu Hause per WLAN ins Netz kommt, unterwegs aber nur per UMTS. Des Weiteren gibt es bspw. Apps, die mit Ortungsdiensten arbeiten. Ortungsdienste wie GPS oder Gallileo sind aber nicht immer und überall verfügbar. Stellenweise werden anstatt Satelliten-gestützter Ortungsdienste WLAN-basierte oder funkmastbasierte Ortungsfunktionen geringerer Güte genutzt. Display Auflösungen, Speicher- und Prozessorausstattung unterscheiden sich zwischen den mobilen Geräteklassen teilweise erheblich.

Im Rahmen dieser Bachelorarbeit soll deshalb ein allgemeiner Überblick über Softwaretestverfahren gegeben werden und anhand einer geeigneten Application für Android dargestellt werden. Hierzu sind verschiedenste Open Source Testframeworks zu recherchieren, die die folgende Matrix abdecken sollen.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Auf Basis dessen erfolgt eine Evaluation der Testframeworks anhand folgender Kriterien:

- Installationskomplexität
- Bekanntheitsgrad
- Komplexität der zu entwickelnden Tests
- Aufwand der Einarbeitung

So soll die Eignung der jeweiligen Frameworks für Testphasen/Testarten erhoben und aufbereitet werden.

Erklärung zur Bachelorarbeit

Ich versichere, dass ich die Arbeit selbstständig und ohne fremde Hilfe verfasst habe. Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.
Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

Ort, Datum	Unterschrift

Danksagung

Ich möchte mich recht herzlich bei Herrn Prof. Dr. rer. nat. Nane Kratzke bedanken, der stets für mich ansprechbar war sowie mit viel Engagement meine Arbeit betreut hat.

Ebenfalls herzlich danken möchte ich Herrn Prof. Dr. Stefan Krause, der sich sofort bereit erklärte meine Arbeit als Zweitkorrektor zu betreuen.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das Bedürfnis an mobilen Geräten, wie Smartphones und Tablets steigt täglich. Aber auch der Gebrauch von Applikationen findet immer mehr Zuspruch und die Funktionen werden immer umfangreicher. Somit wächst auch das Bedürfnis an einer hohen Softwarequalität.

Ist die Applikation erst einmal im AppStore[1], aber es wurde keine ausreichende Qualitätssicherung durchgeführt, kann dieses zu Problemen mit der Applikation führen. So ist es zum Beispiel möglich, dass es zu einem Absturz der Anwendung kommen kann und somit werden auch die Bewertungen für die jeweilige Applikation schlechter. Zudem ist es auch gerade bei kostenpflichtigen Applikationen wichtig, dass diese einwandfrei laufen. Auch nachdem eine Änderung an der Software durchgeführt wurde, sollten die bereits vorher verwendeten Funktion noch lauffähig sein. Da Google die Applikationen nicht prüft, bevor diese in den AppStore kommen, können auch Applikationen veröffentlicht werden, die nicht lauffähig sind. Erst nachdem ein Nutzer ein Problem mit der jeweiligen Applikation gemeldet hat, prüft Google diese. (Stefan, 2013)

1.2 Zielsetzung

In dieser Bachelorarbeit soll, aufgrund der vorher genannten Probleme, genauer auf das Thema Testen im mobilen Bereich eingegangen werden. Der Schwerpunkt liegt hier auf dem Betriebssystem Android, da es am meisten verbreitet ist.[2] Zunächst wird ein allgemeiner Überblick über die Teststufen und die Testarten gegeben. Darauf aufbauend werden verschiedene Testframeworks recherchiert und anhand einer selbstentwickelten Applikation ausprobiert.

Am Ende erfolgt dann eine Evaluation der Ergebnisse. In dieser geht es darum, die Testframeworks anhand verschiedener Kriterien zu bewerten und zu prüfen, ob sich diese für den Testeinsatz bei Applikationen auf Basis von Android eignen.

[1] Bezeichnung für eine digitale Vertriebsplattform von Anwendungssoftware. Der Service ermöglicht es Benutzern Software aus einem Anwendungskatalog von Erst- und Drittanbieterentwicklern zu suchen und herunterzuladen.

[2] Nähere Erläuterung im Kapitel 2.1 Android

2 Grundlagen

2.1 Android

Android ist sowohl ein Betriebssystem als auch eine Software-Plattform für mobile Endgeräte. Entwickelt wird Android von der "Open Handset Alliance", bei der Google das Hauptmitglied ist. Die erste Android Version kam am 21. Oktober 2008 auf den Markt und hatte noch die einfache Bezeichnung "Android 1.0". Erst ab der 2009er-Version begann es, dass das Betriebssystem Namen wie "Android Cupcake", "Android Donut" oder die aktuelle Version "Android Jelly Bean" bekam. Auffallend hierbei ist, dass die Namen an Süßspeisen erinnern und dessen Anfangsbuchstabe dem Alphabet aufsteigend ist. Mit "Android Honeycomb" brachte Google sein erstes System rein für Tablets auf den Markt.

Bedient wird Android mit Hilfe des Touchscreens und vordefinierter Soft- und Hardwaretasten. Die Standardoberfläche besteht meist aus drei, fünf oder sieben Startbildschirmen. Am oberen Rand findet sich die Benachrichtigungsleiste mit der Uhrzeit, Akkustand, WLAN, Internetverbindung und Bluetooth Platz. Hier werden Meldungen von laufenden Programmen, neuen Nachrichten oder Systemmeldungen angezeigt.

Hinter Android befindet sich ein Linux-Kernel 2.6, welcher für die Speicher- und Prozessverwaltung zuständig ist. Weitere wichtige Elemente sind auf der Java-basierenden virtuellen Maschine Dalvik und den dazu gehörigen Android-Java-Klassenbibliotheken zu finden. Die "Dalvik Virtuel Machine"(Dalvik-VM) führt, wie auch die Java-VM, ByteCode aus. Der Unterschied zwischen den beiden besteht darin, dass die Java-VM auf einem Kellerautomaten basiert und die Dalvik-VM auf einer Registermaschine. (wikipedia, 2013)

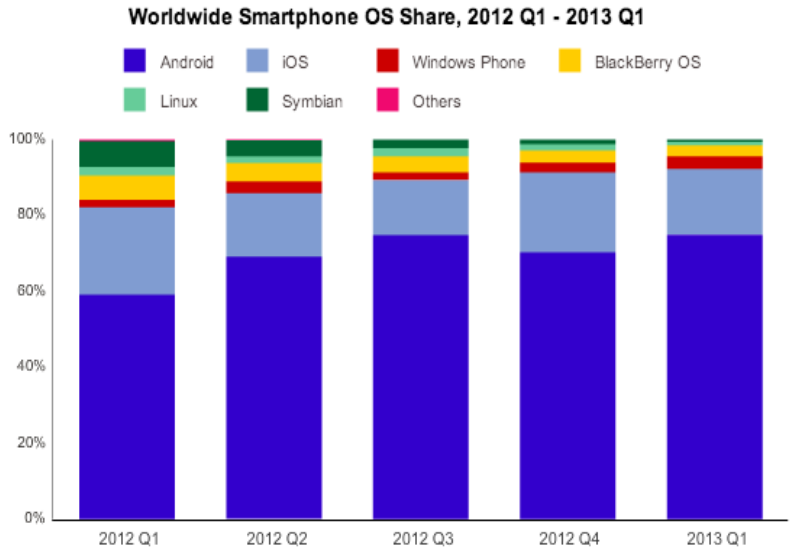


Abbildung 2.1-1 Übersicht der Anteile der OS weltweit (Tiefenhäler, 2013)

In der obigen Abbildung 2.1-1 ist klar zu erkennen, dass Android weltweit den größten Marktanteil aufweisen kann. Kurz dahinter folgen iOS, Windows Phone und Blackberry OS.

2.2 Testen von mobilen Applikationen

Wie bereits in der Aufgabenstellung erwähnt, ist das Testen von mobilen Applikationen besonders problematisch. Die Umgebung in der sich die mobilen Geräte befinden ist sehr komplex, da diese überall ihren Einsatz finden können. Daher müssten beim Testen möglichst alle Begebenheiten der Umgebung berücksichtigt werden, damit man sicher sein kann, dass die Anwendung immer und überall funktionsfähig ist.

Hinzu kommt, dass es eine große Anzahl verschiedener Android Geräte gibt, welche sich in erster Linie in Leistung, Displaygröße und Android Version unterscheiden. Diese Versionen sind vom Hersteller und gegebenenfalls vom Provider an das jeweilige Gerät angepasst. Abgesehen davon gibt es für mobile Geräte meist Updates auf aktuellere Versionen, welche allerdings nicht immer von den Nutzern installiert werden. Des Weiteren existieren viele sogenannter Custom Roms[3]. Um alles zu erfassen müsste man also alle möglichen Konfigurationen auf allen möglichen Geräten testen. Zu alledem sind auch die Hardware-Ressourcen auf mobilen Geräten sehr beschränkt obwohl diese immer besser werden. So gibt es kaum Möglichkeiten für Ausgabe und Protokollierung, wodurch die Auswertung erschwert wird.

[3] Custom Rom: Ersetzt Standardversion des Betriebssystems, um Funktionen zu erweitern oder die Herstellerbeschränkungen zu umgehen.

3 Testarten im Softwarelebenszyklus

Im Folgenden werden die Testarten beschrieben, welche im Laufe eines Softwarelebenszyklus eine wichtige Rolle spielen können.

3.1 Black-Box Test

Bei dieser Testart werden die Testfälle anhand der vorher festgelegten Spezifikation ermittelt. Allerdings bleibt der Sourcecode hier unbeachtet. Das heißt, es werden nur die fertigen Funktionen getestet, ohne den Sourcecode im Hintergrund in Betracht zu ziehen. Eine Auswertung der Testüberdeckung erfolgt mit Hilfe des im Voraus festgelegten Ein-/ Ausgabeverhaltens der Funktionen. Ein Beispiel für eine Black-Box Test ist das funktionsorientierte Testen, welches im Kapitel 3.3 weiter ausgeführt wird.

3.2 White-Box Test

Bei den White-Box Tests werden die Testfälle anhand des Sourcecode ermittelt. Das heißt, hier wird nur mit der inneren Struktur getestet und die Spezifikation außer Acht gelassen. White-Box Tests eignen sich dann dafür, um Fehler in Teilkomponenten aufzudecken. Die Auswertung der Testüberdeckung erfolgt anhand des Sourcecode. Zu den White-Box Tests gehört unter anderem das im Kapitel 3.4 beschriebene kontrollflussorientierte Testen.

3.3 Funktionsorientiertes Testen

Funktionsorientierte Testarten haben ihren Namen daher erhalten, weil sie gegen die Spezifikationen, also die Soll-Funktionen, einer Software testen. Anwendung findet diese Testart in allen Teststufen. Beim Modultest wird gegen die Modulspezifikation getestet, beim Integrationstest gegen die Schnittstellenspezifikation und beim Systemtest gegen die Anforderungsdefinition.

Mit Hilfe der vorher definierten Spezifikationen der Software ist es dem Tester also möglich passende Testfälle zu entwickelt. Er liest sich vorher die Spezifikation durch und kann anhand dessen beurteilen, wann die Software eine korrekte Reaktion auf den Testfall zeigt. Abgeschlossen ist ein Test dann, wenn der Inhalt der Spezifikation durch die Testfälle abgedeckt wurde.
Da funktionsorientierte Testarten also nur gegen die Spezifikationen testen und somit die Beachtung der Programmstruktur außer Acht lassen, gehören sie zu den Black-Box- Tests.

(Liggesmeyer, 2009)

3.3.1 Äquivalenzklassenbildung

In der Praxis ist es für den Tester oft schwierig aus einer großen Anzahl von Betriebssituationen der Software die richtigen Testfälle rauszusuchen. Meist bleiben viele

der Situationen ungetestet. Zudem kommt, dass die Auswahl der Testfälle sehr genau sein muss. Diese sollen am besten als Stellvertreter der kompletten Funktionalität dienen, es sollen Fehler möglichst zuverlässig erkannt werden und sie sollen frei von Überschneidung sein. Um das alles zu gewährleisten gibt es die sogenannte funktionale Äquivalenzklassenbildung. Hierbei werden anhand gleicher Ein- und Ausgabedaten von Klassen oder Objekten sogenannte Äquivalenzklassen gebildet. Die Bildung der Testfälle läuft nach folgendem Schema ab:

- Analyse und Spezifikation der Eingabedaten, der Ausgabedaten und der Bedingungen gemäß den Spezifikationen der Software
- Bildung der Äquivalenzklassen durch Klassifizierung der Wertebereiche für Ein- und Ausgabedaten
- Bestimmung der Testfälle durch Werteauswahl für jede Äquivalenzklasse

Die erstellten Testfälle gelten somit für alle Objekte der Äquivalenzklasse, so dass nicht für jedes Objekt ein eigener Testfall geschrieben werden muss.

Es wird noch zwischen gültigen und ungültigen Äquivalenzklassen unterschieden. Die gültigen Äquivalenzklassen enthalten die positiven Eingabewerte, die ungültigen Äquivalenzklassen die negativen Eingabewerte, also solche, die nicht auftreten dürfen. Des Weiteren gibt es noch die Grenzwertanalyse. Sie arbeitet mit den Rand- bzw. Grenzwerten einer Äquivalenzklasse. (Liggesmeyer, 2009) (wikipedia, 2013)

3.3.2 Zustandsbasierter Test

Hierbei werden die Spezifikationen als Zustandsautomat definiert, welcher keine Fehlerzustände enthält. Diese müssen extra abgebildet werden, indem zu jeder Ausgangssituation und deren Ergebnis der Folgezustand und die ausgelösten Aktionen angegeben werden. Die Kombination aus Ausgangssituation und deren Ergebnis wird dann getestet. Die Vollständigkeit der ermittelten Testfälle ist gegeben, wenn folgende Kriterien erfüllt sind:

- Werden alle Zustandsübergänge durchlaufen?
- Werden alle Ereignisse, die Zustandsübergänge hervorrufen sollen, getestet?
- Werden alle Ereignisse, die keine Zustandsübergänge hervorrufen dürfen, getestet?

(Liggesmeyer, 2009)

3.3.3 Ursache-Wirkungs-Analyse

Es werden Kombinationen von zulässigen Äquivalenzklassen, welche unterschiedliche Programmreaktionen zur Folge haben, auf ihr korrektes Zusammenspiel getestet.

Bei dieser Methode werden Ursachen und Wirkungen jeder Teilspezifikation identifiziert. Eine Ursache ist dabei eine Situation oder Bedingung, die eine Entscheidung beeinflussen. Also die zulässigen Äquivalenzklassen. Eine Wirkung ist eine mögliche Ausgabe, Programmreaktion oder Systemzustand. Die Teilspezifikationen werden in einen Booleschen Graphen überführt, welcher die Ursache und die Wirkung logisch miteinander verbindet. Im Anschluss daran werden Abhängigkeiten eingefügt, welche durch Umgebungsbedingungen entstehen. Aus dem entstanden Graphen wird dann eine Entscheidungstabelle erstellt, wobei aus jeder Spalte ein Testfall entsteht. (Liggesmeyer, 2009)

3.4 Kontrollflussorientiertes Testen

Kontrollflussorientierte Testarten orientieren sich an dem Kontrollflussgraphen einer Software. Da sie die Vollständigkeit des Tests mit Hilfe der Abdeckung des Quellcodes beurteilen, werden sie auch strukturorientierte Testarten genannt. Die wichtigste Anwendungsart ist der Modultest. Im Integrationstest spielen sie nur eine kleine Rolle und im Systemtest finden sie keine Anwendung mehr.

Kontrollflussorientierte Tests betrachten die Struktur des Codes und gehören somit zu den White-Box-Tests. Im Einzelnen werden die Anweisungen, Zweige, Bedingungen, Schleifen und Pfade einer Software beachtet. Grundlage zum Testen bildet der Kontrollflussgraph. Dieser kann für jedes Programm erstellt werden. Frameworks zur Unterstützung kontrollflussorientierter Testarten erzeugen solche Graphen und nutzen sie zur Darstellung der Testergebnisse. (Liggesmeyer, 2009)

In der folgenden Abbildung ist ein Ausschnitt aus einem Sourcecode und der dazugehörige Kontrollflussgraph dargestellt.

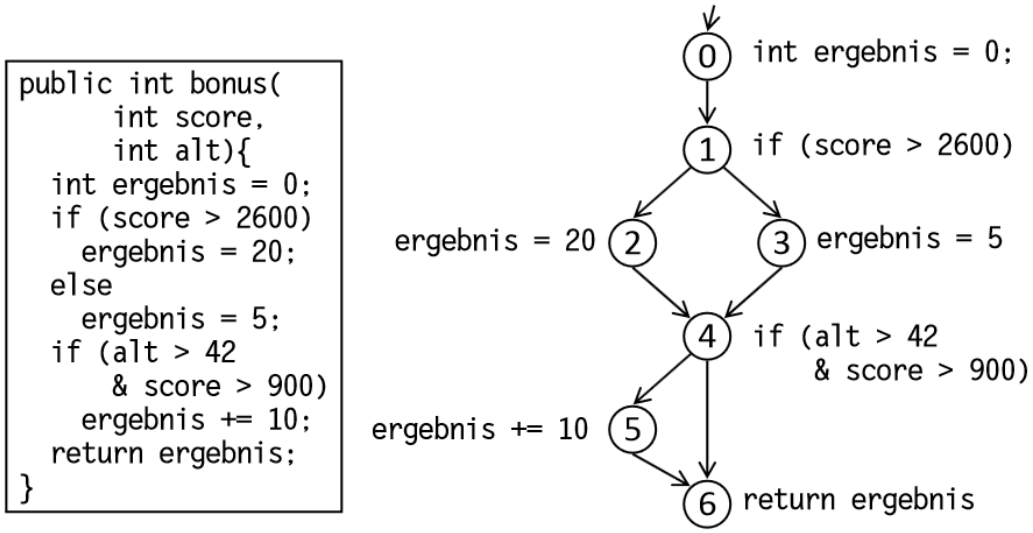


Abbildung 3.4-1 Beispiel Sourcecode und Kontrollflussgraph (FH Osnabrück)

3.4.1 Anweisungsüberdeckungstest (Statement Coverage)

Der Anweisungsüberdeckungstest wird auch als C_0-Test bezeichnet. Das Ziel des Tests ist es, dass alle Anweisungen des zu testenden Programms mindestens einmal ausgeführt werden. Das heißt, dass alle Anweisungen (Knoten) des Kontrollflussgraphen durchlaufen werden. Somit wird sichergestellt, dass sich kein "toter Code", also Anweisungen die niemals durchlaufen werden, im Programm befindet. Als Testergebnis wird der sogenannte Anweisungsüberdeckungsgrad bestimmt:

$$C_{Anweisung} = \frac{\text{Anzahl der ausgeführten Anweisungen}}{\text{Anzahl der Anweisungen}}$$

Eine vollständige Anweisungsüberdeckung wird erreicht, wenn alle eingegebenen Testdaten mindestens einmal ausgeführt wurden. (Liggesmeyer, 2009)

Angenommen es soll für den Sourcecode im Kapitel 3.4 in der Abbildung 3.4-1 ein Anweisungsüberdeckungstest durchgeführt werden.

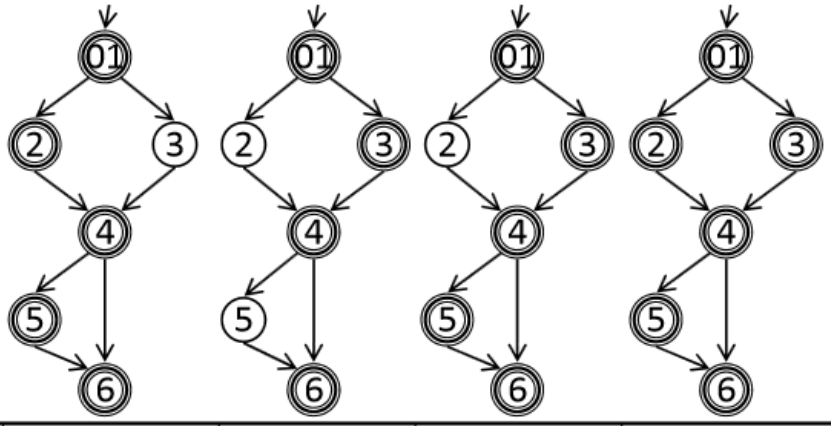
Test	T1	T2	T3	T1+T3
score	2601	900	2600	
alt	43	43	88	
ergebnis	30	5	15	
				
C0	4/6	4/6	5/6	6/6

Abbildung 3.4-2 Anweisungsüberdeckung berechnen (FH Osnabrück)

In der Abbildung 3.4-2 ist zu sehen, dass verschiedene Testfälle konstruiert werden und dann geprüft wird, wie viele Anweisungen, also Knoten, im Sourcecode durchlaufen werden. Beim ersten Testfall werden 4 von 6 durchlaufen, so auch beim zweiten Testfall und beim dritten dann 5 von 6 Knoten. Erst im letzten Testfall, dem Zusammenschluss vom ersten und dritten Testfall, ist es gelungen, das Ziel einer Anweisungsüberdeckung zu erreichen, nämlich das Durchlaufen aller Anweisungen. *Cobertura* ist ein Framework, welches diesen Mechanismus nutzt. Eine nähere Erläuterung zu diesem Tool ist im Kapitel 8.4 zu finden.

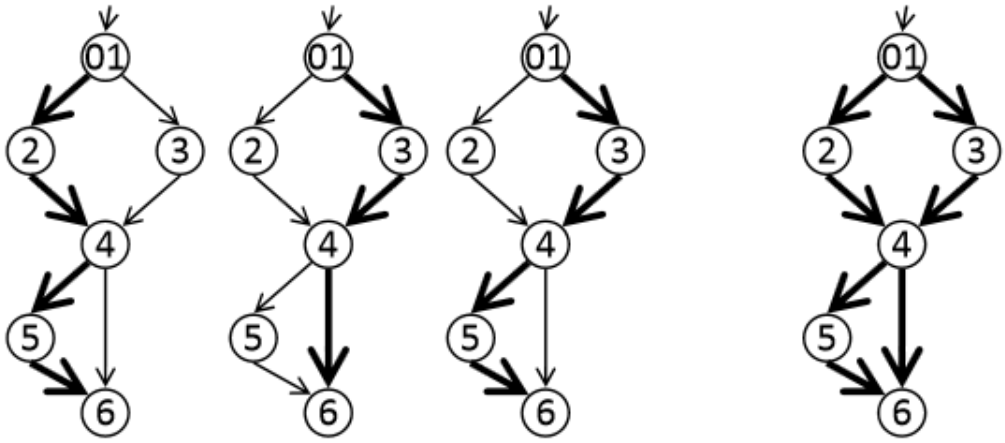
3.4.2 Zweigüberdeckungstest (Branch Coverage)

Der Zweigüberdeckungstest wird auch als C_1-Test bezeichnet. Diese Testart umfasst den Anweisungsüberdeckungstest vollständig, es müssen aber strengere Kriterien erfüllt werden. Im Gegensatz zum Anweisungsüberdeckungstest werden hier nicht die Knoten durchlaufen, sondern es sollen alle Kanten (Zweige) ausgeführt werden. Somit wird hier also sichergestellt, dass sich im Programm keine Zweige befinden, die niemals durchlaufen werden. Als Testergebnis wird der Zweigüberdeckungsgrad definiert: (Liggesmeyer, 2009)

$$C_{Zweig} = \frac{\text{Anzahl der ausgeführten Zweige}}{\text{Anzahl der Zweige}}$$

Auch hier wieder ein Beispiel anhand des Sourcecode aus Kapitel 3.4 in der Abbildung 3.4-1. Wie schon bei dem Anweisungsüberdeckungstest werden verschiedene Testfälle konstruiert, nur das dieses mal nicht alle Knoten durchlaufen werden sollen, sondern alle Kanten des Kontrollflussgraphen.

Test	T1	T2	T3	T1+T3	T1+T2
score	2601	900	2600		
alt	43	43	88		
ergebnis	30	5	15		



C0	4/7	3/7	4/7	6/7	7/7

Abbildung 3.4-3 Zweigüberdeckung berechnen (FH Osnabrück)

In der Abbildung 3.4-3 ist zu erkennen, dass beim ersten Testfall 4 von 7 Kanten, beim zweiten Testfall 3 von 7 Kanten und beim dritten Testfall wieder 4 von 7 Kanten durchlaufen werden. Erst wenn der erste und dritte beziehungsweise der erste und zweite Testfall zusammengeführt werden, ist es möglich, alle Kanten des Graphen zu durchlaufen.

3.4.3 Einfacher Bedingungsüberdeckungstest (Condition Coverage)

Bei dieser Testart wird jede atomare (a,b) Bedingung der Gesamtbedingung mindestens einmal mit true und einmal mit false getestet. Dadurch wird allerdings nicht garantiert, dass

auch die Gesamtbedingung (a && b) einmal true und false wird. Daher ist der C_2-Test ein eher schlechter Test. (Liggesmeyer, 2009)

Auch hier wird anhand des Sourcecode aus Kapitel 3.4 in der Abbildung 3.4-1 ein Beispiel eines einfachen Bedingungsüberdeckungstest gezeigt. Dieser Test läuft im Grunde genauso wie ein Anweisungsüberdeckungs- oder Zweigüberdeckungstest ab. Es werden verschiedene Testfälle konstruiert und geprüft, so dass alle atomaren Bedingungen mindestens einmal mit false und einmal mit true getestet wurden.

Test	T1	T2	T3	T1+T2	T4	T2+T4
score	2601	900	2600		2601	
alt	43	43	88		42	
ergebnis	30	5	15		20	
score > 2600	t	f	f	t f	t	f t
alt > 42	t	t	t	t	f	t f
score > 900	t	f	t	t f	t	f t
C2	3/6	3/6	3/6	5/6	3/6	6/6

Abbildung 3.4-4 Einfache Bedingungsüberdeckung berechnen (FH Osnabrück)

Allerdings wird bei dem einfachen Bedingungsüberdeckungstest, wie in Abbildung 3.4-4 zu sehen, noch festgelegt, wann die einzelnen Variablen einen Wert mit true oder mit false belegen. Hier wird deutlich, dass eine Belegung der Variablen mit gleichzeitig true als auch false erst möglich wird, wenn zwei Testfälle kombiniert werden. Einer der Testfälle muss dann den Wert true aufweisen und der andere den Wert als false.

3.4.4 Mehrfacher Bedingungsüberdeckungstest (Multiple Condition Coverage)

In dem Fall dieser Testart werden alle atomaren Bedingungen einer Bedingung betrachtet. Das heißt, wenn n atomare Bedingungen in einer Bedingungen sind, dann werden 2^n Kombinationen gebildet. (Liggesmeyer, 2009)

Auch hier wieder ein Beispiel anhand des Sourcecode aus Kapitel 3.4 in der Abbildung 3.4-1. Der mehrfache Bedingungsüberdeckungstest funktioniert genauso wie der einfache Bedingungsüberdeckungstest. Es werden verschiedene Testfälle konstruiert und zusätzlich wird festgelegt, wann die einzelnen Variablen den Wert mit true oder false belegen. Einziger Unterschied zwischen den beiden Bedingungsüberdeckungstests ist, dass beim mehrfachen Test nicht nur die einzelnen Variablen geprüft werden, sondern auch die Verknüpfungen.

Test	T1	T2	T3	T4	T2+T4	T1+T2+T4
score	2601	900	2600	2601		
alt	43	43	88	42		
ergebnis	30	5	15	20		
score > 2600	t	f	f	t	f t	t f
alt > 42	t	t	t	f	t f	t f
score > 900	t	f	t	t	f t	t f
alt > 42 && score >900	t	f	t	f	f	t f
C3	4/8	4/8	4/8	4/8	7/8	8/8

Abbildung 3.4-5 Mehrfache Bedingungsüberdeckung berechnen

In der Abbildung 3.4-5 wird diese Verknüpfung deutlich. Es soll nicht nur die Variable *alt > 42* sein, sondern auch noch die Variable *score > 900*. Bei der Auswertung der Testfälle ist dann zu sehen, dass erst das Zusammenlegen dreier Testfälle alle möglichen Belegungen zum einen true und zum anderen false werden lässt.

3.4.5 Pfadüberdeckungstest (Path Coverage)

Bei dieser Testart werden alle möglichen Pfade im Kontrollflussgraphen vom Startknoten bis zum Endknoten betrachtet. Für die meisten Software-Module ist dieser Test daher nicht ausführbar, da sie eine unendliche hohe Anzahl an Pfaden besitzen können. Diese hohe Anzahl wird meist durch Schleifen verursacht, welche keine feste Wiederholungszahl haben. (Liggesmeyer, 2009)

3.5 Softwaremessung

Hinter dieser Testart versteckt sich das Bestreben die Software zahlenmäßig besser zu verstehen. Messwerte können in diesem Fall Produkteigenschaften ausdrücken oder zur Kontrolle und Steuerung des Entwicklungsprozesses einer Software genutzt werden, aber auch bei der Definition von prüfbaren Zielen können sie von Vorteil sein. Häufig liefern diese Messwerte allerdings erst im direkten Vergleich mit anderen bekannten Softwareprodukten verwertbare Aussagen. Einige verbreitete Einsatzgebiete für Maße sind: (Liggesmeyer, 2009)

- Kontrolle der Qualität
- Kontrolle der Komplexität

- Aufwands-, Kosten- und Zeitabschätzung
- Vergleich und Beurteilung von Produkten

In der Softwareentwicklung spricht man hierbei häufig von Metriken.
Eine Softwarequalitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit. (IEEE Standard 1061)

Es gibt sehr viele Softwaremetriken. Den Überblick zu behalten ist sehr schwer, deshalb wurde die Metriken in verschiedene Kategorein eingeteilt.

Die Produktmaße beziehen sich auf Eigenschaften des Softwareproduktes. Möchte man beispielsweise die Zuverlässigkeit oder die Wartbarkeit messen, kann die Software hier der Entwurf der grafischen Oberfläche sein oder ein Spezifikationsdokument.

Bei den Prozessmaßen wird der Erstellungsprozess der Softwareeinheit als Messgegenstand verwendet. Ein Prozess ist schwerer zu messen als ein Produkt. In den meisten Fällen wird deshalb dem Prozess eine Zahl zugeordnet. Je höher die Zahl ist, desto reifer ist der Prozess.

Das Projektmaß beschreibt, wie sich das einzelne Projekt im Vergleich zu den Vorgaben entwickelt. Aufwands- und Kostenmaße sind hier typische Metriken.

Ein Beispiel für ein Framework, welches diese Mechanismen beschreibt, ist *metrics*, das in der Entwicklungsumgebung *eclipse* genutzt werden kann. Eine nähere Beschreibung dieses Tools erfolgt in Kapitel 6.5.

3.6 Codeanalyse

Die Tätigkeit der Codeanalyse hat starke Ähnlichkeit mit der eines Compilers. Das heißt, wie auch bei den Compilern, wird zunächst eine lexikalische Analyse des Quellcodes durchgeführt. Der Quelltext wird dann einer Reihe formaler Prüfungen unterzogen, bei denen bestimmte Sorten von Fehlern entdeckt werden können.

Die Stilanalyse gehört zur Codeanalyse. In den meisten Fällen wird Software mit Hilfe sogenannter Programmierkonventionen entwickelt, welche häufig bestimmte Qualitätseigenschaften unterstützen sollen. Diese können von der Firma selber aufgestellt werden, wie zum Beispiel, dass eine Klasse im Programm nur eine bestimmte Anzahl von Codezeilen haben darf. Es existieren aber auch Programmierkonventionen die von der Programmiersprache selber ausgehen. Es wird aber auch noch zwischen semantischen und syntaktischen Konventionen unterschieden.

Bei semantischen Konventionen handelt es sich zum Beispiel um eine aussagekräftige Namensgebung bei Variablen oder um geeignete Kommentare. Die erforderliche Bewertung ist nur durch den Menschen möglich, das kann noch kein Werkzeug leisten.

Bei den syntaktischen Konventionen hingegen ist es möglich diese mit Hilfe von Werkzeugen zu prüfen. Hierbei handelt es sich zum Beispiel um das Prüfen, ob die schließende Klammer eines Blocks in einer eigenen Zeile steht. Alle diese Konventionen werden mit Hilfe der Stilanalyse auf ihre Einhaltung hin geprüft. (Liggesmeyer, 2009)

Ein Beispiel für ein Framework, welches die oben beschriebenen Mechanismen einsetzt, ist *checkstyle*, das in der Entwicklungsumgebung *eclipse* verwendet werden kann. Eine nähere Erläuterung erfolgt in Kapitel 7.9.

4 Teststufen im Softwarelebenszyklus

Im Folgenden werden die einzelnen Teststufen beschrieben, welche bei der Entwicklung von Software von Bedeutung sind. Diese sind aus dem Vorgehensmodell "V-Modell" abgeleitet, welches aus Sicht des Testens eine wichtige Rolle spielt. Im linken absteigenden Ast des Modells werden die Phasen der Spezifikation der Software abgebildet. In dem rechten aufsteigenden Ast stehen den jeweiligen Spezifikationsphasen die einzelnen Teststufen gegenüber. Diese Art der Darstellung soll zu einer hohen Testabdeckung führen, da die Spezifikationen die Grundlage für die einzelnen Teststufen bilden. (Andreas Spillner, 2012)

4.1 Modultest

Der Modultest ist auch als Komponententest oder Unit Test bekannt. Es ist die erste Teststufe dem die erstellten Softwarebausteine unterzogen werden. Der Begriff Modul oder auch Komponente bezeichnet dabei die kleinste sinnvoll unabhängig testbare Programmeinheit. Ein Modul ist dabei in der objektorientierten Entwicklung meist eine Klasse. Jedes Modul wird unabhängig von den anderen Modulen getestet. Das hat den Vorteil, dass ein entdeckter Fehler direkt den entsprechenden Modulen zugeordnet werden kann. Somit können bei einem Modultest keine Fehler gefunden werden, welche sich aus der Interaktion mehrerer Module ergeben. Da Module im Allgemeinen nicht selbständig lauffähig sind, sondern von übergeordneten Programmeinheiten aufgerufen werden, werden sogenannte Treiber (Driver) erstellt, die den übergeordneten Aufruf der zu testenden Einheit simulieren. Dienste, die dem betreffenden Modul untergeordnet sind, werden durch sogenannte Dummies ersetzt. (Liggesmeyer, 2009)

Ziel des Modultests ist es unter anderem sicherzustellen, dass das jeweilige Testobjekt die gewünschte Funktionalität korrekt und vollständig realisiert. Des Weiteren ist es wichtig , die Objekte auf Robustheit[4] zu überprüfen. Hier werden als Testeingaben Methodenaufrufe, Daten und Sonderfälle verwendet, welche eigentlich nicht vorgesehen sind oder sogar unzulässig sind. Hierbei wird beispielsweise eine Ausnahmebehandlung erwartet, welche einen Programmabsturz verhindern soll. Ein weiteres Testziel in dieser Phase ist die Effizienz der Komponente. Das heißt, wie hoch ist der Verbrauch des Speicherplatzes und die benötigte Rechenzeit. Ein letztes Ziel stellt die Wartbarkeit dar. Dabei stehen Aspekte wie Codestruktur, Verständlichkeit der Dokumentation oder auch die Kommentierung des Codes im Vordergrund. (Andreas Spillner, 2012)

Typische im Modultest verwendete Testarten:

- Funktionsorientierte Tests
- Kontrollflussorientierte Tests
- Regressionstests

[4] Robustheit ist die Fähigkeit von Software, auch unter außergewöhnlichen Bedingungen zu funktionieren.

4.2 Integrationstest

Als zweite Teststufe nach dem Modultest folgt der Integrationstest. Hier wird vorausgesetzt, dass die einzelnen Komponenten bereits getestet wurden und eventuelle Fehler korrigiert wurden. Diese Komponenten werden nun zu einem Teilsystem integriert. Daher auch der Name dieser Teststufe. Nun muss also getestet werden, ob die Komponenten richtig miteinander interagieren. Es sollen also Fehler in den Schnittstellen und im Zusammenspiel der Komponenten miteinander gefunden werden. Wie schon beim Modultest müssen unterlagerte Module durch Dummies und fehlende aufrufende Module durch Treiber ersetzt werden. Hier können meist die Dummies bzw. Treiber der Modultests verwendet werden. (Liggesmeyer, 2009)

Beim Integrationstest können verschiedene Integrationsstrategien verwendet werden.
Bei der Top-Down Strategie wird mit den Modulen auf oberster Ebene begonnen und die untergeordneten Einheiten werden nach und nach integriert. Noch nicht integrierte Subroutinen werden erneut durch Dummies simuliert, wobei es häufig möglich ist, die im Modultest eingesetzten Dummies wiederzuverwenden.
Bei der Bottom-Up Strategie, wird auf unterster Programmebene begonnen und die übergeordneten Routinen werden eingefügt. In diesem Fall sind wieder Treiber einzusetzen, die ebenfalls aus den Modultests übernommen werden können.
Outside-in stellt eine Mischform der Top-Down und Bottom-Up Strategien dar, um deren Nachteile einzuschränken. (Andreas Spillner, 2012)

Ziele des Integrationstests wurden oben bereits erwähnt: Aufdecken von Schnittstellenfehlern. Es können aber weitere Fehler beim Datenaustausch zwischen den Komponenten auftreten. Diese können sein, dass eine Komponente falsche Daten übermittelt so dass eine andere nicht damit arbeiten kann, oder dass die Daten zu einem falschen Zeitpunkt übergeben werden. Auch diese Fehlerquellen sollen mit Hilfe des Integrationstests aufgedeckt werden. (Andreas Spillner, 2012)

Typische im Integrationstest verwendete Testarten:

- Funktionsorientierte Tests
- Kontrollflussorientierte Tests
- Regressionstest

4.3 Systemtest

Ist der Integrationstest erfolgreich abgeschlossen, folgt als dritte Teststufe der Systemtest. Hier wird nun geprüft, ob das gesamte integrierte System den Anforderungen entspricht. Dies geschieht am besten in einer Testumgebung, welche der späteren Produktumgebung sehr nahe kommt. Statt der bisher verwendeten Treiber und Dummies sollen nun die später tatsächlich verwendeten Hard- und Softwareprodukte in der Testumgebung installiert sein.

In dieser Teststufe werden meist die folgenden Untertests des Systemtests unterschieden.

4.3.1 Funktionstest

Der Funktionstest überprüft, ob alle im Voraus festgelegten Funktionen der Software erfüllt sind und auch wie vorgesehen implementiert wurden. Hierzu dient meist die Anforderungsdefinition in Form eines Lasten- oder Pflichtenheftes, aus der mit Hilfe von funktionsorientierten Testtechniken die Testfälle erarbeitet werden.

4.3.2 Leistungstest

Hier wird das Softwaresystem an Grenzbereiche gebracht, jedoch nicht darüber hinaus. Dabei müssen die entsprechenden Lasten erzeugt werden und gleichzeitig Zeiten und Auslastungen gemessen werden. Als Grundlage dient wieder die Anforderungsdefinition, in welche zum Beispiel die verarbeitbare Menge an Sensoren als auch das Antwortzeitverhalten beinhaltet.

4.3.3 Stresstest

Beim Stresstest wird das System überlastet, zum Beispiel durch entfernen von Ressourcen. Ziele des Tests sind die Klärung des Leistungsverhalten bei Überlast, wie es sich nach Rückgang der Überlast verhält und ob die Ressourcen auch wieder freigegeben werden.

4.3.4 Regressionstest

Der Regressionstest ist der erneute Test einer bereits getesteten Software. Er soll sicherstellen, dass durch eventuelle Änderungen am System keine neuen Fehler aufgetaucht sind. Aus diesem Grunde muss er auch wiederholbar sein. Regressionstests sind nicht nur beim Systemtest sinnvoll, sondern bei allen Teststufen.

Ziele des Systemtests bestehen darin, herauszufinden, ob und wie gut das System den Anforderungsdefinitionen genügt.

Typische im Systemtest verwendete Testarten:

- Black-Box Tests
- Funktionsorientierte Tests

4.4 Abnahmetest

Diese Teststufe schließt sich an den Systemtest an und ist damit die vierte und letzte Teststufe. Dieser Test findet statt, wenn die Entwickler die Software freigeben. Hier liegt es nun beim Auftraggeber die Software zu testen, um zu prüfen, ob diese auch wirklich seinen Anforderungen genügt. Meist wird auch hier die Anforderungsdefinition als Grundlage gewählt. Es gibt auch die Möglichkeit, dass der Auftraggeber die Software an die Endnutzer weitergibt, um zu überprüfen, ob auch diese mit der fertigen Software einverstanden sind.

5 Entwicklung der zu testenden Applikation

5.1 Anforderungen

5.1.1 Anwendungseinsatz

Die Applikation *ImkerApp* ist sowohl für Hobby- als auch für Berufsimker einsetzbar. Zur Nutzung ist ein mobiles Endgerät mit dem Betriebssystem Android Voraussetzung. Diese Applikation bietet dem Nutzer maximale Flexibilität und die Unabhängigkeit von herkömmlichen Computern im Hinblick auf die Nutzung ihres Verwaltungssystems für die einzelnen Bienenstöcke, auch Beuten genannt. In dem Verwaltungssystem werden alle wichtigen Daten, welche zu einer Beute gehören, festgehalten. Hierzu gehören zum Beispiel die Daten, welche die Königin betreffen. Diese sind das Geburtsjahr, das individuelle Zeichen, der Züchter und das Datum, an dem die Königin in die Beute gesetzt wurde. Zu der einzelnen Beute werden außerdem Daten zum Standort, zur Honigleistung, Futterzugabe und andere gespeichert. Dank der Applikation ist es dem Imker möglich, die Daten zu einer Beute direkt am Arbeitsort festzuhalten und muss diese nicht erst zu Hause in den Computer einpflegen. Somit wird ein Arbeitsschritt gespart. Zur Verwendung der Applikation ist keine Internetverbindung nötig.

ImkerApp wird in dieser Bachelorarbeit, wie schon in der Aufgabenstellung erläutert, dafür genutzt, um die Eignung verschiedener Testframeworks für Android Applikationen zu testen. Dafür werden Frameworks recherchiert, welche die Matrix in Abbildung 5.1-1 abdecken. Diese Frameworks werden dann auf *ImkerApp* angewendet und anhand verschiedener Kriterien geprüft, inwieweit diese für den Einsatz unter Android geeignet sind.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 5-1 Matrix zur Abdeckung der Testframeworks

5.1.2 Anwendungsfunktionen

Im Folgenden werden die einzelnen Funktionen der Applikation, welche dem Nutzer zur Verfügung stehen, kurz erläutert:

1. Anlegen einer neuen Beute
2. Beuten anhand des Standortes suchen
3. Beuten anhand der Beutennummer suchen
4. Daten der Beute ansehen/ ändern
5. Historie der Beute ansehen

Die Funktion "Anlegen einer neuen Beute" erlaubt dem Nutzer eine neue Beute im System anzulegen. Er gibt dafür die Nummer der Beute, den Standort und die Informationen zur Königin der jeweiligen Beute an. Die Daten werden dann in der Datenbank gespeichert. Falls der eingegebene Standort noch nicht existiert wird er intern neu angelegt.

Die Funktion "Beute anhand des Standortes suchen" ermöglicht es dem Nutzer einen Standort einzugeben und er erhält dann alle Beuten, welche sich an diesem Standort befinden, in einer Tabelle ausgegeben. Klickt der Nutzer nun auf eine der Beuten gelangt er zum Menü der jeweiligen Beute.

Mit der Funktion "Beuten anhand der Beutennummer suchen" kann der Nutzer eine gewünschte Beutennummer eingeben und gelangt dann in das Menü der jeweiligen Beute.

Hinter der Funktion "Daten der Beute ansehen/ ändern" verbirgt sich das Menü der einzelnen Beuten. Hier hat der Nutzer die Möglichkeit sich den Status, die Versorgung, die Königin, den Stockbau, die Brut oder Anmerkungen zur Beute anzusehen. Die Auswahlmöglichkeiten sollen als Button implementiert werden. Betätigt der Nutzer einen der Button wird er zu den jeweiligen Informationen weitergeleitet und kann diese dann auch ändern und speichern. Die gespeicherten Daten werden dann in einer Historie dargestellt, welche sich hinter der Funktion "Historie der Beute ansehen" verbirgt. Hier kann sich der Nutzer alle bisher eingegebenen Daten anhand einer Tabelle ansehen.

5.1.3 Abgrenzungskriterien

Da diese Applikation als Hauptzweck zum Testen des implementierten Codes gedacht ist, wurde darauf verzichtet nach den vorgegeben Styleguides von Android zu entwickeln. Des Weiteren wurde darauf verzichtet, die Applikation bis zum letzen Detail auszureifen. So ist es nur möglich, Daten zu speichern, wenn alle Felder einen Wert haben, auch wenn sich in einigen Feldern der Wert nicht geändert hat. Zudem ist es auf dem Startbildschirm möglich eine Beutennummer und einen Standort einzugeben. Es wird zwar nur die Beutennummer gewählt, allerdings wäre es schöner, nur einen der beiden Werte eingeben zu können.

Jedoch wurde darauf verzichtet die oben genannten "Mängel" zu beseitigen, da sie für das Testen des Codes nicht relevant sind.

5.1.4 Benutzeroberfläche

Die Bedienung der Benutzeroberfläche erfolgt hauptsächlich mit dem Touchscreen. Für die Eingabe von Text kann optional eine Hardware-Tastatur, sofern vorhanden, verwendet werden. Andernfalls steht die Android-Touchscreen-Tastatur für diese Eingaben zur Verfügung.

Die Bedienelemente sind soweit möglich groß und übersichtlich angeordnet. Mögliche Fehlereingaben sollen verhindert werden.

5.2 Architektur

Im Folgenden werden grundlegende Aspekte der Softwarearchitektur von *ImkerApp* beschrieben.

5.2.1 Paketstruktur

Im Folgenden wird als tabellarische Übersicht kurz erläutert, in welche einzelnen Pakete das Projekt aufgeteilt wurde und welche Klassen in den Paketen enthalten sind.

Paketname	Beschreibung
imkerapp.actvities	In diesem Paket sind die Klassen enthalten, welche eine Activity darstellen. In den jeweiligen Klassen sind auch die Funktionen der Oberfläche implementiert, das heißt beispielsweise das "Klickevent" eines Buttons.
imkerapp.database	In diesem Paket sind die Klassen zur Verwaltung der Datenhaltung enthalten. Das heißt, in diesem Paket sind Klassen enthalten, welche Zugriff auf die Methoden zum Löschen oder Erstellen der einzelnen Tabellen haben oder auf die DAO-Objekte. Zudem ist hier die Klasse *DatabaseManager* zu finden, welche Dummy-Daten für die Applikation erstellt.
imkerapp.database.daoobjekte	Dieses Paket ist ein Unterpaket von *imkerapp.database*. Es beinhaltet die generierten Klassen von *greenDAO*[5], welche die Datensätze der einzelnen Tabellen abbilden. Das heißt, in den Klassen sind unter anderem getter- und setter-Methoden

[5] Nähere Erläuterung in Kapitel 5.2.5.

	für die Attribute der Tabelle enthalten und die Methoden zum Erstellen, Löschen und Aktualisieren der Tabellen.
imkerapp.database.daos	Auch dieses Paket ist ein Unterpaket von *imkerapp.database.* Hier sind alle generierten Klassen von *greenDAO* enthalten, welche für die einzelnen Tabellen die Schnittstellen, also die DAOs, zur Datenbank bereit stellen. Das heißt, es sind beispielsweise Methoden implementiert, welche es dem Entwickler anhand einer Abfrage möglich machen, auf bestimmte Daten aus der Datenbank zuzugreifen.

Tabelle 5.2-1 Beschreibung der Pakete von ImkerApp

5.2.2 Klassendiagramm

Im Folgenden sind die Klassendiagramme, aufgeteilt in die Pakete des Projektes, dargestellt.

Die Beschreibung jeder einzelnen Klasse ist im Header der jeweiligen Klasse zu finden.

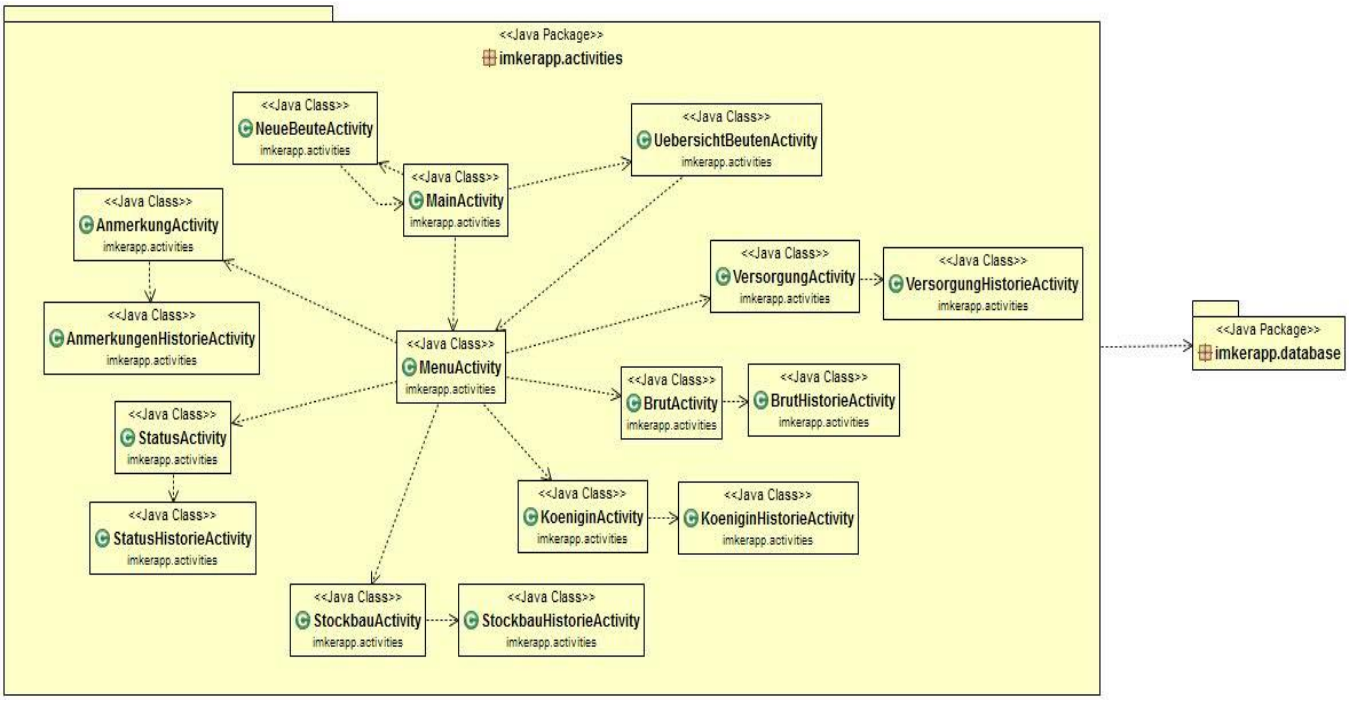


Abbildung 5.2-1 Klassendiagramm Paket imkerapp.activities

In der Abbildung 5.2-1 ist das Klassendiagramm zugehörig zum Paket imkerapp.activities abgebildet. Es zeigt deutlich, dass die MainActivity Zugriff auf alle anderen Klassen hat. So greift sie auf NeueBeuteActivity und UebersichtActivity zu, aber auch auf die MenuActivity, über welche sie auch Zugriff auf die restlichen Klassen erhält. UebersichtActivity greift noch auf die MenuActivity zu, da sie somit die ausgewählte Beute anzeigen kann. Zudem ist ersichtlich, dass die MenuActivity auf die einzelnen Activities der Beute zugreift, aber nicht auf deren Historie. Auf diese haben nur die "Hauptactivities" der Beute Zugriff. Des Weiteren greifen Klassen aus diesem Paket auf das andere Paket imkerapp.database zu, welche die Klassen zur Verwaltung der Datenhaltung enthält.

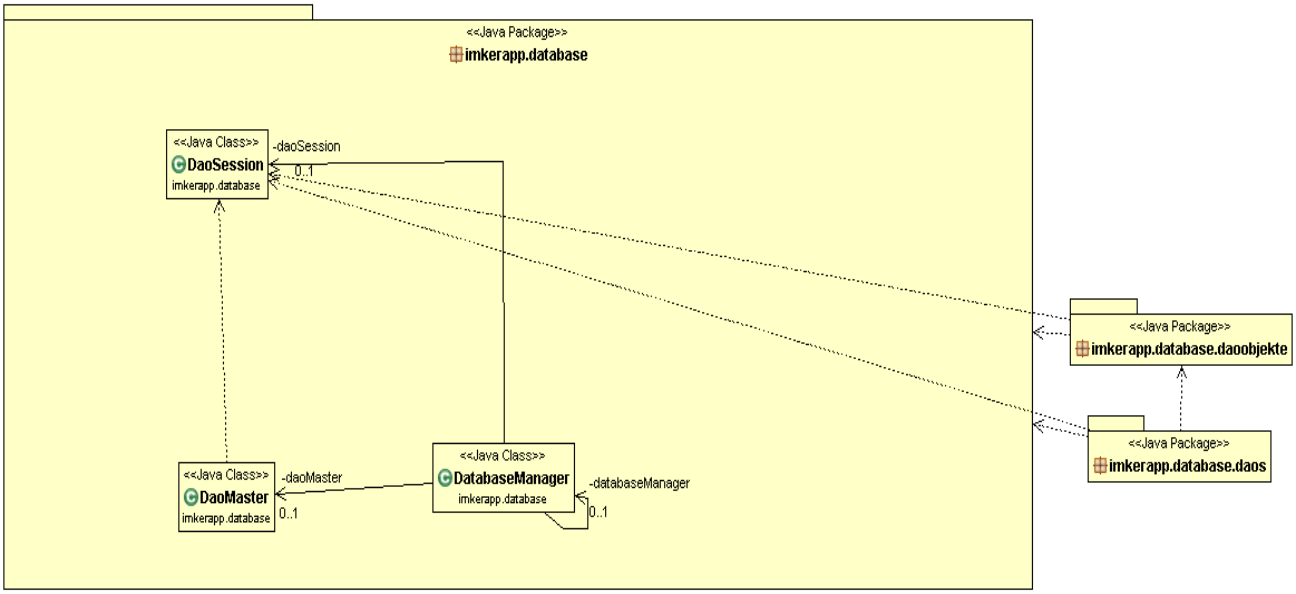


Abbildung 5.2-2 Klassendiagramm Paket imkerapp.database

In der Abbildung 5.2-2 ist das Klassendiagramm zugehörig zum Paket imkerapp.database abgebildet. Es zeigt, dass die Klasse DatabaseManager Zugriff auf die Klassen DaoSession und DaoMaster hat. Die Klasse DaoMaster hat zudem Objekte der Klasse DaoSession. Außerdem ist hier dargestellt, dass die Unterpakete von imkerapp.database, nämlich imkerapp.database.daoobjekte und imkerapp.database.daos, ebenfalls Objekte der Klasse DaoSession enthalten. Und das imkerapp.database.daos Zugriff auf Klassen im Paket imkerapp.database.daoobjekte hat.

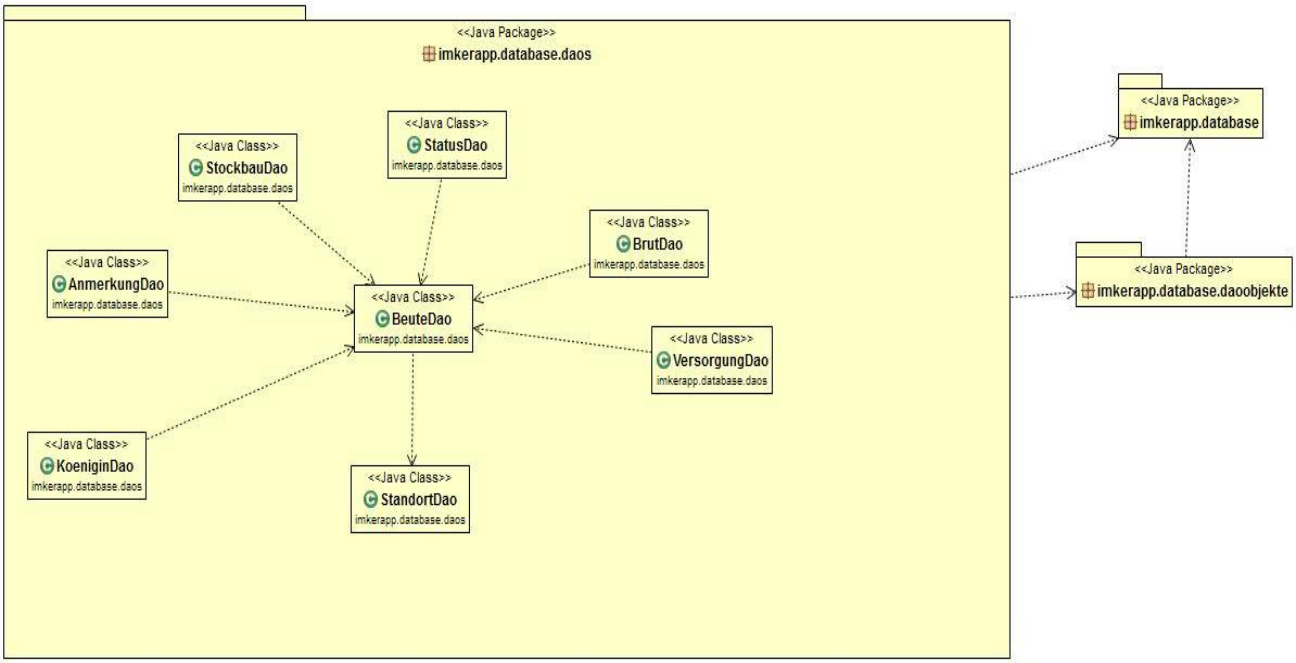


Abbildung 5.2-3 Klassendiagramm Paket imkerapp.database.daos

In der Abbildung 5.2-3 ist nun das Klassendiagramm zum Paket imkerapp.database.daos zu sehen. Hier wird deutlich, dass die "VerwaltungsDAOs" einer Beute alle auf BeutenDAO zugreifen. Außer der StandortDAO, auf diesen greift BeuteDAO zu. Auch hier sind wieder die Pakete mit abgebildet, auf welche das Paket imkerapp.database.daos Zugriff hat.

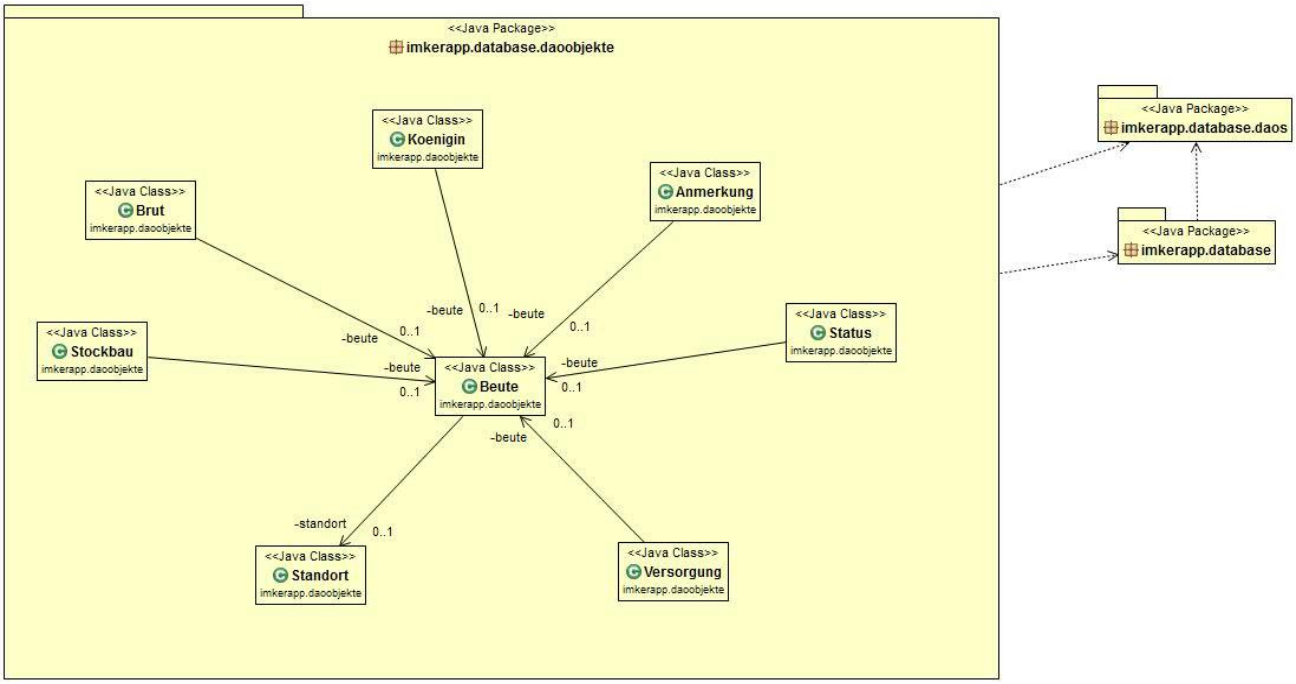


Abbildung 5.2-4 Klassendiagramm Paket imkerapp.database.daoobjekte

Das Klassendiagramm in Abbildung 5.2-4 zum Paket imkerapp.database.daoobjekte, hat sehr viel Ähnlichkeit mit dem in Abbildung 5.2-3. Nur, dass hier nun die Klassen der DAO-Objekte dargestellt sind. Es wird wieder deutlich, dass alle "Verwaltungsobjekte" einer Beute auf die Beute zugreifen, nur die Beute greift auf den Standort zu.

5.2.3 Entwurfsmuster

Unter diesem Punkt werden die verwendeten Entwurfsmuster beschrieben.

a) Data Access Object (DAO) (Tutorialspoint)

   Der Zugriff auf die Datenbank erfolgt über den DatabaseManager, der beim ersten Aufruf aus der MainActivity den Zugriff auf die SQlite Datenbank übergeben bekommt. Der DaoMaster dient dazu, die gesamte Datenbank zu erstellen oder zu löschen. Dazu greift dieser auch auf die einzelnen DAOs zu.

   Über die DaoSession wird dann der Zugriff auf die einzelnen Tabellen (die DAOs z.B. BeutenDAO) realisiert, wo es nun möglich ist mit den DAO-Objekten (Beute) Daten einzufügen.

b) Singelton (Freeman & Freeman)

Die Klasse DatabaseManager wird als Singelton implementiert. Diese Klasse kann generiert Dummy-Daten für die Datenbank und enthält die getter- und setter-Methoden für die DAO-Klassen. So ist der DatabaseManager als zentrale Schnittstelle zur Datenhaltung gedacht, wodurch er hier als Singelton implementiert wurde.

c) Modell-View-Controller (Freeman & Freeman)

In der Software wurde das MVC-Modell umgesetzt, um eine gute Austauschbarkeit zu gewährleisten. Jede der Schichten übernimmt eine klare Aufgabe:

1. UserInterface

2. Verwaltung

3. Datenhaltung

Die Schicht „UserInterface“ stellt die einzelnen grafischen Oberfläche und deren Controller zur Verfügung. Die grafischen Oberflächen sind in diesem Fall die Activities, die Controller die onClickListener der jeweiligen Buttons.

In der Schicht „Verwaltung“, also der Fachkonzeptschicht, befinden sich die Funktionen für die Verwaltung der Daten der Software. Diese ist in der Klasse DatabaseManager umgesetzt.

In der letzten Schicht, der „Datenhaltung“ befindet sich eine SQlite Datenbank, welche sich um die persistente Speicherung der Daten kümmert.

5.2.4 Datenhaltung

Die Datenbank hinter *ImkerApp* basiert auf SQlite, welches eine Programmbibliothek ist, die eine relationale Datenbank bereit stellt. Die Datenbank enthält verschiedene Tabellen zum Speichern der Daten, welche im folgenden Entity-Relationship-Modell dargestellt sind.

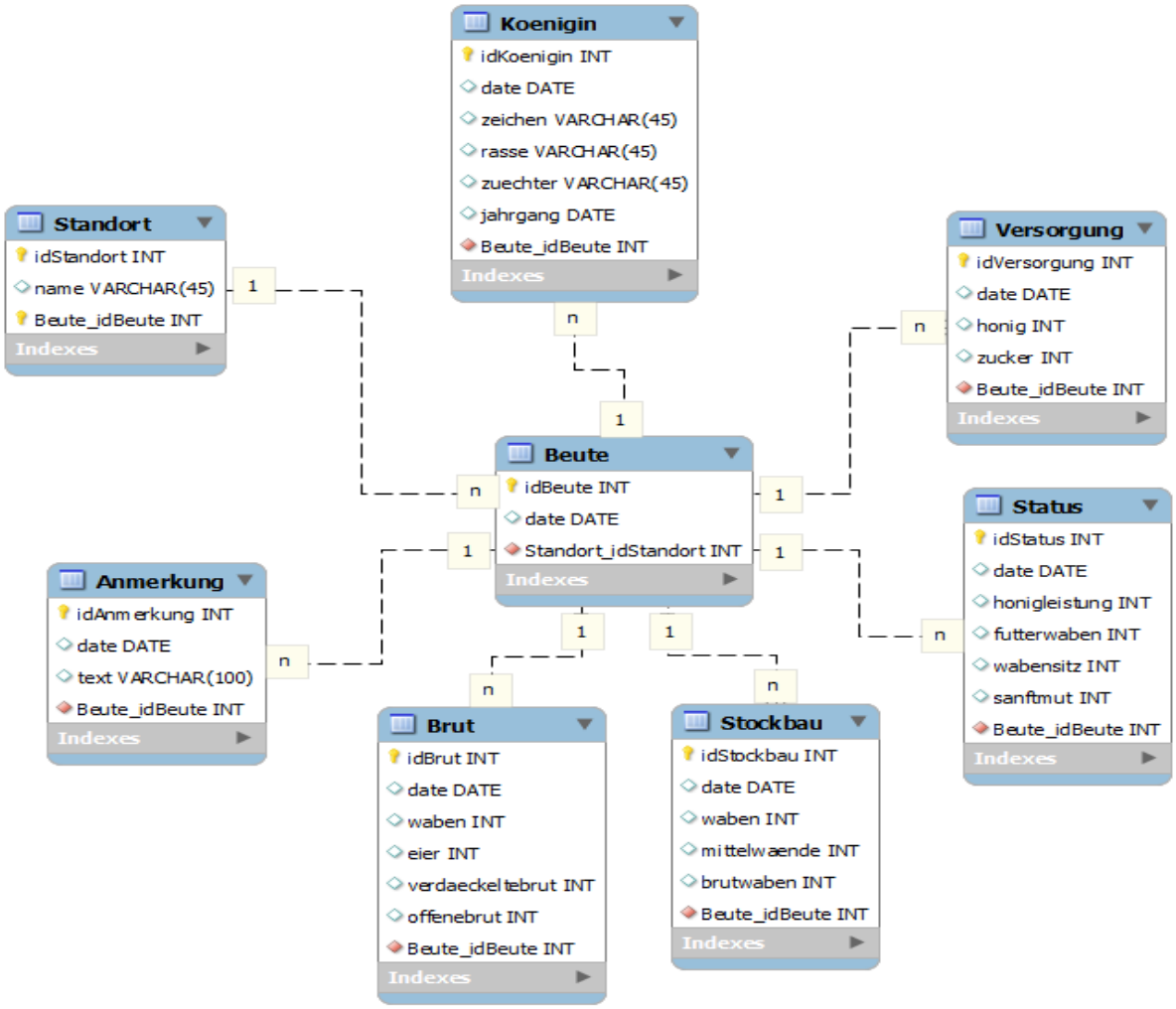


Abbildung 5.2-5 Entity-Realtionship-Modell zeigt das Datenbankschema

Hier ist nun zu sehen, dass es für den Standort, die Beute und deren "Verwaltung" einzelne Tabellen gibt. Die ID des Standortes wird dabei in der Beutetabelle gespeichert. Die IDs der "Verwaltungstabellen" direkt in der Beutentabelle. Somit können die Daten eindeutig zugeordnet werden.

5.2.5 Generierung der Datenhaltung mit greenDAO

Die Datenbank mit ihren Tabellen und die Klassen zur Verwaltung der Datenbank in den Paketen imkerapp.database wurden mit Hilfe von *greenDAO* generiert. *greenDAO* ist ein OpenSource Framework zur Verwaltung von persistenten Daten. Es gehört zu den sogenannten Object Relation Mapper (ORM) und bietet somit einen objektorientierten Zugriff auf eine SQlite Datenbank.

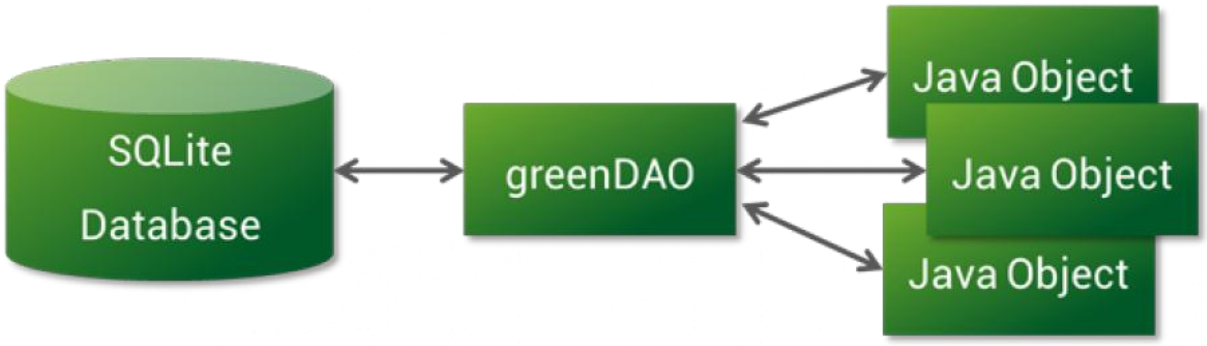


Abbildung 5.2-6 greenDAO

Im Folgenden sind die Vorteile aufgeführt, welche *greenDAO* bietet und welche zur Wahl von *greenDAO* beigetragen haben:

- maximale Performance
- einfache API
- für Android optimiert
- minimaler Speicherverbrauch
- kleine Bibliotheksgrößen, auf das Wesentliche beschränkt

Zum Vorteil Performance ist im folgenden noch einmal eine Grafik zu sehen, welche die Performance von *greenDAO* und ORMLite[6] vergleicht. *greenDAO* ist hier klar im Vorteil.

[6] Eine andere Möglichkeit, mit Hilfe von ORM eine geeignete Datenbank zu erstellen. (ormlite)

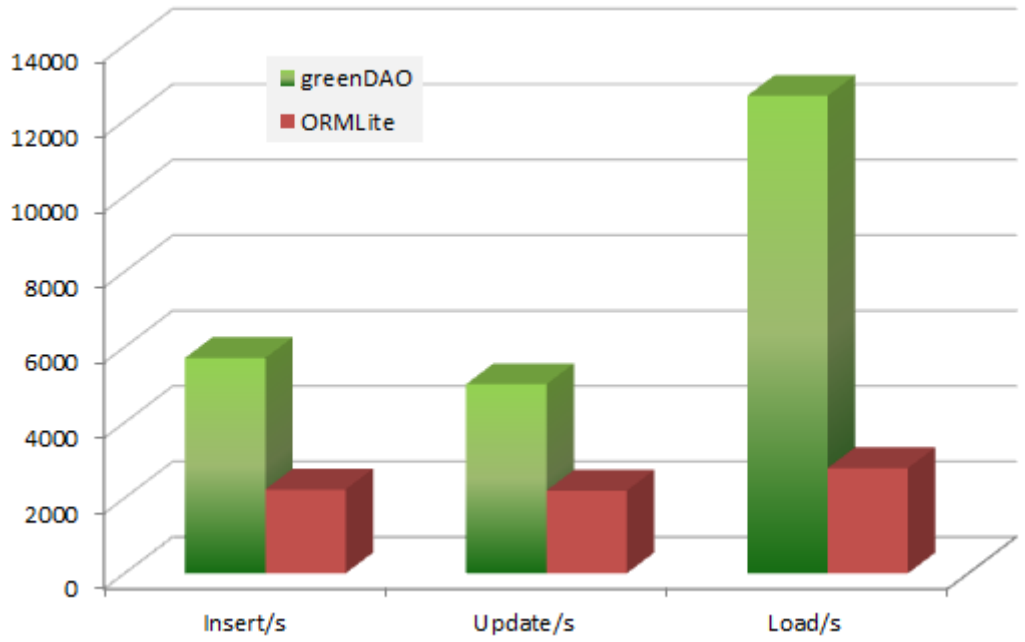


Abbildung 5.2-7 Performancevergleich greenDAO mit ORMlite

Zur Erstellung der Datenbank wird ein neues Javaprojekt angelegt "ImkerAppGenerator" in welches folgende .jar Dateien eingebunden werden:

- freemarker.jar
- greenDAO-generator-javadoc.jar
- greenDAO-generator.jar

Anschließend wird das Metamodell erstellt. Hier werden einmal die Version und das Paket festgelegt:

```
Schema schema = new Schema(1, "imkerapp.database");
```

Zum anderen aber auch die Entities angelegt, hier als Beispiel für Königin.

```
Entity koenigin = schema.addEntity("Koenigin");
koenigin.addIdProperty();
koenigin.addDateProperty("date").notNull();
koenigin.addStringProperty("zeichen").notNull();
koenigin.addStringProperty("rasse").notNull();
koenigin.addStringProperty("zuechter").notNull();
koenigin.addDateProperty("Jahrgang").notNull();
Property beutenIdkoenigin =
       Koenigin.addLongProperty("beutenId").notNull().getProperty();
       koenigin.addToOne(beute, beutenIdkoenigin);
```

Die Entity Königin wird dem vorher angelegen Schema hinzugefügt. Zudem erhält die Entity verschiedene Spalten, welche beispielsweise den Typ String, INT oder Date haben können. Im Anschluss wird die ID der Entity Beute noch in Königin hinzugefügt.

Am Ende des Metamodells wird dann noch der Code generiert und zwar in dem Ordner, welcher in Klammern angegeben wird:

```
new DaoGenerator().generateAll(schema, "../ImkerAppGenerator/Sample/src-gen");
```

Wird das Projekt nun einmal gestartet und läuft bis zum Ende durch, ist in dem angegeben Ordner der generierte Code zu finden. Nun kann man mit den DAOs im gewünschten Projekt weiterarbeiten. (greenDao)

6 Vergleichskriterien für die Evaluation

Im Folgenden werden die schon in der Aufgabenstellung erwähnten Vergleichskriterien für die ausgewählten Testframeworks genauer beschrieben.

6.1 Beschreibung

In der Beschreibung eines Programms wird kurz erläutert, welche Funktionen das jeweilige Programm bietet und in welche Testphase und Testart es eingeordnet werden muss. Diese kurze Erläuterung findet keinen weiteren Einfluss in die Evaluierung des jeweiligen Programms, sondern soll nur eine kurze Einleitung geben.

6.2 Installationskomplexität

Die hier herangezogen Kriterien beziehen sich auf den Installationsaufwand und den Aspekt, ob für den Entwickler eine entsprechende Installationsanleitung zur Verfügung steht. Die Evaluierung des Installationsaufwandes erfolgt, nachdem kurz erläutert wurde, wie die Installation funktioniert. Es existieren vier verschiedenen Schwierigkeitsstufen, welche in der folgenden Tabelle erläutert werden.

Schwierigkeitsgrad	***Beschreibung***
Einfach	Auf der Internetseite des Herstellers kann sich der Entwickler eine Jar-Datei herunterladen und diese auf seinem Rechner starten. Eine andere Möglichkeit besteht darin, in der Entwicklungsumgebung *eclipse* über den Marketplace ein Plugin des Programms zu installieren, soweit dieses vorhanden ist.
Aufwändig	Der Entwickler muss bei der Installation des Programms Systemvariablen verändern oder setzen. Des Weiteren muss eventuell das Betriebssystem bei der Installation berücksichtigt werden.
Komplex	Der Entwickler muss bei der Installation des Programms die Abhängigkeiten von getrennter zu installierter Software beachten. Zum Beispiel muss bei einem Java Programm die Installation einer Java-Version Voraussetzung sein.

Extrem	Der Entwickler muss das Programm selber kompilieren und es gibt enge Abhängigkeiten von bestimmten Software-Versionen. Zum Beispiel läuft das Programm nur mit der Java-Version 5, aber nicht mit der Version 6.

Tabelle 6-1 Schwierigkeitsgrad der Installation eines Programms

6.3 Einarbeitungskomplexität

Bei den hier ausgewählten Kriterien handelt es sich um die bereitgestellten Dokumentationen, Schulungen und die Community des Programms. Des Weiteren wird evaluiert, inwieweit es dem Entwickler möglich ist, sich mit den bereitgestellten Hilfestellungen in das Programm einzuarbeiten. Die Bewertung wird, nach einer kurzen Erläuterung der bereitgestellten Hilfsmittel, in vier Schwierigkeitsgraden erfolgen, welche in der folgenden Tabelle erläutert werden.

Schwierigkeitsgrad	***Beschreibung***
Niedrig	Der Hersteller stellt auf seiner Internetseite ausreichende Dokumentationen für die Verwendung des Programmes zur Verfügung. Des Weiteren besteht eine große Community, welche auch bei Schwierigkeiten Hilfe bietet und wo der Entwickler selber seine Fragen posten kann, um dann eine Antwort zu erhalten.
Mittel	Der Hersteller stellt nur eine kurze Einleitung seines Programms zur Verfügung und der Entwickler muss selber noch nach weiteren Informationen zur Verwendung recherchieren. Eine Community steht zur Verfügung, bietet aber auch nicht bei jedem Problem eine Hilfe an.
Hoch	Der Hersteller bietet für sein Programm eine Schulung oder ein Seminar an, an welchem der Entwickler erst einmal teilnehmen muss, um mit dem Programm arbeiten zu können. Zudem stellt der Hersteller eine Dokumentation zur Verfügung, um die in der Schulung oder im Seminar gelernten Inhalte noch einmal zu verinnerlichen.

Extrem hoch	Vom Hersteller wird keine Dokumentation oder Schulung zur Verfügung gestellt. Des Weiteren existiert keine Community für das Programm. Es ist lediglich der Quellcode des Programms und für eine Beispielanwendung vorhanden.

Tabelle 6-2 Schwierigkeitsgrade der Einarbeitung eines Programms

6.4 Anlegen eines Testprojektes

Hier wird kurz erläutert wie ein Testprojekt für das jeweilige Programm angelegt wird. Kriterien für die Evaluierung beziehen sich auf die Dokumentation des Herstellers zum Anlegen eines solchen Testprojektes und den dahinter stehenden Aufwand. Diese Kriterien werden in verschiedene Schwierigkeitsgrade eingeteilt, welche in der folgenden Tabelle aufgeführt werden.

Schwierigkeitsgrad	Beschreibung
Niedrig	Der Hersteller oder die Community stellen eine ausführliche Beschreibung für das Anlegen eines Testprojektes zur Verfügung.
Mittel	Der Hersteller und die Community stellen eine Beschreibung für das Anlegen eines Testprojektes zur Verfügung. Allerdings muss der Entwickler selber noch verschiedene Einstellungen vornehmen, welche in der Beschreibung nur dürftig angedeutet werden.
Hoch	Der Hersteller bietet nur eine kurze Beschreibung für das Anlegen des Testprojekts. Der Entwickler muss selber noch recherchieren, um das Testprojekt anlegen zu können. Auch die Community bietet kaum Hilfe dafür an.
Extrem hoch	Weder der Hersteller noch die Community bieten Hilfestellungen für das Anlegen eines Testprojekts an. Der Entwickler muss selber sehen, wie er das Testprojekt erfolgreich angelegt bekommt.

Tabelle 6-3 Schwierigkeitsgrade Anlegen eines Testprojekts

6.5 Testmöglichkeiten

Hier wird erläutert, welche Testmöglichkeiten ein Programm zur Verfügung stellt und wie diese bei der Testapplikation *ImkerApp* angewendet wurden. Zudem wird evaluiert, wie die einzelnen Testmöglichkeiten sich auf Android Applikationen anwenden lassen.

6.6 Bewertung

Unter diesem Punkt werden alle vorher evaluierten Punkte noch einmal zusammengefasst, um eine endgültige Bewertung des Programms beziehungsweise Frameworks zu erhalten.

7 Evaluierung der funktionierenden Frameworks

Im Folgenden werden Frameworks nach den in Kapitel 6 beschriebenen Kriterien evaluiert, welche erfolgreich unter Android eingesetzt werden konnten. Es gibt noch eine Vielzahl anderer Frameworks, welche in dieser Bachelorarbeit hätten evaluiert werden können. Es wurde sich in diesem Kontext für Frameworks entschieden, welche speziell für den Einsatz unter Android entwickelt wurden und welche sich unter Java schon bewährt haben. Zudem sollten die gewählten Frameworks die Matrix zur Einordnung in die Testphase und Testart abdecken können. Zunächst aber eine tabellarische Einordnung der gewählten funktionierenden Frameworks in die jeweiligen Testphasen und Testarten.

Testphase x Testart	**Funktionstest**	**Kontroll-flusstest**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest	JUnit Robolectric		Metrics	Android Lint Android Findbugs Checkstyle PMD
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	Robotium monkey		Traceview	

Tabelle 7-1 Einordnung funktionierende Frameworks in Testphase und Testart

7.1 JUnit

7.1.1 Beschreibung

JUnit ist ein Framework mit dem es möglich ist Unittests zu schreiben und diese zu jedem Zeitpunkt als Sammlung - eine sogenannte Testsuite- ablaufen zu lassen. JUnit bietet unter anderem verschiedene Methoden, zum Beispiel assertTrue(), assertFalse() oder assertEquals(), welche es dem Entwickler ermöglichen, Bedingungen im Code nachzuprüfen.

JUnit wird in der Phase des Komponenten- und Modultests eingesetzt, wo dann geprüft wird, ob die implementierten Methoden ihren Zweck erfüllen. Dabei wird vor allem gegen die vorher festgelegten Spezifikationen getestet.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest	x			
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 7-2 Einordung JUnit in die Testphasen bzw. Testarten

7.1.2 Installation

JUnit ist bereits im Android SDK[7] enthalten. Die Installation von JUnit wird daher als einfach eingestuft, da der Entwickler keinen weiteren Aufwand mit der Installation hat, wenn das Android SDK bereits erfolgreich installiert wurde.

7.1.3 Einarbeitung

JUnit, welches speziell für Android angepasst wurde, enthält verschiedene Arten des Testens. Diese sind alle auf der Website der Entwickler beschrieben. Entwickler, welche schon einmal mit JUnit gearbeitet haben, werden keine Probleme haben sich mit den neuen Testmöglichkeiten zurechtzufinden. Aber auch Anfänger auf diesem Gebiet werden dank der guten Dokumentation, welche zusätzlich mit Beispielen angereichert wurde, keine Probleme haben sich schnell in die Syntax einzuarbeiten. Auch die Community, welche sich im Laufe der Zeit um JUnit gebildet hat, bietet allerlei Hilfsmittel und Sourcecodebeispiele an. (android developer)

[7] http://developer.android.com/sdk/index.html

Die Komplexität der Einarbeitung wird daher als niedrig eingestuft, da dem Entwickler mit Hilfe von Dokumentationen und Beispielen ein guter Einstieg in die Verwendung von JUnit gegegeben wird.

7.1.4 Anlegen eines Testprojektes

Um ein Testprojekt im Android SDK anzulegen, muss zunächst ein Android Application Project existieren, welches mit JUnit getestet werden soll.

Nun kann ein Android Test Project angelegt werden, welches bereits auf JUnit basiert. Dieses findet man in der Entwicklungsumgebung unter *File → New → Other → Android.* Nachdem man Android Test Project ausgewählt hat erscheint ein Dialogfenster in welches man den Namen des Projektes einträgt. Klickt man nun auf *Next*> kommt man in das Dialogfenster in welchem man das Projekt auswählt, welches getestet werden soll. Im nächsten Fenster wählt man nun die Android Version aus, auf der man die Applikation testen möchte. Hier empfiehlt es sich die minimale Version des Application Projektes zu nehmen. Im Anschluss daran klickt man auf Finish und das Testprojekt wird erstellt.

Zum Anlegen einer Testklasse geht man nun in das Testprojekt, macht mit der Maus einen Rechtsklick auf den Ordner *src*, geht dann auf *New* und wählt JUnit Test Case aus. In dem Dialogfenster muss der Testklasse wieder ein Name zugewiesen werden, aber auch eine Superclass[8]. Hierbei ist beachten, dass man eine Superclass aus dem Ordner *andoid.test* auswählt, da Android nicht mit den ursprünglichen JUnittests zurechtkommt.

Der Aufwand, um ein Testprojekt erfolgreich anzulegen, wird bei JUnit als mittel eingestuft. Es existieren Beschreibungen und Hilfestellungen, um ein Testprojekt erfolgreich anzulegen. Allerdings gibt es kaum Erläuterungen, welchen Zweck die einzelnen Superclasses, welche ausgewählt werden können, erfüllen. Es wird bei den meisten Hilfestellungen nur auf zwei bis drei Superclasses verwiesen. Diese reichen auch aus, um den Großteil des Sourcecodes zu testen. Aber erst wenn der Entwickler ein wenig genauer recherchiert findet er vereinzelte Hinweise auf die Bedeutung der übrigen Superclasses.

7.1.5 Testmöglichkeiten

Bei dem vom Android SDK verwendeten JUnit gibt es verschiedene Möglichkeiten seine Klassen zu testen. Legt man beim Anlegen einer Testklasse als Superclass zum Beispiel *junit.framework.TestCase* fest, können alle Klassen getestet werden, welche nicht auf die Android API zugreifen. Möchte man nun aber Klassen testen, welche auf die Android API zugreifen, gibt es einige Superclasses die gewählt werden können.

ImkerApp beispielsweise wurde mit der Superclass *ActivityInstrumentationTestCase2* getestet. Diese Klasse bietet die Möglichkeit, eine Activity „fernzusteuern“ und auf ihre Views zuzugreifen. So lassen sich Werte auslesen, Formularfelder füllen und Schaltflächen und Menüoptionen anklicken.

[8] Basisklasse einer anderen Klasse. Die zu testende Klasse nutzt Funktionen aus der Basisklasse.

```
public class AnmerkungActivityTest extends
        ActivityInstrumentationTestCase2<AnmerkungActivity> {
```

Abbildung 7.1-1 Klassendefinition JUnit

In Abbildung 7.1-1 ist zu erkennen, dass die Testklasse *AnmerkungActivityTest* , welche zu der BeispielApplikation *ImkerApp* gehört, von der Superclass *ActivityInstrumentationTestCase2* erbt. Zudem ist der Superclass der zu testenden Android Activity zugeordnet.

Nun können in der *setUp()* Methode die benötigte Activity *AnmerkungActivity* erstellt werden, sowie die zu testenden Button, Textviews oder EditText-Felder.

```
protected void setUp() throws Exception {
    super.setUp();
    // Beutennummer 20 übergeben
    Config.getConfig().setBeutenId((long)20);
    // Activity Anmerkung erstellen
    anmerkungActivity=this.getActivity();


    // zu testende Button und EditText-Felder erstellen
    historie = (Button) anmerkungActivity.findViewById(R.id.anmerkungHistorie);
    speichern = (Button) anmerkungActivity.findViewById(R.id.anmerkungSpeichern);
    anmerkungText = (EditText) anmerkungActivity.findViewById(R.id.anmerkungTexte);

}
```

Abbildung 7.1-2 setUp() Methode

Um nun ein EditText-Feld zu testen wird eine neue Testmethode erstellt, in welcher man beispielsweise einen Text in das Feld schreibt und dann prüft, ob dieser korrekt wieder ausgelesen wird. In Abbildung 7.1-3 ist dieses Beispiel in *ImkerApp* umgesetzt worden.

```
public void testText() {
    //Text
     CharSequence text = "Honig";

     //Text in das Feld Anmerkungtext schreiben
     anmerkungText.setText(text);

     //Prüfen, ob es auch richtig wieder rausgelesen wird.
      assertEquals("Honig", anmerkungText.getText().toString());

}
```

Abbildung 7.1-3 EditText-Feld testen

Die nächste Möglichkeit eines Tests besteht darin, dass das Verhalten eines Buttons überprüft wird, welcher bei einem Klick eine andere Activity öffnet.

```
public void testButtonHistorie() {


    // registriert die nächste Activity
    ActivityMonitor activityMonitor = getInstrumentation().addMonitor(AnmerkungenHistorieActivity.class.getName(), null, false);

    anmerkungActivity.runOnUiThread(new Runnable() {

        @Override
        public void run() {
            // klick Button und öffne nächste Activity
            historie.performClick();

        }
    });

    Activity historieActivity = getInstrumentation().waitForMonitor(activityMonitor);
      // prüft, ob nächste Activity geöffnet wird und schließt diese dann wieder
      assertNotNull(historieActivity);
      historieActivity .finish();

}
```

Abbildung 7.1-4 Button testen

Dabei ist wichtig zu beachten, dass das Auslösen des Klicks in einem Thread läuft.

Es gibt noch eine Vielzahl weiterer Superclasses für Android JUnittests. Andere Superclasses werden für die Fälle eines Service, eines Content Providers oder wenn Ressourcen oder das Dateisystem getestet werden sollen, verwendet.

- ProviderTestCase2: Ermöglicht den Test von Content Providern. Mittels Content Providern lässt sich beispielsweise eine Schnittstelle zu den persistenten Daten einer Anwendung für andere Anwendungen realisieren.
- ServiceTestCase: Ermöglicht isolierte Tests von Services. Hierüber lässt sich das Verhalten eines Services über seine Schnittstellen in allen Phasen seines Lebenszyklus testen.
- AndroidTestCase: Ermöglicht das Testen einer Anwendung. Dieser Testfall ermöglicht den Zugriff auf Ressourcen oder auf den Context der Anwendung. Über den Context hat man beispielsweise Zugriff auf das Dateisystem oder die Datenbank einer Anwendung.

Zudem ist es möglich der Testklasse einen Context als Superclass zuzuweisen, welcher beispielsweise eine eigene Datenbank oder ein eigenes Dateisystem nur für die Testzwecke zur Verfügung stellt. Die drei häufigsten verwendeten Test-Contexte sind:

- MockContext: Durch Überschreiben seiner Methoden kann man den Context einer Anwendung simulieren.
- IsolatedContext: Verhindert den Zugriff auf einen Großteil der Systemumgebung, gibt dem Betriebssystem jedoch die Möglichkeit, mit dem Context zu kommunizieren. Dieser Context wird beispielsweise verwendet, um Broadcast Receiver und Services zu testen oder um zu prüfen, ob die Anwendung die Berechtigung hat eine bestimmte URI aufzurufen.

- RenamingDelegatingContext: Verhindert, dass die Testklasse Daten der Anwendung verändert. Bei Verwendung dieses Context wird allen im Test verwendeten Dateien und Datenbanken ein Präfix vorangestellt. So greift ein Testfall beispielsweise nicht auf die Datenbank meine.db zu, sondern auf die strukturell identische test.meine.db. Für den Programmierer des Tests ist das völlig transparent. Er schreibt den Test so, als würde er auf die Original-Datenbestände zugreifen.

Um einen Test erfolgreich durchlaufen zu lassen wird entweder der Emulator benötigt oder ein externes Gerät, welches die mindeste Android Version der Applikation unterstützt. Die Testergebnisse werden in einem extra Fenster in der Entwicklungsumgebung angezeigt. War der Test erfolgreich wird der grün angezeigt, bei einem Fehler rot. Bei einem Fehler des Tests wird eine kurze Fehlerbeschreibung ausgegeben, welche meist im LogCat, der Debugausgabe von Android, ausführlicher angezeigt wird.

7.1.6 Bewertung

JUnit ist ein einfaches Framework, welches dem Entwickler schnell und effizient die Möglichkeit bietet Unittests für seine Applikation zu entwickeln. Die Installation fällt weg, da es ja schon im Android SDK enthalten ist und dank der ausführlichen Dokumentation ist es jedem Entwickler möglich Tests zu schreiben. Am Anfang muss man sich erst einmal in die verschiedenen Superclasses einlesen, um genau zu wissen, welche nun für den eigenen Test am besten geeignet ist. Hier ist meist eine Recherche erforderlich, da man die gewünschten Informationen nicht sofort findet. Ist dieser Teil aber erst einmal abgeschlossen, geht der Rest schnell von der Hand. Auch dank einer präzisen Fehlerausgabe im LogCat, können Fehler schnell korrigiert werden.

7.2 Robolectric

7.2.1 Beschreibung

Robolectric ist, wie auch JUnit schon, ein Framework welches es dem Entwickler erlaubt, schnell und einfach Unittests für seine Applikation zu verfassen. Es wurde speziell für Android Applications entwickelt. Der Vorteil gegenüber JUnit liegt darin, dass die Tests bei Robolectric ganz ohne Emulator oder externem Gerät ablaufen. Dieses wird möglich, da jede Klasse der Android VM zur Laufzeit durch sogenannte Shadow-Objekte ersetzt wird. Diese Objekte gibt es von Robolectric für die meisten der Klassen im SDK. So existiert beispielsweise *ShadowImageView* für *ImageView*. Es ist aber auch möglich, über die API das Shadow-Objekt direkt abzufragen, wenn man zum Beispiel einen internen Status benötigt. (Google Group, 2010) (Preussler, 2012)

Robolectric wird in der Testphase des Komponenten- und Modultests eingesetzt. Hier wird einmal gegen die Spezifikationen getestet, aber auch die Funktionsweise der einzelnen Methoden getestet.

Testphase x Testart	Funktions-test	Kontrollfluss-test	Softwaremes-sung	Stil-/ Codeanalyse
Komponententest/ Modultest	X			
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 7-3 Einordung Robolectric in die Testphasen bzw. Testarten

7.2.2 Installation

Für die Installation wird ausschließlich die .jar Datei[9] benötigt. Diese Datei muss dann in das Testprojekt eingebunden werden. Die Komplexität der Installation wird daher als einfach bewertet.

7.2.3 Einarbeitung

Robolectric bietet durch die Shadow-Objekte eine einfache Art des Testens an. Auf der Seite der Entwickler findet man ein einführendes Beispiel, welches einem die Möglichkeiten von Robolectric aufzeigt. Wer schon einmal mit JUnit gearbeitet hat, ist hier im Vorteil. Aber durch die leicht verständliche Dokumentation ist es auch für Anfänger einfach sich in die Syntax einzuarbeiten. Am Anfang des Testens kann es Probleme geben, da die neueren .jar Dateien manche Funktionen nicht mehr unterstützen. Hier muss man dann schauen, was genau man testen möchte und in welcher .jar Datei diese Funktionen enthalten sind. (Google Group, 2010)

Der Aufwand der Einarbeitung wird daher als mittel eingestuft. Es existieren zwar durchaus Hilfestellungen und Dokumentationen für die Einarbeitung und auch die Community bietet einen guten Einstieg in Robolectric, allerdings muss der Entwickler immer noch selber recherchieren, um die für ihn passende .jar-Datei zu finden. Dieses kann Zeit in Anspruch nehmen, da erst jede .jar-Datei angeschaut werden muss, um herauszufinden, welche Funktionen diese anbietet.

7.2.4 Anlegen eines Testprojekts

Das Anlegen des Testprojektes ist um einiges aufwändiger als in JUnit. Zunächst muss wieder ein Android Application Project existieren. In dieses muss nun ein *Folder* mit beispielsweise dem Namen *test* angelegt werden.

[9] http://pivotal.github.io/robolectric/download.html

Um nun das Testprojekt zu erstellen wird ein einfaches Java Project angelegt. Nachdem das Projekt einen Namen erhalten hat, muss noch der *src* vom BuildPath entfernt werden. Hierfür klickt man in dem Dialogfenster auf *src* und anschließend auf *Remove source folder 'src' from build path*. Nun muss noch über *Link additional source* der *Folder test* im Android Project hinzugefügt werden. Im Tab *Projects* muss noch das Android Project hinzugefügt werden, welches getestet wird. Nachdem all das geschehen ist, müssen unter *BuildPath* noch die entsprechenden .jar Dateien von Junit und Robolectric hinzugefügt werden, sowie die android.jar und die maps.jar aus dem Ordner der mindesten Android Version der Applikation.

Zu guter Letzt muss noch die *RunConfiguration* für JUnit angepasst werden. Es wird der Radiobutton für *Run all tests in the selected project, package or source folder* ausgewählt und ihm das Android Application Project hinzugefügt. Der TestRunner muss auf JUnit4 gesetzt werden. Nun muss noch die *Use Configuration* in *Eclipse Junit Launcher* geändert werden und die *Working Direction* muss auf *other* stehen.

Um nun eine Testklasse anzulegen geht man in den *Folder test* im Android Application Project und erstellt eine neue Klasse. Um diese durchlaufen zu lassen, wählt man die vorher eingestellte *RunConfiguration* aus. (Google Group, 2010)

Der Aufwand, um ein Testprojekt erfolgreich anzulegen, wird hier als mittel eigestuft. Es sind einige Einstellungen vorzunehmen und es kann passieren, dass der Entwickler eine Einstellung vergisst oder diese im falschen Ordner vornimmt. So muss dann der ganze Aufwand noch einmal von vorne betrieben werden. Allerdings bietet der Hersteller als auch die Community eine Vielzahl von Dokumentationen an. Einige dieser Dokumentationen sind ausführlich, andere beschreiben die einzelnen notwendigen Schritte eher dürftig. So kann aber jeder Entwickler auswählen, welche Art der Dokumentation ihm am besten gefällt.

7.2.5 Testmöglichkeiten

Robolectric bietet auf verschiedene Art und Weisen an den Quellcode zu testen. Wichtig bei allen Tests sind jedoch die Annotationen, welche Auskunft darüber geben, was der Test bezwecken soll.

Damit der Test überhaupt lauffähig ist, muss die Annotation *@RunWith()* von JUnit verwendet werden.

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

Abbildung 7.2-1 Klassendefinition Robolectric

Alle Methoden, welche vor einem Test ausgeführt werden sollen, müssen mit *@Before* gekennzeichnet werden, wie in der Abbildung 7.2-2 in der Klasse *AnmerkungActivity* zu sehen ist.

```
@Before
public  void setUp() throws Exception {
    Config.getConfig().setBeutenId((long)20);


    anmActivity = Robolectric.buildActivity(AnmerkungActivity.class).create().get();
    historie = (Button) anmActivity.findViewById(R.id.anmerkungHistorie);

}
```

Abbildung 7.2-2 setUp() Methode Robolectric

Die Testmethoden selber werden mit *@Test* markiert.

```
@Test
public void testTexte() {
    CharSequence text = "Futter";

    textfeld.setText(text);

    assertEquals("Futter", textfeld.getText().toString());


}
```

Abbildung 7.2-3 @Test Kennzeichnung Robolectric

Wie schon in der Beschreibung erwähnt, arbeitet Robolectric mit Shadow-Objekten. Beim Testen von ImkerApp haben diese ihren Einsatz beim Testen von Button gefunden, welche eine neue Activity öffnen.

```
@Test
public void pressButton() {
    // Klick auf Button
    historie.performClick();
    // AnmerkungActivity als shadow Instanz
    ShadowActivity shadowActivity = Robolectric.shadowOf(anmActivity);
    // nächste Activity, die gestartet wird
    Intent startedIntent = shadowActivity.getNextStartedActivity();
    // nächste Activity als shadow Instanz
    ShadowIntent shadowIntent = Robolectric.shadowOf(startedIntent);

    // Prüfen, ob die gestartete Activitx die richtige ist
    assertThat(shadowIntent.getComponent().getClassName(), equalTo(AnmerkungenHistorieActivity.class.getName()));

}
```

Abbildung 7.2-4 Test eines Button Robolectric

Wie schon erwähnt wird für das Durchlaufen der Tests kein Emulator oder externes Gerät benötigt. Die Testergebnisse werden wie in JUnit in rot oder grün angezeigt und die Debugausgabe erfolgt über die Console der Entwicklungsumgebung.

In dieser Bachelorarbeit war es jedoch nicht möglich die einzelnen Test als Testsuite ablaufen zu lassen. hierbei lief nur der erste Test erfolgreich durch, alle anderen wurden mit einem Fehler in der Datenbank beendet. Es wurde bemängelt, dass die entsprechend benötigte Tabelle der Datenbank nicht vorhanden ist. Wurden die Tests jedoch einzeln

durchlaufen, waren alle erfolgreich. Das heißt, Robolectric und die verwendete Datenbank *greenDAO* harmonieren nicht miteinander.

7.2.6 Bewertung

Robolectric bietet dem Entwickler eine einfache Art des Unitesting an. Zudem laufen die Tests schneller durch als bei anderen Frameworks, da Robolectric mit der JVM agiert. Mit Hilfe der Dokumentation ist es leicht sich in das Framework einzuarbeiten. Die Installation benötigt etwas mehr Zeit und vielleicht auch mehr als einen Versuch. Hat man aber erst einmal das Prinzip hinter den Shadow-Objekten verstanden, sind die Tests schnell und effizient zu implementieren. Der einzige negative Faktor ist, dass Robolectric nicht mit *greenDAO* funktioniert. Es müssen alle Tests einzeln per Hand ausgeführt werden, was ein hohes Maß an Zeit benötigt und somit auch keine aussagekräftige *Code Coverage* entstehen kann.

7.3 Robotium

7.3.1 Beschreibung

Robotium ist ein Testautomatisierungsframework, welches es dem Entwickler erleichtert User-Interface-Tests zu implementieren. Diese werden dann automatisch auf dem Emulator oder dem externen Gerät abgespielt. Es unterstützt native Anwendungen[10] und ab Version 4.0 auch hybride Anwendungen[11]. Testfälle werden aus der Sicht des Nutzers geschrieben, so dass technische Details oder Details zur Implementierung nicht benötigt werden. Die Klasse *Solo* wird verwendet, um das Testen der Activities zu erleichtern. Es ist ein speziell für Android entwickeltes Framework. (Google Project Hosting)

Wie schon erwähnt, gehört Robotium zu den Black-Box-Tests. Diese finden hauptsächllich im Funktionstest ihren Einsatz, da hier geprüft wird, ob die Spezifikation hinsichtlich der User-Interface- Gestaltung eingehalten wurden. Unter anderem wird somit aber auch festgestellt, ob die Software die vorher festgelegen Funktionen erfüllt.

[10] Native Anwendungen werden speziell für ein Betriebssystem programmiert und laufen dann ausschließlich auf Geräten, welche dieses Betriebssystem unterstützen.

[11] Hybride Anwendungen sind eine Mischung aus einer App, die speziell für ein mobiles Gerät(z.B. das Betriebssystem) entwickelt wurde und einer Webapplikation, die in jedem Browser läuft und damit plattformübergreifend fungiert.

Testphase x Testart	Funktions-test	Kontrollfluss-test	Softwaremes-sung	Stil-/ Codeanalyse
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	x			

Tabelle 7-4 Einordung Robotium in die Testphasen bzw. Testarten

7.3.2 Installation

Für die Installation wird ausschließlich die .jar Datei[12] benötigt. Diese Datei muss dann in das Testprojekt eingebunden werden. Die Komplexität der Installation wird daher als einfach eingestuft.

7.3.3 Einarbeitung

Die Einarbeitung in Robotium benötigt wenig Zeit. Auf der Website der Hersteller ist eine einleitende Dokumentation zu finden und durch die Community werden sämtliche Arten von möglichen Testfällen abgedeckt. Dadurch das man nur das User-Interface für den Test zur Verfügung hat und nicht noch den Sourcecode im Hintergrund, muss man sich nur im klarem darüber sein, was das gerade zu testende Interface als Sollfunktion aufweisen muss und testet diese dann auf einfachste Weise durch. Auch hier, wie schon bei Robolectric, baut die Struktur der Testklassen auf Junit auf. (Google Project Hosting)

Der Aufwand der Einarbeitung wird als niedrig bewertet, da die zahlreichen Dokumentationen einen schnellen Einstieg in Robotium bieten.

7.3.4 Anlegen eines Testprojektes

Um ein Testprojekt anzulegen muss zunächst einmal wieder das zu testende Android Application Project existieren. Nun legt man unter *File → New → Other* ein neues Android Test Project an, welches den Verweis auf das zu testende Android Application Project erhält. Anschließend muss noch der *BuildPath* unter *Configure Build Path* angepasst werden. Unter dem Reiter *Libaries* muss die vorher heruntergeladene .jar Datei von Robotium eingebunden werden und diese dann unter dem Reiter *Order und Export* noch ausgewählt werden. Nur so ist es möglich erfolgreiche Tests durchzuführen.

[12] http://code.google.com/p/robotium/downloads/list

Eine neue Testklasse wird angelegt, indem auf das Android Test Project ein Rechtsklick mit der Maus gemacht wird und nun *JUnit Test Case* ausgewählt wird. Zu beachten ist, dass unter Superclass wieder *android.test.ActivityInstrumentationTestCase2 ausgewählt* ist. Um die Testklassen auszuführen, klickt man auf die entsprechende Klasse unter *Run As* auf *Android JUnit Test*. (Google Project Hosting)

Der Aufand, welcher hinter dem Anlegen eines Testprojektes steckt, wird als niedrig bewertet. Die Hersteller, als auch die Community, bieten ausreichende Hilfestellungen an, welche Schritt für Schritt durch die Anleitung führen.

7.3.5 Testmöglichkeiten

Da Robotium das User-Interface testet und jeglichen Sourcecode im Hintergrund unbeachtet lässt, bietet unzählige Möglichkeiten das Verhalten des Interface zu testen. Es gibt aber natürlich den Weg, dass man teilweise auf den Code zugreift, wenn man ein Textfeld testen möchte, muss man sich erst einmal von der entsprechenden Activity holen. Dazu benötigt man die Information, wie das Textfeld in dem Layout benannt wurde.

Bevor überhaupt einer der Testmethoden erfolgreich durchlaufen kann muss ein Objekt der Klasse *Solo* erstellt werden, über welches dann auf alle zu testenden Elemente des Interface einer Activity zugegriffen werden kann.

```
// Soloklasse von Robotium, um UI-Elemente zu testen
private Solo solo;
// Anmerkung Activity
private AnmerkungActivity anmerkungActivity;
```

Abbildung 7.3-1 Anlegen des Solo-Objektes

Wie ist fast jedem auf JUnit basierendem Framework, muss auch hier die setUp() Methode implementiert werden.

```
/**
 * Die setUp() Methode wird vor dem start jeder Testmthode ausgeführt. In diesem Fall bekommt die
 * Klasse die BeutenID 20 übergeben und es wird die zu testende Activity geholt und dem Klassenobjekt
 * solo übergeben.
 */
protected void setUp() throws Exception {
    super.setUp();
    Config.getConfig().setBeutenId((long)20);

    anmerkungActivity = getActivity();
    solo = new Solo(getInstrumentation(), getActivity());
}
```

Abbildung 7.3-2 setUp() Methode

In dieser setUp() Methode bekommt das vorher generierte *solo-Objekt* der Klasse *Solo* die zu testende Activity *AnmerkungActivity* übergeben. Somit ist der Zugriff auf die User-Interface Elemente der jeweiligen Klasse möglich.

Nun ist es beispielsweise möglich ein EditText-Feld zu testen, ob Text richtig in das Feld geschrieben wird, ob das EditText-Feld überhaupt sichtbar ist auf der Oberfläche und ob der Text auch wirklich in dem Feld steht. Hier kommt nun das schon oben erwähnte Hinderniss, dass das Textfeld erst einmal aus der Activity geholt werden muss.

```
/**
 * Die Methode prüft das Textfeld in welches die Anmerkungen geschreiebn werden. Es wird überprüft, ob in das Textfeld
 * geschrieben werden kann, ob dieses auch wieder ausgelesen werden kann und ob es auf der Oberfläche überhaupt sichtbar ist.
 */
public void testTextfeld() {
    // Textfeld holen
    EditText text = (EditText) solo.getView(R.id.anmerkungTexte);
    // Text löschen, falls einer drin steht
    solo.clearEditText(text);
    // Text hineinschreiben
    solo.enterText(text, "gegen Varo behandelt");
    // Ist Textfeld auf Oberfläche sichtbar?
    assertEquals(View.VISIBLE, text.getVisibility());
    // Ist der der Text der drin steht, auch der der reingeschrieben wurde?
    assertEquals("gegen Varo behandelt", text.getText().toString());

}
```

Abbildung 7.3-3 EditText-Feld testen

Zudem ist es möglich einen Button auf seine Funktionalität hin zu prüfen. Bei *ImkerApp* soll der Button *Historie* zum Beispiel die entsprechende Activity öffnen, welche die Historie anzeigt. Hierbei muss man nicht erst den Button von der Activity holen, sowie es bei dem EditText-Feld der Fall war, sondern hier greift man über den Namen auf den Button, welchen er auch auf der Activity enthält.

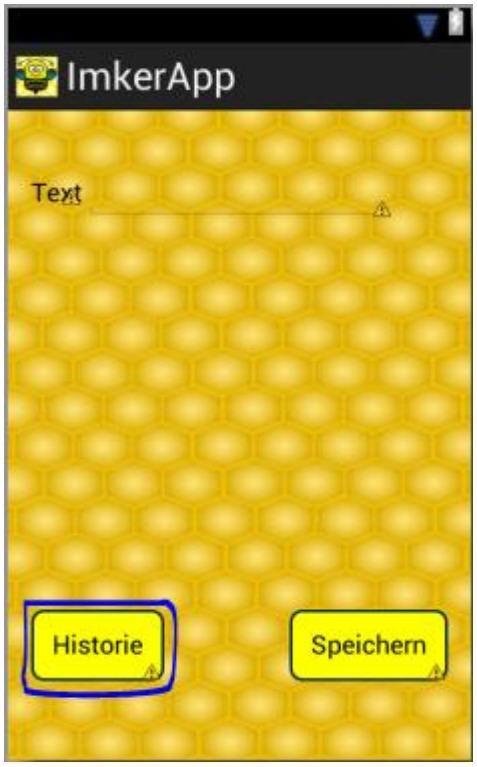


Abbildung 7.3-4 Activity Anmerkung

In Abbildung 7.3-4 ist die zu testenden Activity *AnmerkungActivity* zu sehen, auf welcher der gerade zu testende Button *Historie* blau umrandet ist. Der Button hat auf der Oberfläche ebenfalls den Namen *Historie*.

```
/**
 * Diese Methode prüft den Button Historie, ob dieser auch wirklich die nächste Activityy öffnet.
 */
public void testButtonHistorie() {

    //Klcik auf mit Text "Historie"
    solo.clickOnButton("Historie");

    // Warte auf nächste zu öffnende Activity
    solo.waitForActivity("AnmerkungHistorieActivity");
    // Prüfe, ob die geöffnete Activitx auch die richtige ist
    solo.assertCurrentActivity("AnmerkungHistorieActivity...", AnmerkungenHistorieActivity.class);

}
```

Abbildung 7.3-5 Button Historie Test

In der Abbildung 7.3-5 ist nun die Codezeile der Testmethode in blau unterstrichen wo auf den Button mit dem Namen Historie *geklickt* werden soll.

Eine andere Art des Testens eines Buttons ist im Falle der Testmethode für den Button *Speichern* zu sehen. Hier öffnet sich, nach das Speichen erfolgreich war, ein Dialogfenster mit dem Text *Ihre Daten wurden gespeichert!*. Es wird hier also überprüft, ob dieser Text auch auf der Oberfläche erscheint.

```
/**
 * Diese Methode prüft, ob der Button seine Funktionalität erfüllt. Nachdem ein Klcik auf dem Button ausgeführt wurde, wird ein
 * Dialogfenster geöffnet. Es wird also geprüft, ob der Text des Dialogfensters auch auf der oberfläche erscheint.
 */
public void testSpeichern() {

    // prüft, ob die richtige Activity ausgewählt ist
    solo.assertCurrentActivity("wrong activity", AnmerkungActivity.class);
    // klick auf den Button Speichern
    solo.clickOnButton("Speichern");
    // Prüfen, ob der erwünschte Text auf der Oberfläche erscheint
    assertTrue(solo.waitForText("Ihre Daten wurden gespeichert!"));

    }
```

Abbildung 7.3-6 Button Speichern Test

Auch hier wird wieder deutlich, dass der Button anhand seines Namens auf der Oberfläche ausgewählt wird.

Eine weitere wichtige Funktion zum Testen, welche Robotium anbietet, ist die *goBack()-Methode*. Es gibt Activities bei denen sich beim Starten zunächst die digitale Tastatur des Gerätes öffnet. Ist aber nun die erste Testaufforderungen einen Button auf der Oberfläche zu klicken "sieht" Robotium diesen nicht und gibt einen Fehllauf des Tests aus. Die *goBack()-Methode* verhindert dieses, in dem sie das Klicken des Zurückbuttons auf dem Gerät simuliert und somit die Tastatur schließt.

Am Ende jeder Testklasse sollte die *tearDown-Methode* implementiert sein, welche alle geöffneten Activities nach jedem Durchlauf einer Testmethode schließt. Ansonsten können ebenfalls Konflikte auftreten in der Hinsicht, dass Robotium auf einer falschen Activity testet.

```
/**
 * Schließt alle geöffenten Activities und beendet die setUp()-Methode.
 */
@Override
public void tearDown() throws Exception {

solo.finishOpenedActivities();
super.tearDown();

}
```

Abbildung 7.3-7 tearDown()-Methode

7.3.6 Bewertung

Robotium ist ein einfaches und schnell erlernbares Framework, mit dem es für jeden Entwickler möglich wird effiziente Oberflächentests zu implementieren. Die Webseite informiert über alle wichtigen Schritte, welche für das Arbeiten mit Robotium wichtig sind. Die Oberflächen an sich zu testen ist sehr einfach, da man dank der *Solo-Klasse* auf alle wichtigen Objekte zu greifen kann. Robotium ist ein Black-Box-Test, dennoch ist es nicht möglich ohne die Kenntnisse der *layout.xml* einer Activity auf die EditText-Felder zuzugreifen. Das ist zwar nicht direkt der Sourcecode, aber dennoch braucht man zum Testen mehr als nur die fertige App als *.apk-Datei*, welche auf einem externen Gerät oder dem Emulator läuft. Am Anfang ist es etwas ungewohnt, dass der Sourcecode im Hintergrund nicht mehr beachtet werden muss. Aber hat man erst einmal den ersten Test erfolgreich zu Ende gebracht, gehen alle anderen auch schnell von der Hand.

7.4 Metrics

7.4.1 Beschreibung

Metrics ist ein Framework, welches als Plugin für die Entwicklungsumgebung *eclipse* entwickelt wurde. Es funktioniert aber auch einwandfrei mit Android Projekten. Metrics kommt immer dann zum Einsatz, wenn in Java Projekten Projektkennzahlen ermittelt werden sollen. Zu diesen zählen unter Anderem die *Lines of Code* oder *Number of Classes*, aber auch die *Abstractness*, welche anzeigt, wie abstrakt die Software ist und daher wiederverwendbar. Die Ausgabe der Metriken kann entweder direkt in eclipse geschehen, aber auch als XML-Datei exportiert werden. (Walton) (FH Osnabrück)

Wie schon erwähnt, können mit Hilfe von metrics Projektkennzahlen erhoben, das heißt, es gehört zur Testart der Softwaremessung.

Testphase x Testart	Funktions-test	Kontrollfluss-test	Softwaremes-sung	Stil-/ Codeanalyse
Komponententest/ Modultest			x	
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	x			

Tabelle 7-5 Einordung metrics in die Testphasen bzw. TestartenInstallation

Das metrics-Plugin kann direkt über die Entwicklungsumgebung eclipse installiert werden. Unter dem Menüpunkt *Help → Install new Software..* gelangt man zum Installationsdialog. Im oberen Bereich auf der rechten Seite befindet der Button *Add* unter welchem der Installationspfad von metrics[13] eingegeben wird. Klickt man nun auf *Next* folgt man den weiteren Anweisungen der Installation und metrics wird in eclipse integriert.

Die Installation wird daher als einfach eingestuft, da lediglich das vorhandene Plugin installiert werden muss.

7.4.2 Einarbeitung

Die Einarbeitung von metrics ist dank der einfachen Handhabung und sehr übersichtlichen Ausgabe schnell gemeistert. Wer schon einmal mit Projektkennzahlen zu tun hatte wird schnell fündig werden. Für Anfänger bieten die Webseite[14] und zahlreiche Beiträge von Anhängern der Community einen schnellen Einstieg in den Umgang mit metrics, aber auch das Verstehen der Ausgabe wird dadurch erleichtert.

Die Komplexität der Einarbeitung wird als niedrig bewertet, da die Hersteller als auch die Community gut verständliche Hilfestellungen zur Verfügung stellen.

7.4.3 Anlegen eines Testprojektes

Nachdem das Plugin erfolgreich in eclipse integriert wurde, bedarf es nur noch einiger weniger Schritte, bis man die gewünschten Ausgaben erhält. Um metrics nun nutzen zu können, muss ein Rechtsklick auf das jeweilige Projekt gemacht werden und unter *Properties* der Menüpunkt *Metrics* ausgewählt werden. Hier setzt man nun das Häkchen bei *Enable Metrics.* Um die Projektkennzahlen anzeigen zu lassen geht man auf den Menüpunkt

[13] http://metrics.sourceforge.net/update.

[14] http://metrics.sourceforge.net

Window → Show View und wählt dort *Metrics View* aus. Erscheint nicht sofort die gewünschte Ansicht, kann es von Nöten sein, das jeweilige Projekt noch einmal zu *refreshen*.

Auch der Aufwand, welcher hinter dem Anlegen eines Testprojektes steht, wird mit niedrig bewertet, da ausreichende Dokumentationen von der Community bereit gestellt werden.

7.4.4 Testmöglichkeiten

Testmöglichkeiten in dem Sinne, wie sie bei den bisherigen Frameworks vorkamen, bietet metrics nicht an. Mit Hilfe der *Metrics View* in *eclipse* werden alle wichtigen Projektkennzahlen in einer Tabelle aufgelistet. Die einzelnen Metriken sind als Baumstruktur aufgelistet, welche man immer weiter aufsplitten kann, um zu sehen, mit welcher Gewichtung bestimmte Ressourcen in den Gesamtwert einfließen.

Afferent Coupling (avg/max per packageFragn		22	11,243	31	/ImkerApp/src/imkerapp/database	
Number of Interfaces (avg/max per packageFra	0	0	0	0	/ImkerApp/src/imkerapp/activities	
McCabe Cyclomatic Complexity (avg/max per		1,545	1,008	6	/ImkerApp/src/imkerapp/database/daos/BrutDao.java	bindValues
src		1,545	1,008	6	/ImkerApp/src/imkerapp/database/daos/BrutDao.java	bindValues
imkerapp.database.daos		2,015	1,334	6	/ImkerApp/src/imkerapp/database/daos/BrutDao.java	bindValues
BrutDao.java		2,278	1,693	6	/ImkerApp/src/imkerapp/database/daos/BrutDao.java	bindValues
BrutDao		2,278	1,693	6	/ImkerApp/src/imkerapp/database/daos/BrutDao.java	bindValues
bindValues	6					
readEntity	6					
loadAllDeepFromCursor	5					
loadDeep	4					
createTable	2					
dropTable	2					
readKey	2					
getKey	2					
getSelectDeep	2					
loadCurrentDeep	2					
BrutDao	1					
BrutDao	1					
attachEntity	1					
readEntity	1					
updateKeyAfterInsert	1					
isEntityUpdateable	1					
loadDeepAllAndCloseCursor	1					
queryDeep	1					
Properties		0	0			
StatusDao.java		2,278	1,693	6	/ImkerApp/src/imkerapp/database/daos/StatusDao.j...	bindValues
AnmerkungDao.java		1,944	1,129	5	/ImkerApp/src/imkerapp/database/daos/Anmerkung...	loadAllDeepFromCursor
BeuteDao.java		1,833	1,067	5	/ImkerApp/src/imkerapp/database/daos/BeuteDao.ja...	loadAllDeepFromCursor
KoeniginDao.java		1,833	1,067	5	/ImkerApp/src/imkerapp/database/daos/KoeniginDa...	loadAllDeepFromCursor
VersorgungDao.java		2,056	1,268	5	/ImkerApp/src/imkerapp/database/daos/Versorgung...	loadAllDeepFromCursor
StockbauDao.java		2,167	1,462	5	/ImkerApp/src/imkerapp/database/daos/StockbauDa...	bindValues
StandortDao.java		1,545	0,498	2	/ImkerApp/src/imkerapp/database/daos/StandortDa...	createTable
imkerapp.database.daoobjekte		1,331	0,711	4	/ImkerApp/src/imkerapp/database/daoobjekte/Stock...	getBeute
imkerapp.activities		1,415	0,685	3	/ImkerApp/src/imkerapp/activities/BrutActivity.java	onCreate
imkerapp.database		1,073	0,322	3	/ImkerApp/src/imkerapp/database/DatabaseManage...	ini
gen		0	0			
Total Lines of Code	3993					
Instability (avg/max per packageFragment)		0.31	0.356	1	/ImkerApp/src/imkerapp/activities	

Abbildung 7.4-1 Aufsplitten von Metriken

Alle blau angezeigten Zeilen in der Auswertung liegen mit ihren Werten im berechneten Bereich. Es gibt auch Zeilen, welche rot markiert sind. Hier liegt dann der Wert außerhalb des berechneten Bereichs.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Afferent Coupling (avg/max per packageFragn		22	11,243	31	/ImkerApp/src/imkerapp/database	
Number of Interfaces (avg/max per packageFra	0	0	0	0	/ImkerApp/src/imkerapp/activities	
McCabe Cyclomatic Complexity (avg/max per		1,545	1,008	6	/ImkerApp/src/imkerapp/database/daos/BrutDao.java	bindValues
Total Lines of Code	3993					
Instability (avg/max per packageFragment)		0,31	0,356	1	/ImkerApp/src/imkerapp/activities	
Number of Parameters (avg/max per method)		1,031	1,041	7	/ImkerApp/src/imkerapp/database/daoobjekte/Statu...	Status
src		1,031	1,041	7	/ImkerApp/src/imkerapp/database/daoobjekte/Statu...	Status
imkerapp.database.daoobjekte		0,703	1,238	7	/ImkerApp/src/imkerapp/database/daoobjekte/Statu...	Status
Status.java		0,739	1,421	7	/ImkerApp/src/imkerapp/database/daoobjekte/Statu...	Status
Status		0,739	1,421	7	/ImkerApp/src/imkerapp/database/daoobjekte/Statu...	Status
Status	7					
Status	1					
__setDaoSession	1					
setId	1					
setDate	1					
setHonigleistung	1					
setFutterwaben	1					
setWabensitz	1					
setSanftmut	1					
setBeutenId	1					
setBeute	1					
Status	0					
getId	0					
getDate	0					
getHonigleistung	0					
getFutterwaben	0					
getWabensitz	0					
getSanftmut	0					
getBeutenId	0					
getBeute	0					
delete	0					
update	0					
refresh	0					
Koenigin.java		0,739	1,421	7	/ImkerApp/src/imkerapp/database/daoobjekte/Koeni...	Koenigin
Brut.java		0,739	1,421	7	/ImkerApp/src/imkerapp/database/daoobjekte/Brut.j...	Brut
Stockbau.java		0,714	1,278	6	/ImkerApp/src/imkerapp/database/daoobjekte/Stock...	Stockbau
Versorgung.java		0,684	1,126	5	/ImkerApp/src/imkerapp/database/daoobjekte/Verso...	Versorgung

Abbildung 7.4-2 Rot gekennzeichnete Zeile in Metrics View

In Abbildung 7.4-2 ist nun zu sehen, dass die Anzahl der Parameter in der Klasse *Status.java* 7 beträgt. Maximal erlaubt sind aber nur 5 Parameter. Neben der *Metrics View* bietet mertics noch die *Dependency Graph View* an, welche eine grafische Übersicht über die Zusammenhänge der einzelnen Pakete enthält.

Abbildung 7.4-3 Dependency Graph ImkerApp

In dieser Ansicht ist es möglich, weiter in die Struktur hinein zu zoomen und diese auch zu drehen. Die roten und blauen Kästchen zeigen die einzelnen Pakete an. Wird ein Paket rot angezeigt und man zoomt weiter in dieses Paket hinein, sieht man, dass es zu viele Abhängigkeiten zu anderen Paketen hat.

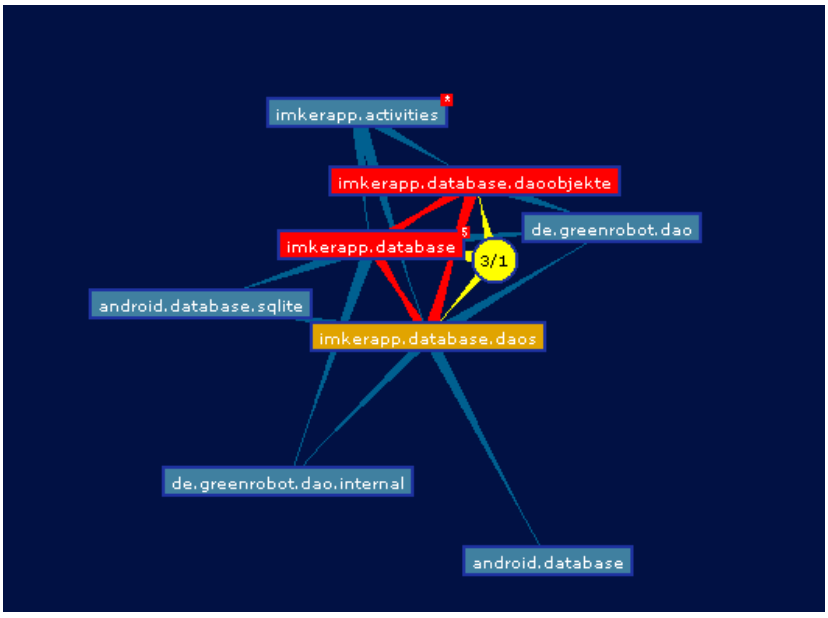


Abbildung 7.4-4 rotes Paket

Die Zahlen in dem kleinen gelben Kästchen zeigen an, wie viele Pakete in dem Gewirr von Abhängigkeiten beteiligt sind(3) und den längsten Weg zwischen den Paketen(1). Der Entwickler sollte nun im besten Fall versuchen diese Abhängigkeiten aufzulösen. Im Falle von *ImkerApp* war dieses leider nicht möglich, da die rot angezeigten Pakete zur generierten Datenhaltung von *greenDao* gehören.

7.4.5 Bewertung

Metrics ist ein einfaches Framework welches leicht zu bedienen ist. Es zeigt dem Entwickler auf eine sehr übersichtliche Art und Weise genau die Projektkennzahlen an, welche zum Beispiel im Sinne einer Statistik benötigt werden. Dank der Funktion des Exports in XML, können diese auch sofort in die Projektdokumentation eingefügt werden. Eine lange Einarbeitungszeit ist ebenfalls nicht von Nöten. Dank der detaillierten Dokumentation auf der Website der Entwickler, wird jedem ein schneller Umgang mit metrics ermöglicht.

7.5 Traceview

7.5.1 Beschreibung

Traceview ist ein grafisches Programm, welches zur Ausführung von sogenannten Logdateien[15] geeignet ist, die in der jeweiligen zu testenden Anwendung gespeichert werden. Es kann dem Entwickler helfen seine Applikation zu debuggen und die Performance zu verbessern. Es ist ein sehr starkes Tool und in der Applikationsentwicklung mit Android weit verbreitet. (android developer traceview)

Wie schon erwähnt ist es mit traceview möglich, die Performance einer Applikation zu messen. Demzufolge gehört dieses Framework in die Kategorie des Leistungstests.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**			x	

Tabelle 7-6 Einordung traceview in die Testphasen bzw. Testarten

[15] Diese Dateien speichern Prozesse und Datenänderungen, welche beim Durchlaufen einer Application anfallen.

7.5.2 Installation

Traceview ist bereits im Android SDK[16] enthalten.

Die Komplexität der Installation wird als einfach bewertet, da der Entwickler keinen weiteren Aufwand hat, wenn das Android SDK bereits installiert ist.

7.5.3 Einarbeitung

Traceview benötigt eine etwas längere Einarbeitungszeit, da der Entwickler erst einmal die Ausgabe der Ergebnisse analysieren muss. Auf der Webseite des Herstellers ist hierfür ein einführendes Beispiel gegeben, in welchem gut verständlich die einzelnen Ausgaben auf der Oberfläche dargestellt werden. Hat der Entwickler die Ausgabe erst einmal verinnerlicht,lassen sich Probleme, wie zum Beispiel zu häufig durchlaufene Schleifen, leicht finden.

Der Aufwand der Einarbeitung wird als mittel eingestuft, da der Entwickler eine gute Einleitung des Herstellers erhält, aber trotzdem nach weiteren Details in der Community recherchieren muss, um die Materie von traceview vollkommen zu verstehen.

7.5.4 Anlegen eines Testprojekts

Um die graphische Darstellung der Ausgabe von traceview in der Entwicklungsumgebung darzustellen muss zunächst unter dem menüpunkt *Window → Open Perspective → Other* die *DDMS* Perspektive ausgewählt werden. Nun schließt man ein externes Gerät an auf welchem die zu testende Application installiert ist und schon kann der Test mit traceview beginnen.

Die Komplexität wird als niedrig bewertet, da der Hersteller und auch die Community ausreichende und gut nachvollziehbare Hilfestellungen zur Verfügung stellen.

7.5.5 Testmöglichkeiten

Wurden die Logdateien der Applikation mit Hilfe von DDMS analysiert, erscheinen zwei verschiedene Ausgaben, welche zur Auswertung genutzt werden können. Zum einen gibt es das sogenannte *Timeline Panel.* In diesem werden, abhängig von der Zeit, die jeweiligen Threads und Methoden angezeigt und wann diese jeweils gestartet beziehungsweise gestoppt werden.

[16] http://developer.android.com/sdk/index.html

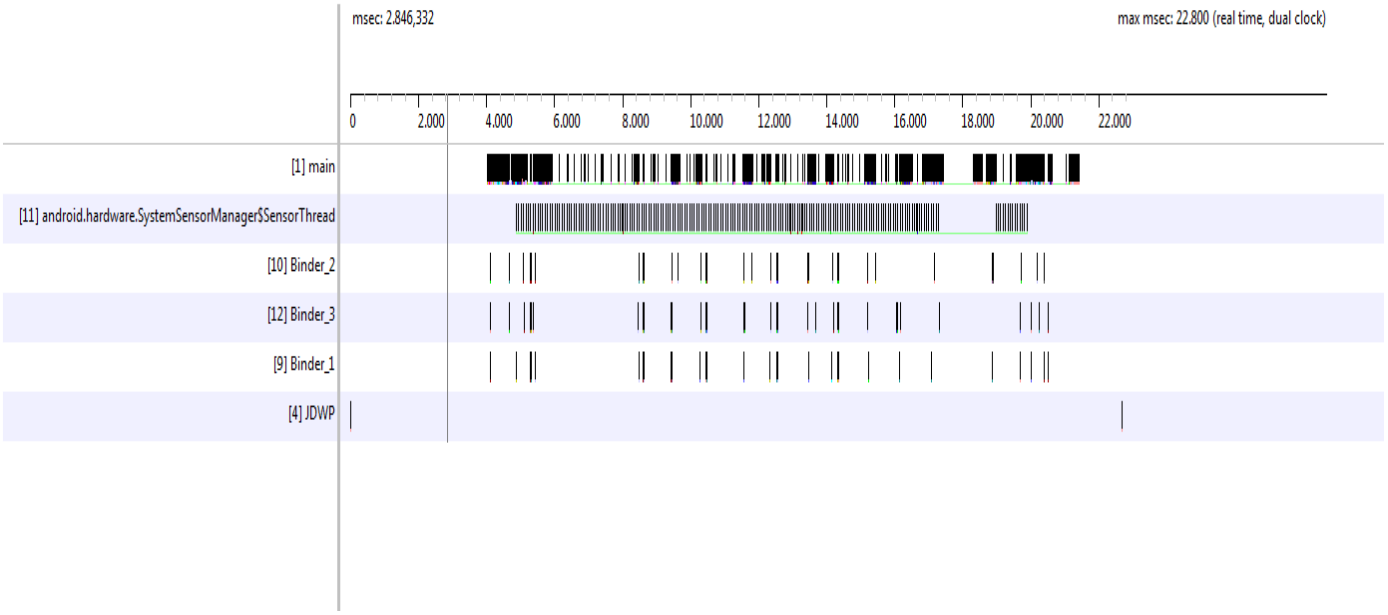


Abbildung 7.5-1 Timeline Panel ImkerApp

In der Abbildung 7.5-11 sind die einzelne Threads zu sehen, jeder in seiner eigenen Zeile mit der Zeitleiste, welche nach rechts hin größer wird. Die Zeit wird hier in Millisekunden angezeigt. Jede Methode, welche in einem Thread aufgerufen wird, erhält eine andere Farbe. Die Farben werden anhand eines Round-Robin-Prinzips[17] vergeben, beginnend bei der Methode, welche die meiste Zeit in Anspruch nimmt. Die dünnen Linien unter der ersten Zeile, welche dem main-Thread zugewiesen ist, zeigen den Umfang (vom starten bis zum beenden) aller Aufrufe einer ausgewählten Methode an. Geht man nun mit der Maus über die dünnen Linien, wird angezeigt, in welcher Methode sich die Application zu dem Zeitpunkt befand. So kann man beispielsweise herausfinden, wenn sich die Application sehr lange in ein und derselben Methode aufhält. Als Konsequenz daraus kann der Entwickler nun überlegen, wie er dieses Problem beheben kann, damit die Abfolge der Funktionen performanter wird. Auswählen kann man die entsprechenden Methoden in dem *Profile Panel*, welches zugleich die zweite Möglichkeit der Auswertung darstellt.

[17] Das Round-Robin-Verfahren gewährt allen Methoden nacheinander für jeweils einen kurzen Zeitraum Zugang zu den benötigten Ressourcen.

Name	Incl Cpu Time %	Incl Cpu Time	Excl Cpu Time %	Excl Cpu Time	Incl Real Time %	Incl Real Time	Excl Real Time %	Excl Real Time	Calls+RecurCa...	Cpu Time/Call	Real Time/Call
0 (toplevel)	100,0%	4827,637	0,6%	30,751	100,0%	104172,634	0,0%	0,000	12+0	402,303	8681,053
1 android/os/Handler.dispatchMessage (Landroid/os/Message;'	87,1%	4202,882	0,1%	6,012	5,1%	5271,241	0,0%	5,646	314+0	13,385	16,787
2 android/os/Handler.handleCallback (Landroid/os/Message;)V	67,3%	3248,935	0,0%	2,046	4,0%	4123,474	0,0%	2,255	129+0	25,186	31,965
3 android/view/Choreographer$FrameDisplayEventReceiver.run	60,3%	2910,064	0,0%	1,680	3,5%	3687,958	0,0%	1,587	84+0	34,644	43,904
4 android/view/Choreographer.doFrame (JI)V	60,2%	2908,384	0,1%	3,602	3,5%	3686,371	0,0%	3,716	84+0	34,624	43,885
5 android/view/Choreographer.doCallbacks (II)V	60,2%	2904,019	0,2%	7,682	3,5%	3680,734	0,0%	7,478	252+0	11,524	14,606
6 android/view/Choreographer$CallbackRecord.run (J)V	59,8%	2886,781	0,1%	4,795	3,5%	3662,720	0,0%	4,795	175+0	16,496	20,930
7 android/view/ViewRootImpl$TraversalRunnable.run ()V	58,5%	2824,093	0,0%	1,371	3,5%	3598,540	0,0%	1,221	84+0	33,620	42,840
8 android/view/ViewRootImpl.doTraversal ()V	58,5%	2822,722	0,1%	3,174	3,5%	3597,319	0,0%	3,715	84+0	33,604	42,825
9 android/view/ViewRootImpl.performTraversals ()V	58,2%	2807,703	0,2%	10,808	3,4%	3582,091	0,0%	10,708	84+0	33,425	42,644
10 android/view/ViewRootImpl.performDraw ()V	24,6%	1188,444	0,0%	2,316	1,6%	1645,263	0,0%	2,135	72+0	16,506	22,851
11 android/view/ViewRootImpl.draw (Z)V	24,5%	1181,885	0,1%	4,916	1,6%	1622,314	0,0%	5,496	72+0	16,415	22,532
12 android/view/HardwareRenderer$GlRenderer.draw (Landroid	23,5%	1134,728	0,3%	14,397	1,5%	1569,057	0,0%	13,698	70+0	16,210	22,415
13 android/view/ViewRootImpl.performMeasure (II)V	22,1%	1068,055	0,0%	0,367	1,1%	1149,018	0,0%	0,518	15+0	71,204	76,601
14 android/view/View.measure (II)V	22,1%	1067,933	0,2%	8,392	1,1%	1148,774	0,0%	8,102	17+466	2,211	2,378
15 com/android/internal/policy/impl/PhoneWindow$DecorVie	22,1%	1067,017	0,0%	1,008	1,1%	1147,889	0,0%	1,038	15+0	71,134	76,526
16 android/widget/FrameLayout.onMeasure (II)V	22,0%	1064,116	0,2%	8,187	1,1%	1142,245	0,0%	8,482	18+50	15,649	16,798
17 android/view/ViewGroup.measureChildWithMargins (Landro	21,8%	1053,926	0,1%	5,101	1,1%	1131,989	0,0%	5,049	18+140	6,670	7,164
18 android/widget/LinearLayout.onMeasure (II)V	21,6%	1041,627	0,0%	0,980	1,1%	1118,651	0,0%	1,069	18+50	15,318	16,451
19 android/widget/LinearLayout.measureVertical (II)V	21,6%	1041,412	0,2%	9,402	1,1%	1118,468	0,0%	9,001	18+32	20,828	22,369
20 android/view/ViewRootImpl.measureHierarchy (Landroid/vie	19,2%	925,902	0,0%	0,456	1,0%	1003,297	0,0%	0,489	12+0	77,159	83,608
21 android/view/View.getDisplayList ()Landroid/view/DisplayLis	18,2%	879,857	0,3%	14,045	1,0%	1091,430	0,0%	15,161	70+779	1,036	1,286

Abbildung 7.5-2 Ausschnitt Profile Panel ImkerApp

In Abbildung 7.5-22 ist zu erkennen, dass hier sämtliche Methoden der zu testenden Application angezeigt werden. Hier wird auch deutlich, welche Farbe im *Timeline Panel* zu welcher Methode gehört. Zudem wird in dieser Tabelle angezeigt, wie viel genau in den einzelnen Methoden verbracht wurde und zwar die inklusive und exklusive Zeit, als auch die Gesamtzeit. Exklusive Zeit zeigt die Zeit an, wie lange sich in der Methode aufgehalten wurde. Die inklusive Zeit hingegen ist die Zeit die in der Methode verbracht wurde plus die Zeit, welche in jeglichem Aufruf von Funktionen verbracht wurde. Klickt man auf der angezeigten Methoden wird diese weiter aufgespalten und zwar in *Parents* und *Children.*

Name	Incl Cpu Time %	Incl Cpu Time	Excl Cpu Time %	Excl Cpu Time	Incl Real Time %	Incl Real Time	Excl Real Time %	Excl Real Time	Calls+RecurCa...
6 android/view/Choreographer$CallbackRecord.run (J)V	61,2%	1616,636	0,1%	1,491	3,9%	2170,261	0,0%	1,344	103+0
Parents									
5 android/view/Choreographer.doCallbacks (II)V	100,0%	1616,636			100,0%	2170,261			103/103
Children									
self	0,1%	1,491			0,1%	1,344			
7 android/view/ViewRootImpl$TraversalRunnable.run ()\	99,4%	1606,779			99,5%	2160,277			50/50
487 android/view/ViewRootImpl$ConsumeBatchedInput	0,5%	7,874			0,4%	7,936			52/52
1847 android/view/Choreographer.access$400 ()Ljava/la	0,0%	0,370			0,0%	0,582			103/103
2466 android/view/ViewRootImpl$InvalidateOnAnimatio	0,0%	0,122			0,0%	0,122			1/1
7 android/view/ViewRootImpl$TraversalRunnable.run ()V	60,8%	1606,779	0,0%	0,337	3,9%	2160,277	0,0%	0,457	50+0

Abbildung 7.5-3 Aufspalten einer Methode

Die Methode welche unter dem Punkt *Parents* eingeordnet wurde, ist die Methode, in welcher die ausgewählte Methode aufgerufen wird. Die Methoden unter *Children* sind die Methoden, welche die ausgewählte Methode selber aufruft. In dem in Abbildung 7.5-3 gezeigten *Profile Panel* sind in der letzen Spalten die Anzahl der Aufrufe und die Anzahl der rekursiven Aufrufe der Methoden zu erkennen. (android developer traceview)

7.5.6 Bewertung

Traceview ist im Allgemeinen ein sehr schönes Tool, um Performanceprobleme in der Applikationsentwicklung aufzudecken. Die Einarbeitung und das Verstehen der verschiedenen Ausgabemöglichkeiten benötigt eine gewisse Zeit. Und selbst dann ist es manchmal schwierig das Performanceproblem zu finden. Dennoch erleichtert es die Arbeit des Entwicklers um einiges. Gerade das *Timeline Panel* trägt einen großen Beitrag zur Findung des Problems bei. Denn hier ist schon nach kurzer Analyse der dünnen Linien klar, welche Methode die eventuelle Verzögerung verursacht. Ist das Problem gefunden, muss der Entwickler sich eine Strategie überlegen, um jenes zu überarbeiten.

7.6 monkey

7.6.1 Beschreibung

Monkey ist ein Stresstest für Android Applikationen. Mit einem einzigen Befehl in der Konsole wird ein Strom von Nutzerevents, wie zum Beispiel Klicks oder Touchgesten, auf dem externen Gerät oder Emulator simuliert. Wenn monkey läuft generiert es Ereignisse und sendet diese an das System. Darüber hinaus beobachtet es die zu testende Application und sucht nach drei Bedingungen, welche besondere Beachtung erfahren:

- Hat man dem monkey den Befehl gegeben in einem oder mehreren Paketen zu laufen, wird es nach einem Weg suchen in das andere Pakete zu navigieren und diesen dann zu blockieren.
- Stürzt die Application ab oder es tritt ein anderer Fehler auf, wird monkey die Application stoppen und den Fehler im LogCat ausgeben.
- Ebenfalls wird ein Fehler ausgegeben, wenn die zu testende Application einen *Application antwortet nicht* Fehler generiert.

Je nachdem wie ausführlich die Befehle in der Konsole gesetzt werden, können Berichte über den Fortschritt von monkey angezeigt werden und welche Ereignisse gerade bearbeitet werden. (android developer monkey)

Wie schon erwähnt gehört monkey zu der Testphase Stresstest.

Testphase x Testart	Funktions-test	Kontrollfluss-test	Softwaremes-sung	Stil-/ Codeanalyse
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	x			

Tabelle 7-7 Einordung monkey in die Testphasen bzw. Testarten

7.6.2 Installation

Monkey ist bereits im Android SDK[18] enthalten.

Der Aufwand der Installation wird daher als einfach eingestuft.

7.6.3 Einarbeitung

Die Einarbeitungszeit in monkey ist schnell abgeschlossen. Auf der Webseite des Herstellers sind alle Befehle, welche über die Konsole genutzt werden können, erläutert. Somit sind die richtigen Befehle für die gewünschte Ausgabe schnell gefunden.

Die Komplexität der Einarbeitung wird als niedrig bewertet, da der Hersteller alle wichtigen Befehle übersichtlich zusammengefasst hat.

7.6.4 Anlegen eines Testprojekts

In der Konsole müssen lediglich folgende Befehle eingegeben werden, um monkey zu starten:

Zunächst muss in den Ordner navigiert werden, in welchem die *adb.exe* Datei zu finden ist. Diese liegt meist im /platform-tools Ordner des Android SDK. Nun muss der Befehl *adb start-server* eingegeben werden. Als nächstes muss entweder der Emulator gestartet werden oder ein externes Gerät angeschlossen werden. Um zu prüfen, ob das angeschlossene Gerät verfügbar ist, tippt man den Befehl *adb devices* ein. War dieses erfolgreich erhält man eine Auflistung der gefundenen Geräte. Durch *adb shell* greift man auf die Shell des Geräts zu.

[18] http://developer.android.com/tools/help/monkey.html

Um den Stresstest zu starten gibt man *monkey -p Name der Application auf dem Gerät Anzahl Tests* ein und erhält ein Feedback, wie lange die Tests gedauert haben und wie lange eine Verbindung zu einem Wifi bestand.

Das Anlegen des Testprojektes ist auf der Webseite des Herstellers und in der Community einfach und schnell verständlich beschrieben worden, so dass der Aufwand als niedrig eingestuft wird.

7.6.5 Testmöglichkeiten

Neben der Eingabe des Testprojektes und der Anzahl der Tests, welche zwischen 500 und 1000 liegen können, gibt es noch weitere Eingabemöglichkeiten für monkey.

Ein sehr interessanter Befehl ist *-v*. Es wird auf der Konsole eine detaillierte Ausgabe über den Start, Testabschluss und Endergebnisse geliefert. Es gibt drei Stufen *(-v, -vv, -vvv)*, in welchen die Ausführlichkeit der Ausgabe immer mehr vertieft wird.

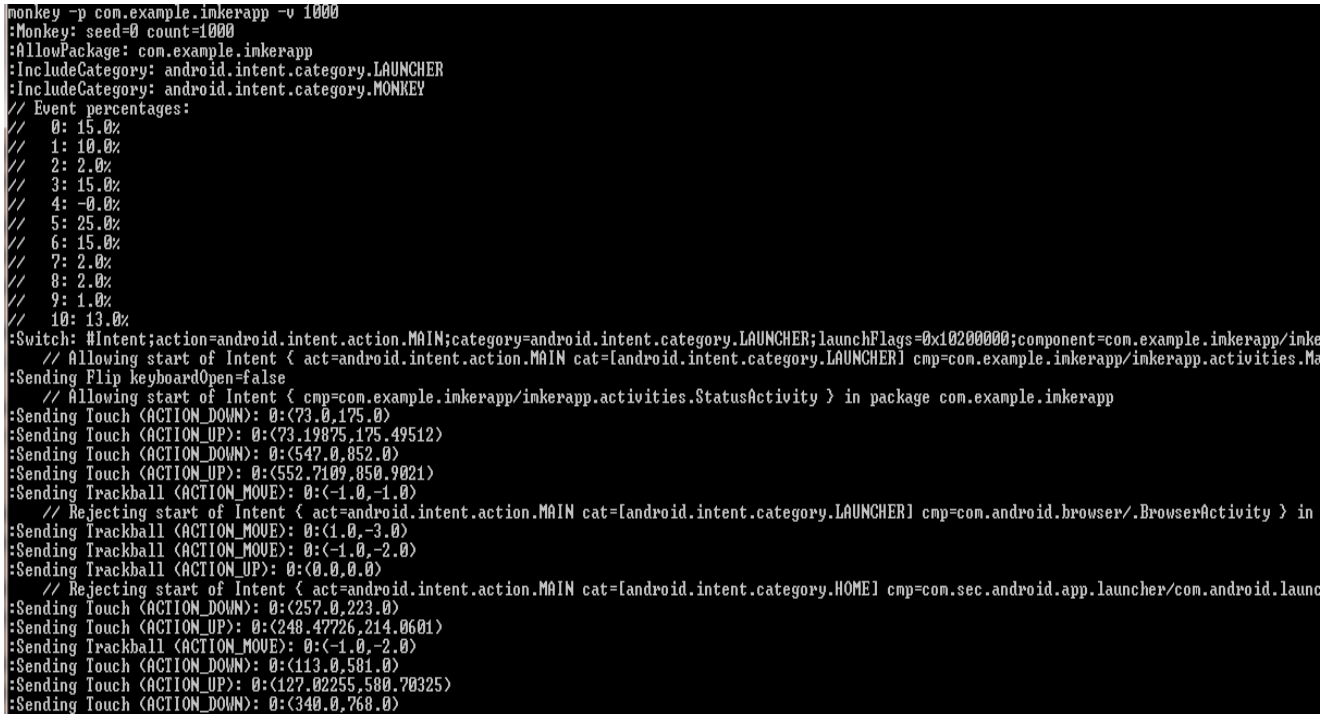


Abbildung 7.6-1 Ausgabe monkey mit Befehl -v 1000

In der Abbildung 7.6-1 ist nun die Ausgabe eines Tests zu sehen, welcher 1000 Nutzerevents simuliert hat und als zusätzlichen Befehl *-v*. Hier ist nun zu erkennen, wo genau ein Touchevent ausgelöst wurde und wann ein anderer Intent geöffnet wurde.

Ein weiterer interessanter Befehl ist *-s.* Sein Aufruf in der Konsole lautet für ImkerApp wie folgt: *monkey -p com.example.imkerapp -s 3 500*. Dem Wert -s wurde hier die *3* zugewiesen. Möchte man nun, dass monkey genau diesen Test noch einmal ausführt, gibt man wieder *-s 3* an und es werden genau dieselben Ereignisse noch einmal durchlaufen.

Wird der Befehl *-throttle* eingegeben, gibt eine Verzögerung zwischen den Ereignissen. Dieses ist sehr von Vorteil, wenn zum Beispiel nach einem Klickevent erst noch Daten geladen werden. Zudem können noch verschiedene Prozentzahlen, zum Beispiel für die Betätigung der Systemtasten, ausgegeben werden. Diese Prozentzahlen zeigen an, wie lange von der gesamten Zeit des Tests mit den dementsprechenden Events interagiert wurde. Eine weitere sehr hilfreiche Reihe von Befehlen sind die für das Debugging. Hier können dann beispielsweise Abstürze der Applikation oder Sicherheitsfehler einfach unbeachtet bleiben. Also Warnungen, bei denen monkey normalerweise den Test abbricht und eine Fehlermeldung ausgibt.

7.6.6 Bewertung

monkey ist ein hilfreiches Werkzeug, wenn es darum geht seine Application am Verhalten eines Nutzers zu testen. Der Entwickler hat keinerlei Einfluss darauf, wann und welche Events auf der Application ausgeführt werden. So, als würde ein Endnutzer die Application in der Hand halten. Zudem werden die Events sehr schnell hintereinander ausgeführt, so dass gleichzeitig geprüft wird, wie die Application mit schnellen Abfolgen zurechtkommt. Für die Auswertung am Ende des Tests, gibt es entsprechende Einstellungsmöglichkeiten, je nachdem wie detailliert der Entwickler das Endergebnis evaluiert haben möchte. Wer allerdings noch nie mit der Konsole gearbeitet hat muss sich zunächst an die Eingabe der einzelnen Befehle und auch der späteren Ausgabe gewöhnen. Diese ist bei vielen Informationen recht unübersichtlich und bietet keine Möglichkeit an, die Informationen in ein anderes Format zu exportieren. Da kann es dann schon mal schwierig werden die gesuchten Informationen auch zu finden. Hier wäre eine grafische Ausgabe in der Entwicklungsumgebung wünschenswerter gewesen.

7.7 Android Lint

7.7.1 Beschreibung

Android Lint ist ein statisches Code-Analyse Tool. Es durchsucht den Sourcecode des jeweiligen Android Projects nach möglichen Bugs und Optimierungsverbesserungen für die Korrektheit, Sicherheit, Leistung, Benutzerfreundlichkeit, Zugänglichkeit und Korrektheit. Im folgenden ist eine Liste von möglichen Fehlern, welche Android Lint erkennen kann:

- nicht übersetzte Strings
- hartkodierte Strings im Quelltext außerhalb der strings.xml
- invalides Layout
- unbenutzte Ressourcen allgemein wie Bilder etc.
- Accessibility Probleme: fehlende Bildbeschreibungen
- unperformante Layout-Konstruktionen

- fehlende Bilder in bestimmten Auflösungen (hdpi, mdpi, ldpi)

Wie schon erwähnt gehört Android Lint zu den Code-Analyse Tools.

(android developer Android lint)

Testphase x Testart	Funktions-test	Kontrollfluss-test	Softwaremes-sung	Stil-/ Codeanalyse
Komponententest/ Modultest				x
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 7-8 Einordung Android Lint in die Testphasen bzw. Testarten

7.7.2 Installation

Android Lint ist bereits im Android SDK[19] enthalten.

Der Entwickler hat keinen weiteren Aufwand mit der Installation, daher wird dieser Vorgang als einfach bewertet.

7.7.3 Einarbeitung

Für Android Lint wird kaum Zeit für eine Einarbeitung benötigt. Der Start über die Konsole als auch über die Entwicklungsumgebung sind auf der Webseite der Entwickler ausreichend und übersichtlich dokumentiert. Auch die Ausgabe der gefundenen Bugs und Verbesserungsvorschläge ist leicht zu verstehen.

Auch hier wird der Aufwand als niedrig eingestuft, da der Hersteller eine kurze, aber aussagekräftige Dokumentation anbietet.

7.7.4 Anlegen eines Testprojektes

Um Android Lint in der Entwicklungsumgebung nutzen zu können, muss lediglich unter *Window → Show View → Oth*er die *Lint Warnings* View ausgewählt werden. Klickt man nun auf das gewünschte Android Project erscheint eine tabellarische Auflistung der gefundenen Bugs und Verbesserungen.

[19] http://developer.android.com/tools/help/lint.html

Das Anlegen des Testprojektes erfordert keinen weiteren Aufwand von Seiten des Entwicklers und wird somit als niedrig eingestuft.

Die Möglichkeit Android Lint über die Konsole zu starten wurde in dieser Bachelorarbeit nicht in Betracht gezogen, da eine stetige Fehlerausgabe während der Entwicklung geschehen sollte und die Konsole dafür als ungeeignet erschien.

7.7.5 Testmöglichkeiten

Die *Lint Warnings* View zeigt nun sämtliche Bugs und Verbesserungen an, welche bereits im Kapitel 7.7.1 erwähnt wurden.

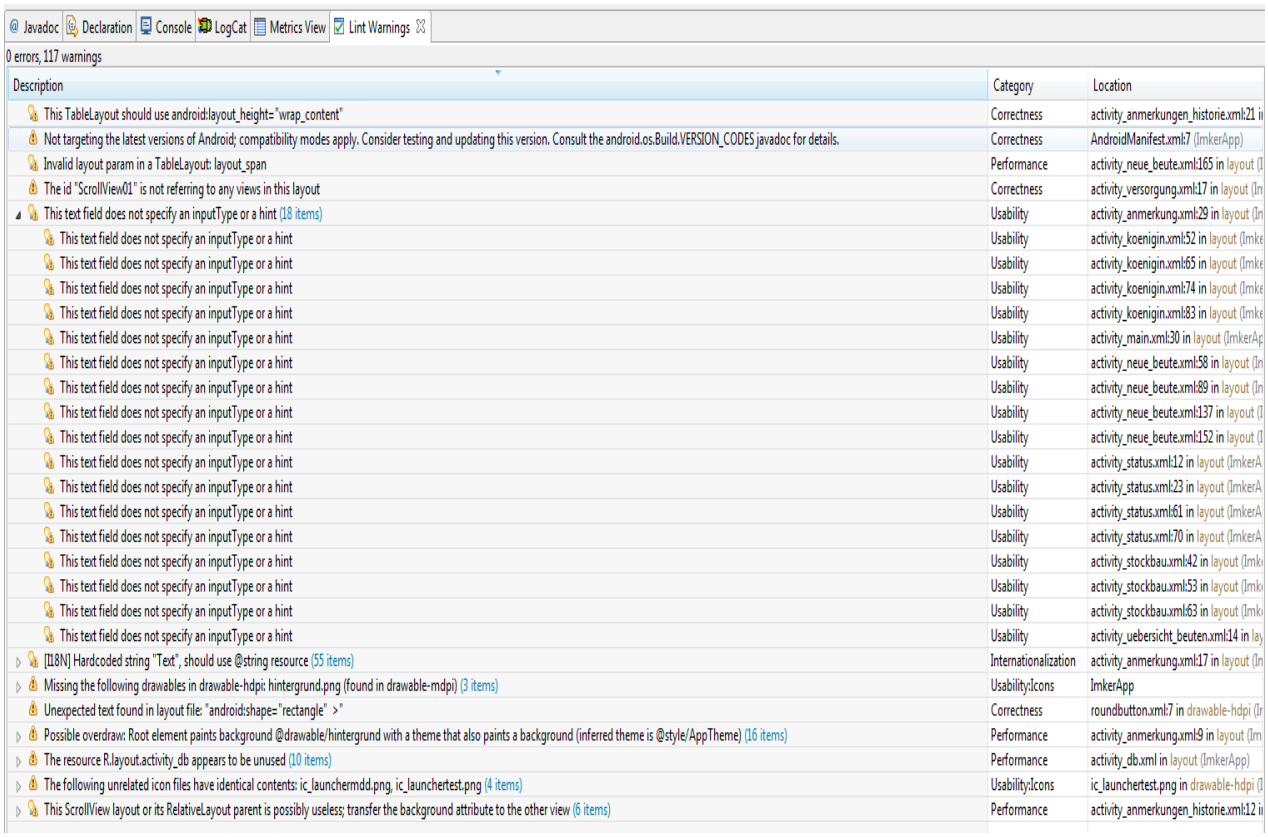


Abbildung 7.7-1 Lint Warnings View ImkerApp

In Abbildung 7.7-1 ist die *Lint Warnings View* von *ImkerApp* zu sehen. Tritt eine Warnung mehrmals auf, wird eine Gesamtüberschrift gebildet unter welcher alle Codezeilen aufgelistet werden, die dieser Warnung entsprechen. Klickt man nun auf eine dieser Warnung gelangt man direkt zu der Codezeile in welcher sich die Warnung befindet und kann diese dann dort verbessern. Ist die Warnung behoben, aktualisiert sich die *Lint Warnings* View automatisch. Es wird aber nicht nur die Beschreibung und die entsprechende Codezeile der Warnung angezeigt, sondern auch noch zu welcher Kategorie die Warnung gehört. Diese wurden ebenfalls schon im Kapitel 7.7.1 erwähnt.

7.7.6 Bewertung

Android Lint ist ein schönes und einfaches Tool zur schnellen Erkennung und Behebung von Optimierungsverbesserungen und Bugs im Sourcecode. Es ist einfach zu bedienen und bietet eine übersichtliche Ausgabe der gefunden Warnungen an. Es bleibt dann dem Entwickler selber überlassen, welche er davon beheben möchte. Hierzu tragen sicher auch die Kategorien bei, in welche die Warnungen eingeteilt werden. Möchte der Entwickler die Performance erhöhen, ist ihm sicher gut geraten erst einmal die Warnungen aus der Kategorie Performance zu korrigieren. Von Vorteil ist auch, dass Android Lint über den ganzen Zeitraum der Entwicklung genutzt werden kann und sich automatisch aktualisiert, sobald eine Warnung entfernt wurde. So können Bugs und Optimierungsprobleme gar nicht erst auftreten.

7.8 Android Findbugs

7.8.1 Beschreibung

Android Findbugs ist ein Code-Analyse Tool, welches auf dem ursprünglichen Findbugs-Plugin von eclipse basiert. Die in dieser Bachelorarbeit verwendete Version ist in der Lage, Fehler auch in Android-spezifischem Quellcode zu finden. Im Gegensatz zu im Kapitel 7.7 beschriebenen Android Lint, ist Android Findbugs nur in der Lage Fehler im Java-basiertem Quellcode zu finden und nicht in den Layout-basierten .xml-Dateien.

Wie schon erwähnt ist Android Findbugs ein Code-Analyse Tool.

(Google Project Hosting Android findbugs) (Vogel, 2013)

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				x
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 7-9 Einordung Android Findbugs in die Testphasen bzw. Testarten

7.8.2 Installation

Um Android Findbugs in die Entwicklungsumgebung einzubinden muss diese unter *Help* → *Install New Software* installiert werden. Unter dem Button *Add* fügt man nun den Link[20] für

[20] http://findbugs.cs.umd.edu/eclipse

die Installation hinzu und folgt den weiteren Schritten. Wurde die Entwicklungsumgebung nach erfolgreicher Installation neu gestartet, kann Android Findbugs verwendet werden.

Da der Entwickler ein Plugin installieren muss, um Android Findbugs zu verwenden, wird der Aufwand als einfach bewertet.

7.8.3 Einarbeitung

Es ist nur eine kurze Einarbeitungszeit von Nöten. Auf der Webseite des Herstellers ist eine kurze Anleitung zur Erklärung zu finden. Die Ausgabe der gefundenen Fehler erfolgt in einer übersichtlichen tabellarischen Zusammenstellung, welche dank der kurzen Fehlerbeschreibung leicht zu verstehen ist.

Die Komplexität der Einarbeitung wird daher als niedrig eingestuft.

7.8.4 Anlegen eines Testprojektes

Wurde Android Findbugs erfolgreich in der Entwicklungsumgebung installiert müssen lediglich noch die entsprechenden Views zur Anzeige der gefundenen Fehler aktiviert werden. Dies geschieht über *Window → Show View → Other → Findbugs.* Klickt man nun auf das gewünschte Android Project erscheint eine tabellarische Auflistung der gefundenen Bugs und Verbesserungen.

Der Aufwand, welche hinter dem Anlegen eines Testprojektes steht, wird als niedrig bewertet.

Es besteht auch die Möglichkeit eine grafische Oberfläche von Findbugs zu nutzen, welche über die Konsole gestartet wird. Auf diese Art der Fehlerausgabe wurde in dieser Bachelorarbeit verzichtet, da eine stetige Fehlerausgabe während der Entwicklung geschehen sollte und die Konsole dafür als ungeeignet erschien.

7.8.5 Testmöglichkeiten

In der *Bug Explorer View* werden nun die gefundenen Fehler angezeigt. Jeder Fehler erhält eine kurze Beschreibung und es wird die Klasse mit ausgegeben, in welcher der Fehler verursacht wird. Klickt man auf den Fehler gelangt man direkt in diese Klasse.

Android Findbugs war bei der Entwicklung von *ImkerApp* in der gesamten Implementierungsphase aktiviert und es kam zu keiner Fehlerausgabe.

7.8.6 Bewertung

Android Findbugs ist ein geeignetes Tool, um Fehler zu finden, welche sich auf den Android-spezifischen Sourcecode beziehen. Es ist einfach zu installieren und auch die Ausgaben der Fehler sind übersichtlich und gut verständlich.

7.9 Checkstyle

7.9.1 Beschreibung

Checkstyle ist ein Framework für die statische Codeanalyse. Das heißt, es prüft den Sourcecode in Java auf vorgegebene Programmierrichtlinien. Seit der Version 3 können aber auch Klassendesign-Probleme, duplizierter Code oder verschiedene (mögliche) Bugs gefunden werden. In der Entwicklungsumgebung eclipse ist Checkstyle bereits integriert. Hier hat der Entwickler einmal die Möglichkeit die Default-Einstellungen zu nutzen, um seinen Sourcecode zu prüfen oder er erstellt sich eine eigene Konfiguration. Dieses wird im Kapitel 7.9.4 näher erläutert. (Burn)

Wie schon erwähnt, gehört Checkstyle zu den Codeanalyse-Frameworks.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				x
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 7-10 Einordnung Checkstyle in Testphase bzw. Testart

7.9.2 Installation

Checkstyle[21] ist bereits im Android SDK enthalten.

Die Installation wird daher als einfach eingestuft, da der Entwickler lediglich das Android SDK installieren muss.

[21] http://checkstyle.sourceforge.net/

7.9.3 Einarbeitung

Die Einarbeitung in Checkstyle geht dank der guten und ausführlichen Dokumentation des Herstellers als auch der Community schnell von der Hand. Einzig und allein in die verschiedenen Möglichkeiten der Konfiguration, die sogenannten *Checks*, muss sich der Entwickler etwas einlesen, um zu erkennen, was die einzelnen *Checks* für eine Funktion der Prüfung anbieten. Dieser werden aber in vielen der vorhandenen Dokumentationen erläutert. Trotzdem muss der Entwickler meist selber noch einmal recherchieren, um alle Konfigurationen entsprechend seiner Richtlinien einzustellen.

Die Komplexität der Einarbeitung wird daher als mittel eingestuft, da der Entwickler zwar eine sehr gute Einleitung in Checkstyle durch die Hersteller und die Community erhält, aber selber recherchieren muss, um die gesamte Funktionalität von Checkstyle zu verinnerlichen.

7.9.4 Anlegen des Testprojektes

Da Checkstyle schon im Android SDK integriert ist, müssen lediglich nur noch ein paar Schritte erfolgen, um Checkstyle in dem jeweiligen Projekt zu verwenden. Mit einem Rechtsklick auf das Projekt öffnet sich ein Menü mit verschiedenen Einstellmöglichkeiten. Unter anderem befindet sich hier auch der Punkt *Checkstyle*. Geht man nun mit der Maus auf diesen Punkt öffnet sich wieder ein Menü mit Einstellungen für Checkstyle. Klickt man hier nun *Activate Checkstyle* wird Checkstyle aktiviert, dass entsprechende Projekt wird noch einmal neu gebildet und im Sourcecode sind die Verstöße gegen die Programmierrichtlienen zu erkennen. Wurden keine weiteren Einstellungen vorgenommen, werden die Verstöße anhand der Default-Einstellungen analysiert. Wie es dem Entwickler möglich ist, diese Einstellungen zu ändern wird im nächsten Kapitel 7.10.5 erklärt.

Der Aufwand zum Anlegen eines Testprojektes wird somit als niedrig bewertet.

7.9.5 Testmöglichkeiten

Wie schon im Kapitel 7.9.1 erwähnt bietet Checkstyle dem Entwickler die Möglichkeit einzelne Konfigurationen zu verschiedene Checks einzustellen. Diese vorhandenen Standartchecks sind in 15 Kategorien unterteilt:

Standartchecks	Beschreibung
Annotations	Diese Checks überprüfen den Quellcode im Zusammenhang mit Annotationen. Z.B. wird geprüft, ob die java.lang.Override-Annotation vorhanden ist, wenn das {@inheritDoc} Javadoc-Tag vorhanden ist.
Block Checks	Diese Checks überprüfen den Quellcode im Zusammenhang mit einzelnen Codeblöcken,

	z.B. wird nach leeren Blöcken gesucht.
Class Design	Diese Checks überprüfen den Quellcode im Zusammenhang mit dem Klassendesign. Sollte eine Klasse, die nur private Konstruktoren enthält, als final deklariert werden.
Coding	Diese Checks überprüfen den Quellcode im Zusammenhang mit der Codierung. Z.B. wird nach leeren Anweisungen gesucht oder ob lokale Variablen oder Methodenparameter, die ihren Wert nie ändern, als final deklariert sind.
Duplicate Code	Dieser Check überprüft den Sourcecode auf duplizierten Code.
Headers	Diese Checks überprüfen die Sourcecode-Dateien, ob sie mit einem Header beginnen.
Imports	Diese Checks überprüfen den Sourcecode im Zusammenhang mit Import - Anweisungen, z.B. werden ungenutzte Paket-Importe gefunden.
Javadoc Comments	Überprüft den Quellcode im Zusammenhang mit Javadoc-Kommentaren. Z.B. wird geprüft, ob jede Variable mit einem Javadoc-Kommentar versehen ist.
Metrics	Überprüft die definierten Klassen auf Eihaltung bestimmter Metriken, wie z.B. die *zyklomatische Komplexität.*
Missellaneus	Unter diesem Punkt sind verschiedene Überprüfungen zusammengefasst. Z.B. kann geprüft werden, ob eine Sourcecode-Datei am Ende einen Zeilenumbruch enthält.
Modifiers	Diese Checks beziehen sich auf die Verwendung der Modifier in Java. Z.B. kann überprüft werden, ob die Angabe der Modifier die Reihenfolge, wie sie in der Java Sprachspezifikation angegeben ist, einhält.
Naming Conventions	Diese Checks überprüfen den Quellcode auf die Einhaltung festgelegter Namenskonventionen. Diese Konventionen

	können durch reguläre Ausdrücke festgelegt werden.
Regexp	Hierrüber können verschiedene reguläre Ausdrücke bestimmt werden, auf die der Sourcecode überprüft werden soll. Z.B. kann nach Verwendungen von System.out.println“ gesucht werden.
Size Violations	Diese Checks überprüfen den Sourcecode auf die Einhaltung bestimmter Größenbeschränkungen. Z.B. kann der Sourcecode auf die Einhaltung bestimmter Zeilenlängen überprüft werden.
Whitespace	Diese Checks überprüfen den Sourcecode auf die Verwendung von Leerzeichen.

Tabelle 7-11 Beschreibung Standartchecks von Checkstyle (Burn)

Der Entwickler hat nun die Möglichkeiten die Standartchecks aus der Tabelle 7.9-2 verschieden zu konfigurieren. In der Entwicklungsumgebung über den Rechtsklick auf das jeweilige Projekt gelangt man wieder in das Menü mit den verschiedenen Einstellmöglichkeiten. Hier klickt man nun auf *Properties* gelangt man in ein Dialogfenster in welchem unter Anderem auch *Checkstyle* ausgewählt werden kann. Auf der rechten Seite erscheinen nun die einzelnen Einstellmöglichkeiten.

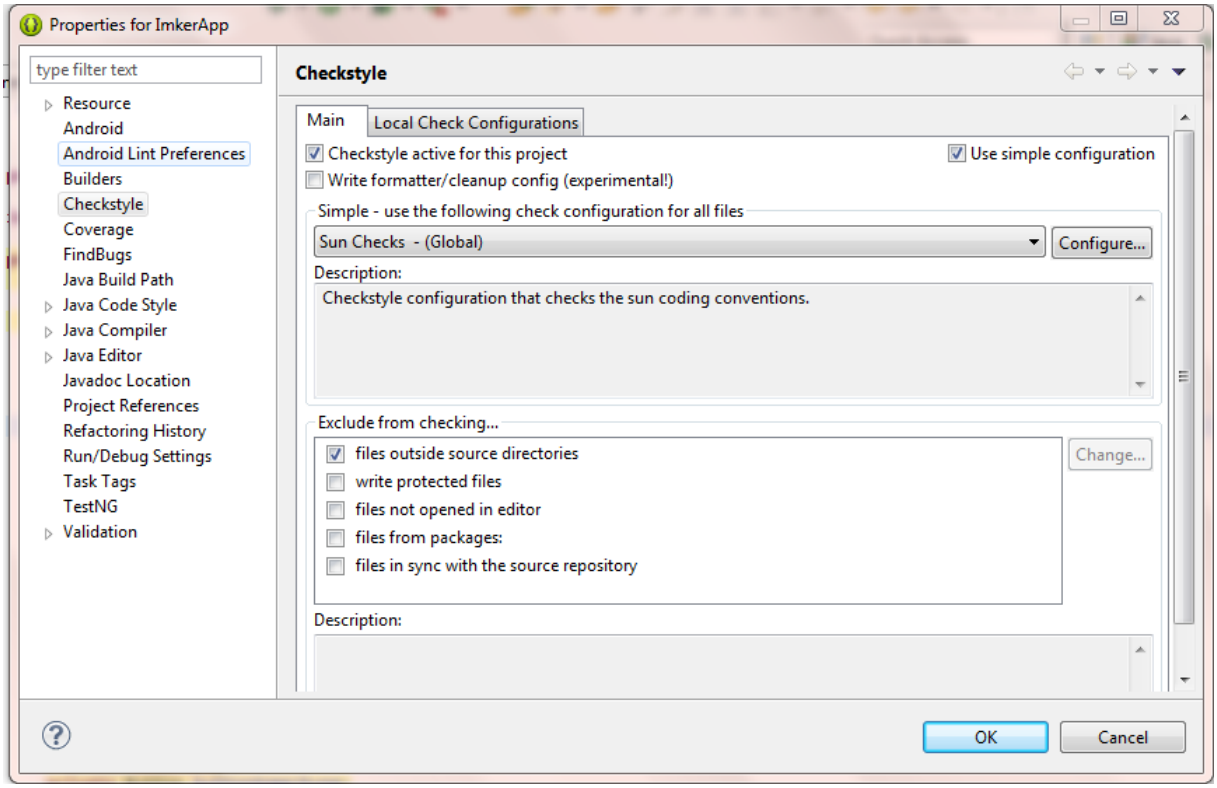


Abbildung 7.9-1 Einstellmöglichkeiten Checkstyle

Unter dem Punkt *Simple - use the following check configuration for all files* können nun entweder die Default-Konfigurartion, welche *Sun Checks* heißt, ausgewählt werden oder man klickt auf den Tab *Local Check Configuration* und erstellt sich eine eigene Konfiguration.

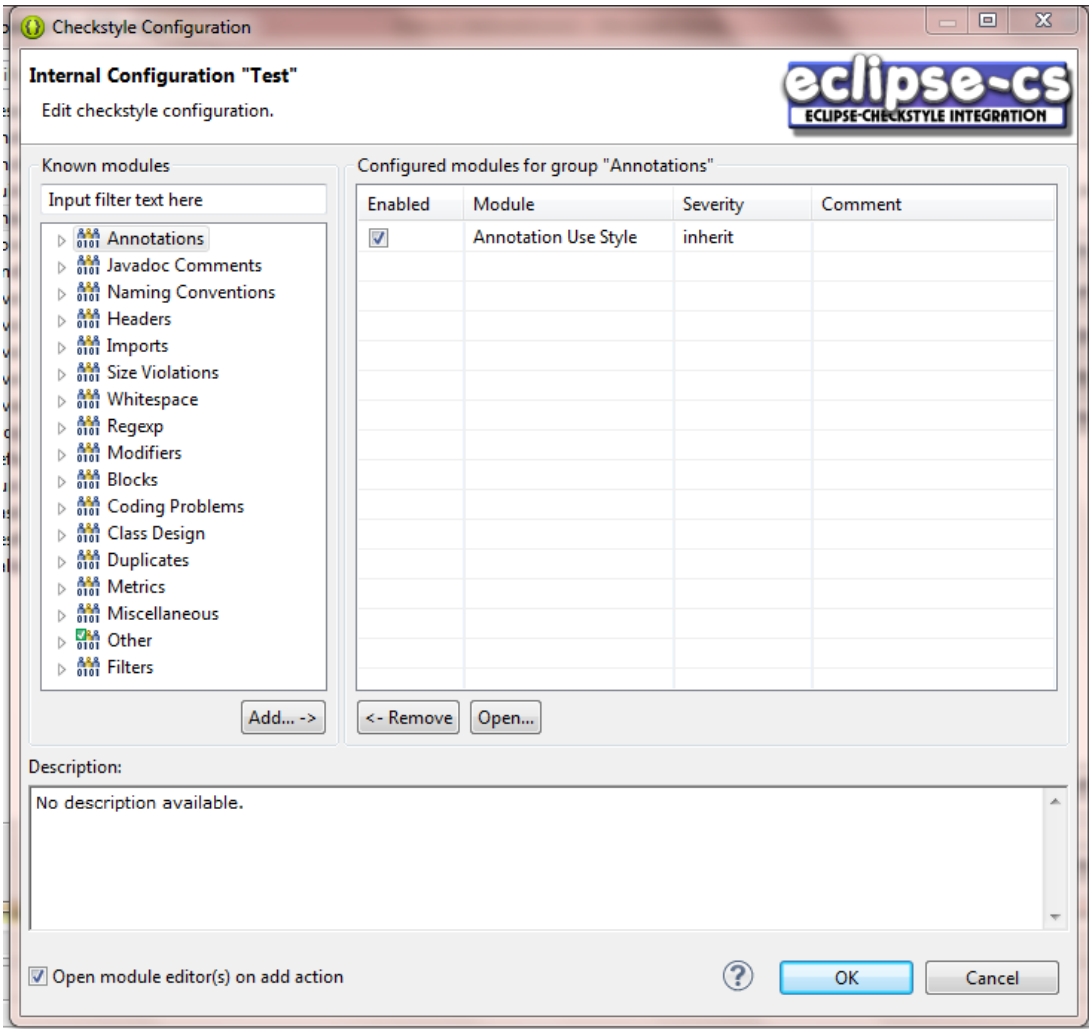


Abbildung 7.9-2 Konfiguration für Checks in Checkstyle

In der Abbildung 7.9-2 sind nun auf der linken Seite unter anderem die Standartschecks zu sehen, welche bereits in der **Fehler! Verweisquelle konnte nicht gefunden werden.** erklärt wurden. Hier hat der Entwickler nun sämtliche Möglichkeiten die Standartchecks nach seinen Kriterien zu ändern. Zum Beispiel kann eingestellt werden, dass die maximale Zeilenlänge einer Methode bei 200 liegen darf oder wie das Format einer lokalen Variablen auszusehen hat.

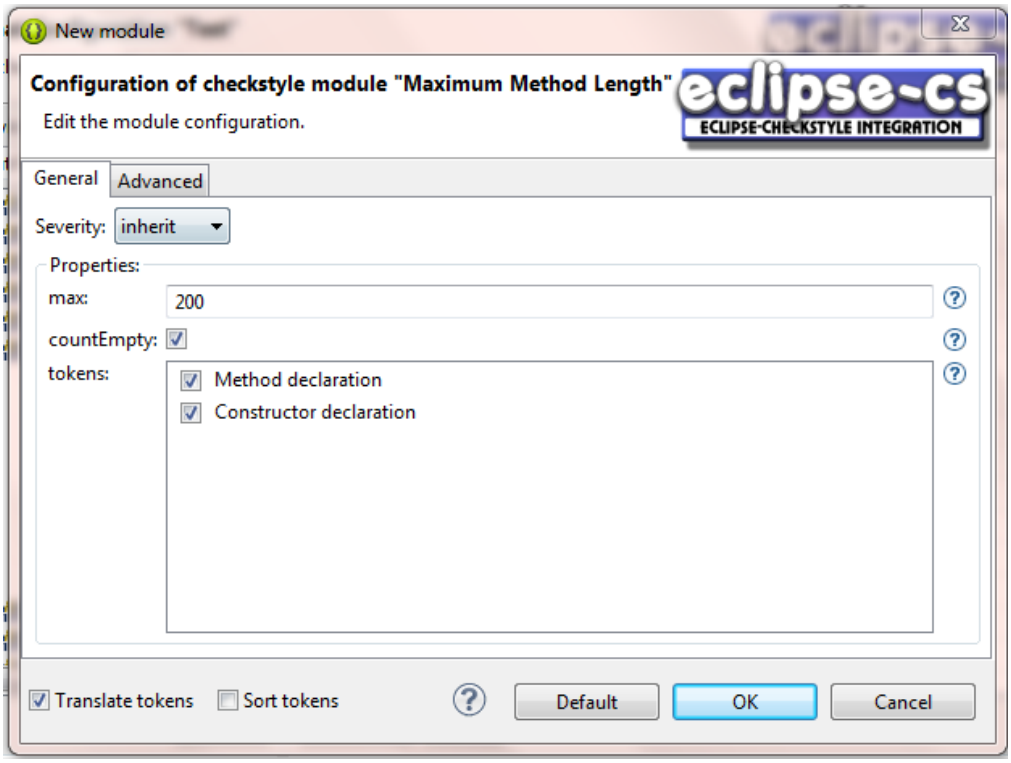


Abbildung 7.9-3 Einstellung maximale Länge einer Methode

In der Abbildung 7.9-3 ist nun der Standartcheck zu sehen, welcher es ermöglicht die maximale Zeilenlänge einer Methode einzustellen. Unter dem Tab *Advanced* hat der Entwickler zusätzlich die Möglichkeit einzustellen, welcher Text bei einem Verstoß gegen diesen Check ausgeben werden soll.

Hat der Entwickler nun alle Standartchecks auf seine Kriterien hin eingestellt, werden im gesamten Sourcecode des jeweiligen Projekts die Verstöße angezeigt.

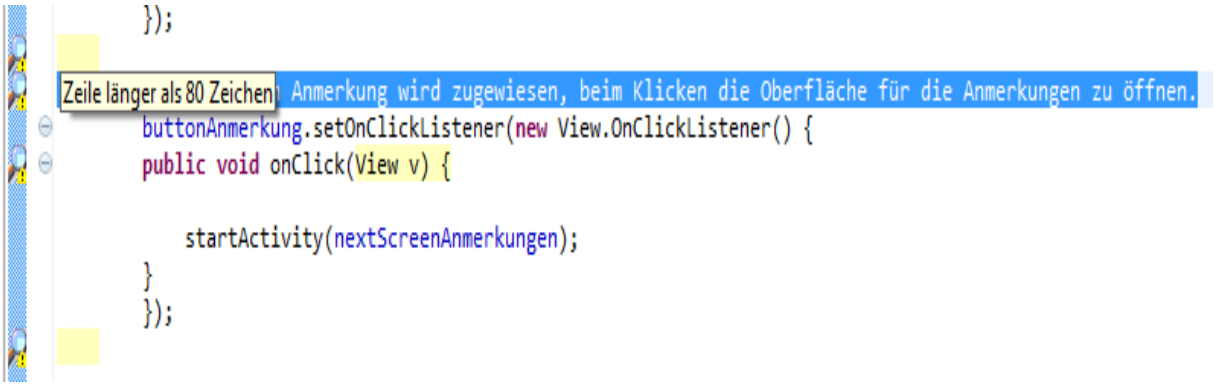


Abbildung 7.9-4 Beispielanzeige Verstöße in Checkstyle

In der Abbildung 7.9-4 sind solche Verstöße in der *MenuActivity* von *ImkerApp* dargestellt. Jede Codezeile, in welcher auf der linken Seite eine kleine Lupe auftaucht, enthält einen Verstoß. Geht man mit dem Mauszeiger über diese Lupe erhält man eine genauere Beschreibung zu dem entsprechenden Verstoß. Diese Lupe verschwindet erst, wenn der Entwickler den Verstoß verbessert hat und die Änderung speichert.

7.9.6 Bewertung

Checkstyle ist ein geeignetes Framework, um entsprechende Verstöße gegen Programmierrichtlinien aufzudecken. Dabei bleibt es dem Entwickler selber überlassen, ob er die Default-Einstellungen vom Hersteller übernimmt oder ob er die Standartchecks nach seinen Kriterien anpasst. Vom Hersteller und auch der Community gibt es ausreichende und gut verständliche Hilfestellungen. Möchte der Entwickler allerdings tiefer in die Materie einsteigen, zum Beispiel um die Bedeutung aller Standartchecks zu verstehen, muss er selber noch auf Recherche gehen, um passende Informationen zu finden.

7.10 PMD

7.10.1 Beschreibung

PMD ist ein Werkzeug für die statische Code-Analyse von Java-Quelltexten. Die Fehler, die PMD findet, befinden sich auf der Ebene von ineffizientem Code, d.h. die Software wird in der Regel trotzdem korrekt ausgeführt, wenn die Fehler nicht korrigiert werden. PMD findet auf Basis von statischen Regeln potentielle Probleme wie beispielsweise:

- mögliche Bugs (leere try/catch/finally/switch-Blöcke)
- toter Code (ungenutzte lokale Variablen, Parameter und private Methoden)
- leere if/while- Ausdrücke
- unnötige if-Ausdrücke oder for-Schleifen, welche stattdessen als while-Schleife genutzt werden können
- Klasse mit hoher zyklomatischer Komplexität

(Levent)

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				x
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 7-12 Einordung PMD in Testphase bzw. Testart

7.10.2 Installation

PMD kann als Plugin in das Android SDK installiert werden. Unter dem Menüpunkt *Help → Install new Software..* gelangt man zum Installationsdialog. Im oberen Bereich auf der rechten Seite befindet der Button *Add* unter welchem der Installationspfad von PMD[22] eingegeben werden. Klickt man nun auf *Next* folgt man den weiteren Anweisungen der Installation und PMD wird in eclipse integriert.

Die Installation wird daher als einfach eingestuft, da lediglich das vorhandene Plugin installiert werden muss.

7.10.3 Einarbeitung

Die Einarbeitung in PMD geht schnell von der Hand. Auf der Webseite des Herstellers existieren entsprechende Dokumentationen, welchen den Einstieg in PMD erleichtern. Aber auch die Community bietet schnelle Hilfe an. Somit ist es dem Entwickler möglich, sich schnell und effizient in PMD einzuarbeiten.

Die Komplexität der Einarbeitung wird daher als niedrig bewertet.

7.10.4 Anlegen eines Testprojektes

Nachdem das Plugin in eclipse erfolgreich installiert wurde, sind es nur noch ein paar Schritte, um PMD zu verwenden. Mit einem Rechtsklick auf das Projekt öffnet sich ein Menü mit verschiedenen Einstellmöglichkeiten. Unter anderem befindet sich hier auch der Punkt *PMD*. Geht man nun mit der Maus auf diesen Punkt öffnet sich wieder ein Menü mit Einstellungen für PMD. Klickt man hier nun *Check Code With PMD* wird PMD aktiviert, dass entsprechende Projekt wird noch einmal neu gebildet und im Sourcecode sind die Verstöße gegen die Programmierrichtlienen zu erkennen.

Der Aufwand, um ein Testprojekt erfolgreich anzulegen, wird somit als niedrig eingestuft.

7.10.5 Testmöglichkeiten

PMD bietet eine Reihe von möglichen Regeln an, um den jeweiligen Sourcecode auf Verstöße gegen Programmierrichtlienen zu prüfen.

[22] http://pmd.sourceforge.net/pmd-5.0.4/integrations.html#eclipse

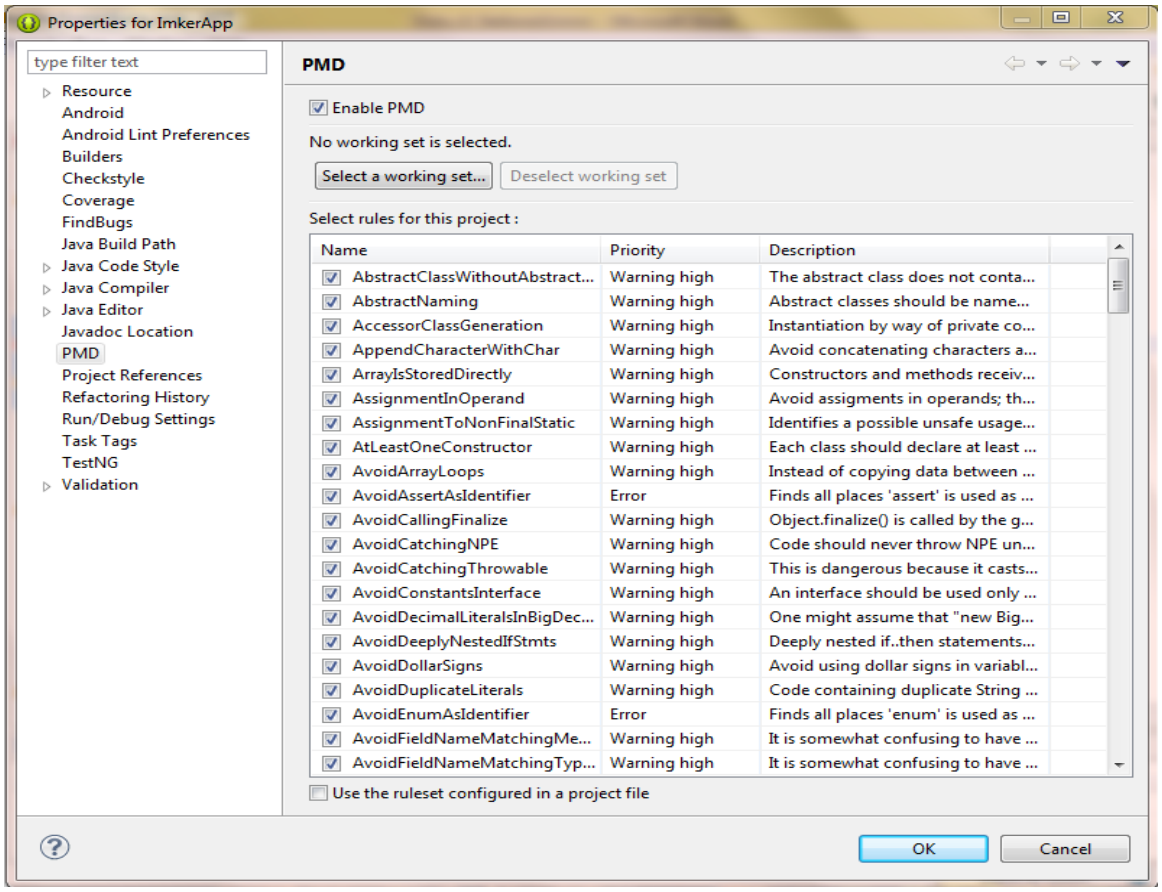


Abbildung 7.10-1 Ausschnitt Regeln von PMD in eclipse

In der Abbildung 7.10-1 sind ein paar von diesen Regeln zu sehen, welche der Entwickler nach seinen Kriterien hin auswählen kann. PMD bietet jedoch keine Möglichkeit die einzelnen Regeln zu ändern oder neue hinzuzufügen. In der Spalte *Priority* werden die einzelnen Regeln in Prioritäten eingeteilt. Es gibt insgesamt 5 Stufen:

- Stufe 1: error high
- Stufe 2: error
- Stufe 3: warning high
- Stufe 4: warning
- Stufe 5: information

In der folgenden Tabelle werden einige der Regeln näher erläutert.

Regel	Beschreibung
EmptyFinalizer	Ist die Abschluss-Methode leer, muss diese auch nicht existieren.
EmptyFinallyBlock	Leere finally()-Blöcke können gelöscht werden.

UnnescessaryReturn	Unnötige return-Anweisungen können vermieden werden.
OnlyOneReturn	Eine Methode sollte nur einen Punkt zum Beenden haben und das sollte die letzte Anweisung sein.
TooManyFields	Klassen sollten nicht zu viele Felder enthalten. Zum Beispiel könnten Stadt/Land/Ort als Adress-Feld zusammengefasst werden.
LongVariable	Variablen und Felder sollten keine zu langen Namen haben.
NoPackage	Jede Klasse und jedes Interface sollten eine Package-Deklaration enthalten.

Tabelle 7-13 Beschreibung einiger Regeln aus PMD (Levent)

Die in der Tabelle 7-13 Beschreibung einiger Regeln aus PMD (Levent)

2 beschriebenen Regeln sind nur ein Teil derer Regeln, welche PMD anbietet. Gesamt bietet PMD um die 149 verschiedenen Regeln an.

Hat der Entwickler nun alle für ihn relevanten Regeln ausgewählt und das Projekt wurde neu gebildet, werden die Verstöße im Sourcecode sichtbar.

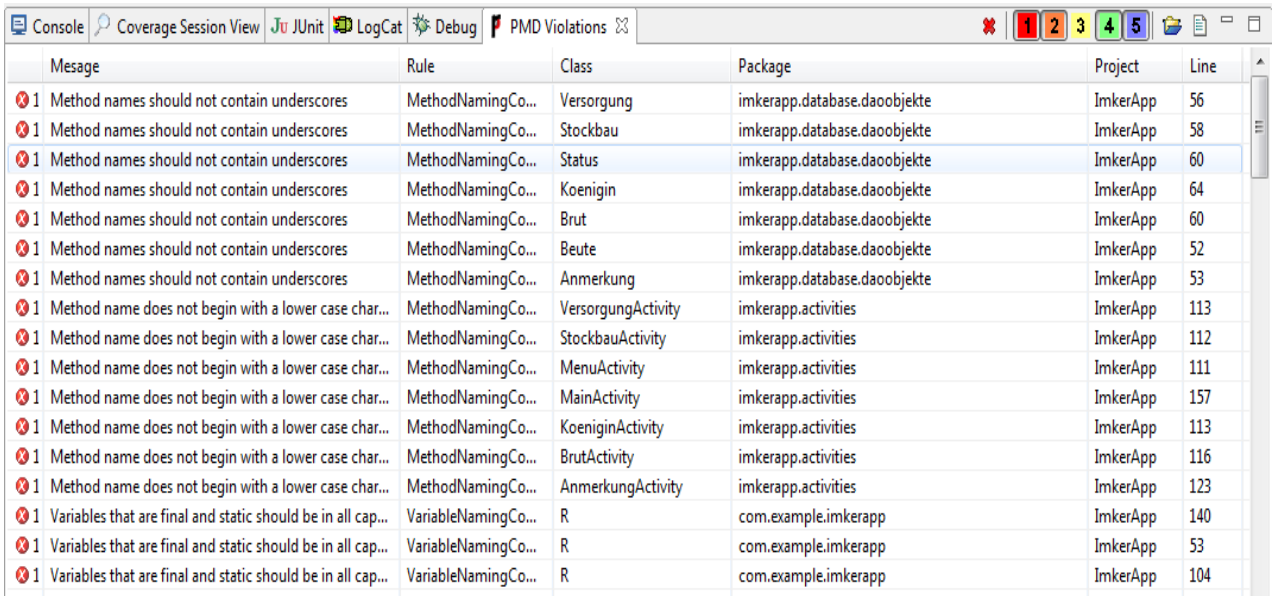
```
    /**
     * In dieser Methode wird zum einen auf die entsprechenden Button zugegriffen und ihnen Funktionen zugewiesen.
     */
    private void ButtonListener() {

     // Anlegen der Button
        buttonHistorie = (Button) findViewById(R.id.stockbauHistorie);
        buttonSpeichern = (Button) findViewById(R.id.stockbauSpeichern);

        // Erstellt einen Dialog
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setMessage("Ihre Daten wurden gespeichert!")
               .setCancelable(false)
               .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                   public void onClick(DialogInterface dialog, int id) {

                   }
               });
        final AlertDialog alertDialog = builder.create();
```

Abbildung 7.10-2 Anzeige der Verstöße bei PMD

Gravierende Verstöße gegen die Programmierrichtlinien werden mit einem weißen Kreuz auf weißem Grund markiert, weniger gravierende Verstöße mit einem schwarzen Ausrufezeichen in einem gelben Dreieck. Diese Symbole sind etwas verwirrend, da das weiße Kreuz eher als Fehler auf Syntaxebene bekannt ist, wonach das entsprechende Programm nicht lauffähig ist. Auch das gelbe Dreieck hat eine andere Bedeutung, denn es

zeigt normalerweise Warnhinweise an. In diesem Fall sind es aber nur Hinweise für Verbesserungen, welche der Entwickler nach eigenem Ermessen anwenden kann.

Eine andere Möglichkeit die gefundenen Verstöße anzeigen zu lassen ist *PMD Violations*. Dabei handelt es sich um eine tabellarische Auflistung der jeweiligen Verstöße.



	Mesage	Rule	Class	Package	Project	Line
1	Method names should not contain underscores	MethodNamingCo...	Versorgung	imkerapp.database.daoobjekte	ImkerApp	56
1	Method names should not contain underscores	MethodNamingCo...	Stockbau	imkerapp.database.daoobjekte	ImkerApp	58
1	Method names should not contain underscores	MethodNamingCo...	Status	imkerapp.database.daoobjekte	ImkerApp	60
1	Method names should not contain underscores	MethodNamingCo...	Koenigin	imkerapp.database.daoobjekte	ImkerApp	64
1	Method names should not contain underscores	MethodNamingCo...	Brut	imkerapp.database.daoobjekte	ImkerApp	60
1	Method names should not contain underscores	MethodNamingCo...	Beute	imkerapp.database.daoobjekte	ImkerApp	52
1	Method names should not contain underscores	MethodNamingCo...	Anmerkung	imkerapp.database.daoobjekte	ImkerApp	53
1	Method name does not begin with a lower case char...	MethodNamingCo...	VersorgungActivity	imkerapp.activities	ImkerApp	113
1	Method name does not begin with a lower case char...	MethodNamingCo...	StockbauActivity	imkerapp.activities	ImkerApp	112
1	Method name does not begin with a lower case char...	MethodNamingCo...	MenuActivity	imkerapp.activities	ImkerApp	111
1	Method name does not begin with a lower case char...	MethodNamingCo...	MainActivity	imkerapp.activities	ImkerApp	157
1	Method name does not begin with a lower case char...	MethodNamingCo...	KoeniginActivity	imkerapp.activities	ImkerApp	113
1	Method name does not begin with a lower case char...	MethodNamingCo...	BrutActivity	imkerapp.activities	ImkerApp	116
1	Method name does not begin with a lower case char...	MethodNamingCo...	AnmerkungActivity	imkerapp.activities	ImkerApp	123
1	Variables that are final and static should be in all cap...	VariableNamingCo...	R	com.example.imkerapp	ImkerApp	140
1	Variables that are final and static should be in all cap...	VariableNamingCo...	R	com.example.imkerapp	ImkerApp	53
1	Variables that are final and static should be in all cap...	VariableNamingCo...	R	com.example.imkerapp	ImkerApp	104

Abbildung 7.10-3 PMD Violations

Der Entwickler hat hier die Möglichkeit entweder alle Prioritäten anzeigen zu lassen oder nur bestimmte. Dieses geschieht über die 5 bunten Zahlen in der rechten oberen Ecke, wobei jede Zahl für eine der 5 Stufen steht. Klickt der Entwickler auf einen der Verstöße gelangt er direkt in die jeweilige Klasse und Sourcecodezeile, wo dieser Verstoß auftaucht.

Nachdem der Entwickler alle für ihn relevanten Verstöße verbessert hat, muss er PMD unter Rechtsklick auf das Projekt → *PMD* → *Clear PMD Violations* wieder deaktivieren. Ansonsten wird das Programm nicht lauffähig sein, da Verstöße der Priorität 1 und 2 als Fehler ausgelegt werden, nach denen das Programm nicht lauffähig ist.

7.10.6 Bewertung

PMD ist ein nützliches Werkzeug für die statische Codeanalyse. Es existiert eine einfache und gut verständliche Dokumentation des Herstellers und auch der Community. Somit ist dem Entwickler ein schneller Einstieg in PMD garantiert. Auch die Auswahl der einzelnen Regeln ist dank der Beschreibung zu jeder Regel einfach zu erledigen. Dank der tabellarischen Auflistung der Verstöße, welche nach den 5-Stufen der Priorität gefiltert werden können, ist eine schnelle Verbesserung möglich. Negativ ist, dass die Verstöße als Fehler angezeigt werden, nach denen das Programm nicht lauffähig ist. Möchte der Entwickler neben dem Implementieren sofort die Verstöße verbessern, ist es notwendig PMD vor jedem Start des Programms wieder zu deaktivieren. Dieses kann auf Dauer störend sein.

8 Evaluierung der nicht funktionierenden Frameworks

In diesem Kapitel werden alle Frameworks erläutert, welche nicht erfolgreich mit der Testapplikation *ImkerApp* funktioniert haben. Bei diesen Frameworks erfolgt keine ausführliche Beschreibung und Evaluation, wie bei den Frameworks aus Kapitel 7, sondern nur eine kurze Erklärung zu der Funktionsweise des jeweiligen Frameworks und die Gründe, warum diese nicht mit Android kompatibel sind.

In der folgenden Tabelle werden die Frameworks in die jeweilige Testphase beziehungsweise Testart eingeordnet.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest		Cobertura		
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	Abbot UISpec4J			

Tabelle 8-1 Einordnung nicht funktionierender Frameworks in Testphase und Testart

8.1 Abbot

8.1.1 Beschreibung

Abbot dient zum Testen von grafischen Benutzeroberflächen. Es bietet die Möglichkeit, Testskripte mittels Costello[23] aufzunehmen und diese zu einem späteren Zeitpunkt erneut abzuspielen. Dabei werden bei der Ausführung des Skripts bestimmte Werte oder Ereignisse miteinander verglichen und ein Testergebnis generiert. (Wall)

Abbot gehört zu den Frameworks der Oberflächentests. Diese prüfen die Funktionalität hinsichtlich der Spezifikationen.

[23] Costello basiert auf Abbot und ist die Standalone-Anwendung, um Benutzeroberflächen zu testen.

Testphase x Testart	Funktions-test	Kontrollfluss-test	Softwaremes-sung	Stil-/ Codeanalyse
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	x			

Tabelle 8-2 Einordung Abbot in die Testphasen bzw. Testarten

8.1.2 Probleme mit Android

Abbot ist nicht kompatibel mit Android. Bereits bei dem Anlegen des Testprojektes taucht ein Problem auf. In Java ist es nur möglich auf eine Basisklasse zu verweisen. Da aber die Bsisklasse *ActivityTestCase* zwingend erforderlich ist, um Testanwendungen in Android zu implementieren, kann *ComponentTestFixure*, welche für Abbot benötigt wird, nicht zusätzlich als Basisklasse fungieren.

Ein weiterer Punkt, weshalb Abbot nicht für den Einsatz mit Android Applications genutzt werden kann, wird ersichtlich, wenn man sich einmal genauer anschaut, was Abbot testen kann. Ein Beispiel ist in folgendem Sourcecodeausschnitt zu sehen.

```
public void testButtons() {

    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
            clickType = ev.getActionCommand();
            }
        };

    /*Fügt die TestObjekte zum ActionListener hinzu*/
    oPanel.getButtonCount().addActionListener(al);
    oPanel.getJButtonReset().addActionListener(al);
    oPanel.getComboBox().addActionListener(al);


    /*Count Button*/
    tester.actionDelay(1000);
    tester.actionClick(oPanel.getButtonCount());
    assertEquals("Click Count failed", "Count", clickType);

    tester.getLabel(oPanel.getJLabel());

    assertEquals("Label not equals!", "Counter : 1", oPanel.getJLabel().getText());
```

Abbildung 8.1-1 Sourcecode Beispiel Abbot (FH Osnabrück)

Hier ist zu erkennen, dass Abbot nur in der Lage ist auf Methoden der Klassen *java.awt* und *java.swing* zuzugreifen, welche beispielsweise die Methoden *ActionListener()* oder *Label* enthalten. Es besteht aber keine Möglichkeit mit Abbot auf Android-spezifische Klassen zuzugreifen.

8.2 UISpec4J

8.2.1 Beschreibung

UISpec4J ist ein Framework zum Testen von Benutzeroberflächen, welches komplett auf JUnit aufsetzt. Somit ist es auch für den Einsatz von komplexen Testszenarien in großen Projekten geeignet. Dank der vorhandenen Bibliotheken, welche in das jeweilige Testprojekt eingebunden werden müssen, lassen sich schnell Oberflächentests implementieren. Die Ausgabe der Testergebnisse funktioniert genau wie bei JUnit. Erfolgreich durchlaufene Testmethoden werden grün angezeigt, nicht erfolgreich durchlaufene rot.
(Medina & Pratmarty)

Wie schon erwähnt gehört UISpec4J zu den Oberflächentests, welche die Funktion der Software nach den Spezifikationen prüfen.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	x			

Tabelle 8-3 Einordnung UISpec4J in Testphasen bzw. Testarten

8.2.2 Probleme mit Android

UISpec4J ist nicht kompatibel mit Android. Dieses wird deutlich, wenn man sich einmal anschaut, wie eine Testklasse aufgebaut ist.

```
public class MyTest extends TestCase{
    Panel testThisOne;
    MyMainPanel content;
    static {
        UISpec4J.init();
    }

    public void setUp(){
        content = new MyMainPanel();
        testThisOne = new Panel(content);
    }

    public void testNrOne(){

        testThisOne.getButton("Count").click();
        assertTrue(testThisOne.containsLabel("Counter : 1").isTrue());
        testThisOne.getButton("Count").click();
        assertTrue(testThisOne.containsLabel("Counter : 2").isTrue());
        for(int i=0;i<10;i++){
            testThisOne.getButton("Count").click();
        }
        assertTrue(testThisOne.containsLabel("Counter : 12").isTrue());
    }
```

Abbildung 8.2-1 Sourcecodebeispiel UISpec4J (FH Osnabrück)

In der Abbildung 8.2-1 wird deutlich, dass UISpec4J nur in der Lage ist mit Swing-Applikationen zu interagieren, denn es wird ein Panel initialisiert und keine Activity, wie bei Android üblich. Somit ist UISpec4J nicht für das Testen mit Android geeignet.

8.3 MonkeyRunner

8.3.1 Beschreibung

MonkeyRunner ist ein Framework für Oberflächentests, welches bereits im Android SDK enthalten ist. Mit Hilfe von Programmen, welche mit Python implementiert werden, besteht die Möglichkeit das Verhalten eines Nutzers zu simulieren. Während das Programm abläuft, werden Screenshots von den jeweiligen Oberflächen der Applikation erstellt, welche dem Entwickler helfen, dass Arbeiten von monkeyRunner besser nachzuvollziehen. monkeyRunner bietet verschiedene Features an. Zum einen können mehrere externe Geräte oder Emulatoren mit dem implementieren Python-Skript zur gleichen Zeit getestet werden. Des Weiteren kann ein "Start-to-Finish" Test durchgeführt werden. Das heißt, es werden durch das Testprogramm Eingaben und Touchbefehle implementiert. Mit Hilfe der Screenshots können dann die Ergebnisse ausgewertet werden. Ein weiteres Feature liegt darin, dass mit monkeyRunner Regressionstests durchgeführt werden. Mit anderen Worten, es kann die Stabilität der Applikation überprüft werden. Dieses geschieht, indem die Applikation durchläuft und die gemachten Screenshots mit einer Sammlung von Screenshots verglichen werden, von denen der Entwickler sicher ist, dass diese das korrekte Verhalten

der Applikation anzeigen. monkeyRunner selber nutzt Jython, eine Implementierung von Python, welche selber Java nutzt. Dank Jython kann die monkeyRunner API problemlos in das Android SDK integriert werden. Mit Jython kann der Entwickler dann die Python-Syntax verwenden, um auf die Konstanten, Klassen und Methoden der API zuzugreifen. (android developer monkeyRunner)

Neben dem Implementieren von Python-Programmen, besteht auch noch die Möglichkeit monkeyRunner mit Hilfe einer graphischen Oberfläche zu bedienen. Mit Hilfe des monkey_recorder können Eingaben und Touchbefehle aufgezeichnet werden, welche dann mit dem monkey_playback wieder abgespielt werden. (Sirisena, 2011) Da es bis zum jetzigen Zeitpunkt nicht gelungen ist, monkeyRunner in dieser Bachelorarbeit mit Hilfe der Python-Skripte zu nutzen, wurde nur die graphische Oberfläche in die Bewertung mit aufgenommen.

Wie schon weiter oben erwähnt, gehört monekyRunner zu den Oberflächentests. Es besteht aber auch die Möglichkeit, Regressionstests durchzuführen. Beide zielen allerdings auf den Aspekt der Funktionalität der Applikation ab.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	x x			

Tabelle 8-4 Einordung monkeyRunner in die Testphasen bzw. Testarten

8.3.2 Probleme mit Android

Um zu verstehen, zu welchen Problemen es bei monkeyRunner mit dem Testen von *ImkerApp* gab, ist im Folgenden zunächst einmal das Python-Skript mit dem Sourcecode abgebildet.

```
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
import commands
import sys

print "Anfang"

#Verbindung mit externem Gerät herstellen
device = MonkeyRunner.waitForConnection()

#Prüfen, ob ImkerApp schon installiert ist, wenn nicht installieren
apk_path = device.shell('pm path com.example.imkerapp')
if apk_path.startswith('package:'):
    print "imkerapp already installed."
else:
    print "imkerapp not installed, installing APKs..."
    device.installPackage('C:/Users/Stefanie/Documents/Bachelorarbeit/TestenBA/ImkerApp/bin/ImkerApp.apk')

print "launching imkerapp..."

package = 'com.example.imkerapp.activities'

activity= '.MainActivity'

runComponent = package + '/' + activity

# Activity MainActivity starten
device.startActivity(component=runComponent)

#MonkeyRunner anhalten, Snapshot machen und speichern
MonkeyRunner.sleep(1)
result1 = device.takeSnapshot()
result1.writeToFile('C:/Users/Stefanie/Documents/Bachelorarbeit/TestenBA/shot1.png', 'png')
print "screen 1 taken"


print "Ende"
```

Abbildung 8.3-1 Sourcecode monkeyRunner

Gestartet wird das Skript, indem über die Konsole der Pfad zu *monkeyrunner.bat* eingegeben wird und im Anschluss das Python-Skript, welches ausgeführt werden soll.

monkeyrunner monkey.py

```
C:\Users\Stefanie\Documents\Bachelorarbeit\Frameworks\adt-bundle-windows-x86_64-2
Anfang
imkerapp already installed.
launching imkerapp...
screen 1 taken
Ende
```

Abbildung 8.3-2 Konsolenausgabe monkeyRunner

Nachdem das Skript gestartet wurde werden die *print-Anweisungen* ausgegeben, welche in der Abbildung 8.3-2 zu sehen sind. Leider wird *ImkerApp* nicht ausgeführt und auch kein Snapshot der Anwendung erstellt sondern vom Hauptbildschirm des externen Geräts. Es

wurde in dieser Bachelorarbeit auf verschiedene Art und Weisen versucht, *ImkerApp* starten. Leider gab es auch weder beim Hersteller noch bei der Community hilfreiche Vorschläge. Das Problem liegt darin, dass wohl der Pfad zu *ImkerApp* auf dem externen Gerät falsch ist. Da es aber zu keiner Fehlerausgabe kommt, war es nicht möglich, dass Problem zu beheben.

8.4 Cobertura

8.4.1 Beschreibung

Cobertura ist ein Framework für die Ermittlung der Code Coverage für Java-Programme. Es kann also verwendet werden, um herauszufinden welche Teile des Sourcecodes genügend Testabdeckung erzielen. Das Ergebnis wird anhand der einzelnen Pakete eines Programms als Baum dargestellt. Cobertura berechnet für alle Klassen die prozentuale Abdeckung der Sourcecode-Zeilen und die Verzweigungen sowie die McCabe-Komplexität[24]. Cobertura ist in der Lage die Zweigüberdeckung zu berechnen.

(christ66) (Schlauch, 2005)

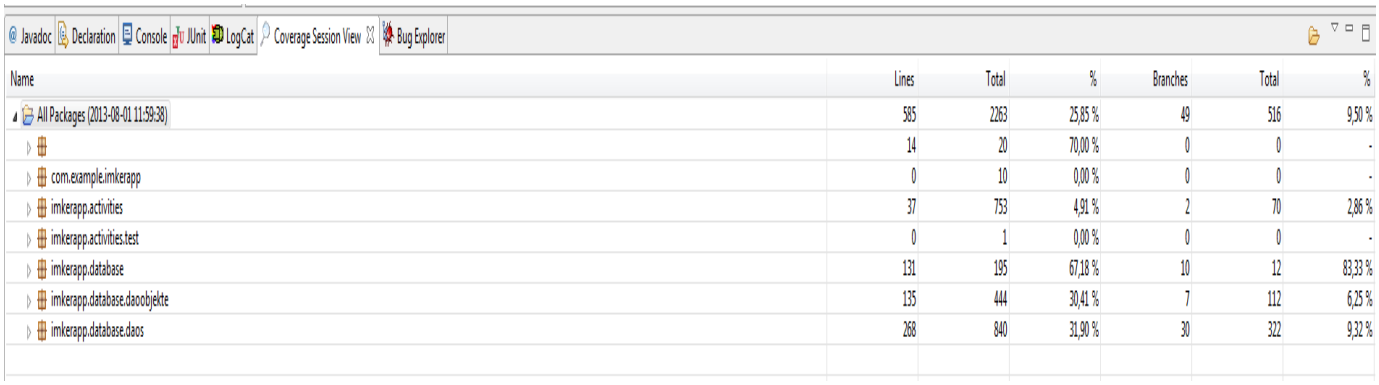
Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest		x		
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 8-5 Einordnung Cobertura in Testphase bzw. Testart

8.4.2 Probleme mit Android

Da Cobertura nur in Verbindung mit dem Java Bytecode funktioniert, konnte es in dieser Bachelorarbeit nicht dem Framework JUnit angewendet werden, sondern wurde mit dem Framework Robolectric angewendet, da dieses mit der Java VM interagiert und nicht mit der Dalvik VM. Da nun aber, wie schon im Kapitel 7.2.5 beschrieben, es bei Robolectric nur die Möglichkeit gab die Tests einzeln durchlaufen zu lassen und nicht als Testsuite, ist auch das Code Coverage Resultat von Cobertura verfälscht.

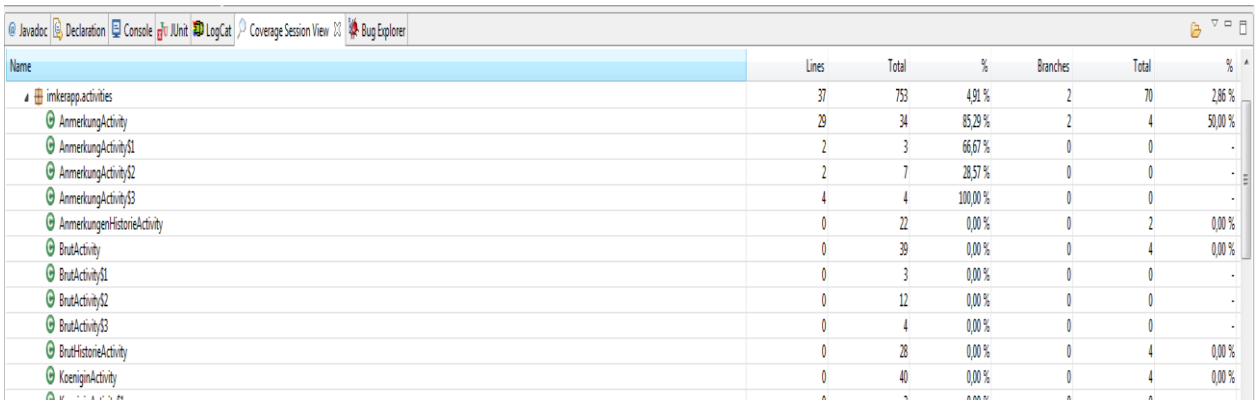
[24] McCabe-Komplexität gehört zu den Software-Metriken welche im Kapitel 3.5 beschrieben wurden.



Name	Lines	Total	%	Branches	Total	%
All Packages (2013-08-01 11:59:38)	585	2263	25,85 %	49	516	9,50 %
	14	20	70,00 %	0	0	-
com.example.imkerapp	0	10	0,00 %	0	0	-
imkerapp.activities	37	753	4,91 %	2	70	2,86 %
imkerapp.activities.test	0	1	0,00 %	0	0	-
imkerapp.database	131	195	67,18 %	10	12	83,33 %
imkerapp.database.daoobjekte	135	444	30,41 %	7	112	6,25 %
imkerapp.database.daos	268	840	31,90 %	30	322	9,32 %

Abbildung 8.4-1 Cobertura Session View Baumstruktur der Pakete

In der Abbildung 8.4-1 ist die Ausgabe der Ergebnisse der Code Coverage von *ImkerApp* zu sehen, nachdem einer der Testfälle ausgeführt wurde. Es werden einmal die gesamten Sourcecodezeilen angezeigt und die berechnete Branch Coverage in Prozenten.



Name	Lines	Total	%	Branches	Total	%
imkerapp.activities	37	753	4,91 %	2	70	2,86 %
AnmerkungActivity	29	34	85,29 %	2	4	50,00 %
AnmerkungActivity$1	2	3	66,67 %	0	0	-
AnmerkungActivity$2	2	7	28,57 %	0	0	-
AnmerkungActivity$3	4	4	100,00 %	0	0	-
AnmerkungenHistorieActivity	0	22	0,00 %	0	2	0,00 %
BrutActivity	0	39	0,00 %	0	4	0,00 %
BrutActivity$1	0	3	0,00 %	0	0	-
BrutActivity$2	0	12	0,00 %	0	0	-
BrutActivity$3	0	4	0,00 %	0	0	-
BrutHistorieActivity	0	28	0,00 %	0	4	0,00 %
KoeniginActivity	0	40	0,00 %	0	4	0,00 %

Abbildung 8.4-2 Cobertura Session View Paket imkerapp.activities

In der Abbildung 8.4-2 wurde in das Paket imkerapp.activies gezoomt, um die Code Coverage der einzelnen Activities näher zu betrachten. Hier wird nun deutlich, dass lediglich bei den Activities eine Coverage berechnet werden konnten, welche mit dem erfolgreich durchlaufenden Testfall in Verbindung stehen. In diesem Fall wurde in der Activity *Anmerkung* der Zugriff auf den *Button Historie* getestet.

8.5 Jenkins

8.5.1 Beschreibung

Jenkins ist ein Framework für die kontinuierliche Integration in Softwareprojekten. Es ist webbasiert und erweiterbar und wurde früher unter dem Namen *Hudson* entwickelt. Es wurde in Java implementiert und läuft in einem beliebigen Servlet-Container. Jenkins unterstützt verschiedene Built-Tools, unter anderem Apache Ant oder Maven. Aber es werden auch Programme wie Subversion, CVS und JUnit unterstützt. Mit Hilfe von

verschiedenen Plugins ist es möglich, dass auch PHP- oder Ruby-Projekte verwaltet werden können. (Fisher, Prakash, & Mills)

Jenkins ist ein Framework für Integrationstests.

Testphase x Testart	**Funktions-test**	**Kontrollfluss-test**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest				
Integrationstest	x			
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**				

Tabelle 8-6 Einordnung Jenkins in Testphase bzw. Testart

8.5.2 Probleme mit Android

In dieser Bachelorarbeit gab es keine direkten Probleme mit Android, da schon beim Anlegen des Projektes in Jenkins Fehler auftraten. In der webbasierten Oberfläche sollte ein neuer *Job* angelegt werden, welcher die Möglichkeit bieten sollte, das Android Projekt kontinuierlich zu integrieren. Bei der Konfiguration des *Job* sollte aus einem Source-Code-Management System, in diesem Fall *Bitbucket*, der jeweilige Sourcecode aus dem *MasterBranch* geholt werden, um im Anschluss das *Build-Verfahren* zu starten. Jedoch war es nicht möglich den Sourcecode aus *Bitbucket* vollständig zu klonen. Im Verlaufe dieser Bachelorarbeit wurde mehrmals versucht eine Lösung zu finden. Dieses hat aber bis zum Ende zu keinem Ergebnis geführt.

9 Fazit der Bachelorarbeit

Die Aufgabe dieser Bachelorarbeit war es, einen Überblick über die Testarten und Teststufen des Softwarelebenszyklus zu geben. Des Weiteren sollten anhand einer Matrix Frameworks recherchiert und evaluiert werden, welche für den Testeinsatz unter Android geeignet sind.

Der Überblick über die Testarten und Teststufen wurde in den Kapiteln 3 und 4 erfolgreich umgesetzt. Um die recherchierten Frameworks zu evaluieren wurde eine entsprechende Applikation entwickelt. Für diese Applikation *ImkerApp* wurden Anforderungskriterien aufgestellt, nach denen entwickelt wurde. Die Beschreibung der Entwicklung wurde im Kapitel 5 festgehalten.

Zur Evaluation der einzelnen Frameworks wurden nun noch Kriterien benötigt, welche im Kapitel 6 dokumentiert sind. Im Kapitel 7 wurden alle Frameworks evaluiert, die erfolgreich in Android umgesetzt werden konnten. Aus diesen Frameworks entstand auch die folgende Matrix, welche als Ergebnis dieser Bachelorarbeit anzusehen ist.

Testphase x Testart	**Funktionstest**	**Kontroll-flusstest**	**Softwaremes-sung**	**Stil-/ Codeanalyse**
Komponententest/ Modultest	JUnit Robolectric		Metrics	Android Lint Android Findbugs Checkstyle
Integrationstest				
Systemtest • **Funktionstest** • **Leistungstest** • **Stresstest** • **Regressionstest**	Robotium monkey Metrics		Traceview	

Tabelle 9-1 Ergebnis der evaluierten Frameworks

Es wurden alle erfolgreich angewendeten Frameworks in das Ergebnis mit aufgenommen, außer *PMD*. Im Vergleich mit *Checkstyle*, das nahezu dieselben Testmöglichkeiten anbietet, hat *PMD* schlechter abgeschnitten. Es bietet dem Entwickler kaum Möglichkeiten Testkriterien nach seinen Vorstellungen einzustellen und die Ausgabe der gefundenen Verstöße ist unpassend. Auch *Robolectric* wurde mit in das Ergebnis aufgenommen. Leider war es durch *greenDAO* nicht möglich, die implementierten Tests als Testsuite durchlaufen lassen. Jedoch liefen die Tests einzeln durch und Robolectric bietet dem Entwickler eine einfache und schnelle Implementierung von Modultests an. Auch JUnit wurde zusätzlich noch als Modultestframework mit aufgenommen. Es bietet dem Entwickler ebenfalls die Möglichkeit effiziente Tests zu verfassen. Bei den Stil-/ Codeanalyseframeworks wurden mehrere mit aufgenommen, da alle eine unterschiedliche Aufgabe erfüllen, zusammen aber eine gute Kombination anbieten. *Android Lint* bietet dabei eine gute Grundlage für die

schnelle Erkennung und Behebung von Optimierungsverbesserungen und Bugs im Sourcecode. *Android Findbugs* bietet eine gute Erweiterung, indem es Bugs im javaspezifischen Sourcecode aufspürt. Und *Checkstyle* ergänzt das Trio, indem es Verstöße gegen die Programmierrichtlinien aufdeckt und dem Entwickler die Möglichkeit bietet, individuelle Einstellungen vorzunehmen.

Bei den Frameworks aus Kapitel 8, die nicht erfolgreich mit Android angewendet werden konnten, war es interessant zu sehen, dass auch Frameworks, welche speziell für Android entwickelt wurden, keine Garantie bieten, erfolgreich Testfälle zu implementieren. *monkeyRunner* war leider nicht in der Lage *ImkerApp* auf dem externen Gerät zu starten, sodass der eigentliche Testfall nicht ausgeführt werden konnte. Und dadurch, dass es Probleme bei *Robolectric* mit *greenDAO* gab, war auch eine Code Coverage Analyse mit *Cobertura* nicht mehr möglich. Daher konnte in dieser Bachelorarbeit kein Framework für Kontrollflusstests mit in das Ergebnis aufgenommen werden. Ebenfalls konnte kein Framework für Integrationstests ermittelt werden, da bei *Jenkins* im Laufe dieser Bachelorarbeit keine Lösung für die Schwierigkeiten mit *Bitbucket* beim Anlegen eines *Job* gefunden wurde.

Mit den Frameworks aus Tabelle 9-1 ist es allerdings möglich Softwaretests durchzuführen und somit die Qualitätssicherung einer Applikation zu gewährleisten.

10 Literaturverzeichnis

Andreas Spillner, T. L. (2012). *Basiswissen Softwaretest.* Heidelberg: dpunkt.verlag GmbH.

android developer Android lint. (kein Datum). Abgerufen am 23. 06 2013 von http://developer.android.com/tools/debugging/improving-w-lint.html

android developer JUnit. (kein Datum). Abgerufen am 18. 06 2013 von http://developer.android.com/tools/testing/activity_testing.html

android developer monkey. (kein Datum). Abgerufen am 22. 06 2013 von http://developer.android.com/tools/help/monkey.html

android developer monkeyRunner. (kein Datum). Abgerufen am 21. 06 2013 von http://developer.android.com/tools/help/monkeyrunner_concepts.html

android developer traceview. (kein Datum). Abgerufen am 22. 06 2013 von http://developer.android.com/tools/help/traceview.html

Burn, O. (kein Datum). *Sourceforge*. Abgerufen am 25. 07 2013 von http://checkstyle.sourceforge.net/

christ66. (kein Datum). *cobertura*. Abgerufen am 30. 07 2013 von http://cobertura.github.io/cobertura/

Do, M. (1. Mai 2010). *Google*. Abgerufen am 3. Mai 2010 von Google: http://www.google.de

FH Osnabrück. (kein Datum). Abgerufen am 21. 06 2013 von http://home.edvsz.fh-osnabrueck.de/skleuker/CSI/index.html

FH Osnabrück. (kein Datum). Abgerufen am 23. 06 213 von http://home.edvsz.hs-osnabrueck.de/skleuker/CSI/Werkzeuge/abbot.html

FH Osnabrück. (kein Datum). Abgerufen am 16. 07 2013 von http://home.edvsz.fh-osnabrueck.de/skleuker/WS10_QS/WS10SQ_Teil04_1.pdf

FH Osnabrück. (kein Datum). Abgerufen am 24. 07 2013 von http://home.edvsz.hs-osnabrueck.de/skleuker/CSI/Werkzeuge/uispec4j.html

Fisher, S., Prakash, W., & Mills, D. (kein Datum). *eclipse foundation*. Abgerufen am 05. 08 2013 von http://wiki.eclipse.org/Hudson-ci/Meet_Hudson#What_is_Hudson.3F

Freeman, E., & Freeman, E. *Entwurfsmuster von Kopf bis Fuß.* O'Reilly.

Google Group. (2010). *Privotal Github*. Abgerufen am 18. 06 2013 von http://pivotal.github.io/robolectric/release-notes.html

Google Project Hosting. (kein Datum). Abgerufen am 20. 06 2013 von Google Code Robotium: http://code.google.com/p/robotium/

Google Project Hosting Android findbugs. (kein Datum). Abgerufen am 23. 06 2013 von https://code.google.com/p/findbugs-for-android/

greenDao. (kein Datum). Abgerufen am 24. 06 2013 von http://greendao-orm.com/contact-support/

Heisler, Y. (16. 5 2013). *tuaw*. Von http://www.tuaw.com/2013/05/16/ios-and-android-comprised-92-3-of-q1-2013-smartphone-shipments/ abgerufen

IEEE. (1998). Softwaremetrik.

Levent. (kein Datum). *EclipseZone*. Abgerufen am 28. 07 2013 von http://www.eclipsezone.com/articles/pmd/

Liggesmeyer, P. (2009). *Software-Qualität.* Heidelberg: Spektrum Akademischer Verlag.

Medina, R., & Pratmarty, P. (kein Datum). *Git Hub Project*. Abgerufen am 24. 07 2013 von http://www.uispec4j.org/

ormlite. (kein Datum). Abgerufen am 24. 06 2013 von http://ormlite.com/

Preussler, D. (2012). Von Cocktails und Robotern. *android 360* , S. 85-89.

Schlauch, T. (2005). *Kurz & Gut Cobertura.* Software Engineering.

Sirisena, H. (19. 09 2011). *Blogger*. Abgerufen am 21. 06 2013 von http://hariniachala.blogspot.de/2011/09/android-application-ui-testing-with.html

Stefan. (11. 1 2013). *go2andoid*. Von http://www.go2android.de/kampf-der-app-stores-google-play-erreicht-800-000-apps/ abgerufen

Tiefenhäler, R. (17. 05 2013). *Notebookcheck*. Abgerufen am 22. 04 2013 von http://www.google.de/imgres?client=firefox-a&hs=ahr&sa=X&rls=org.mozilla:de:official&biw=1366&bih=620&tbm=isch&tbnid=5vnF9XJfGWpJOM:&imgrefurl=http://www.notebookcheck.com/Smartphones-Android-und-iOS-dominieren-Windows-Phone-ueberholt-BlackBerry-OS.92850.

Tutorialspoint. (kein Datum). Abgerufen am 24. 06 2013 von http://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm

Vogel, L. (17. 07 2013). *Vogella*. Abgerufen am 23. 06 2013 von http://www.vogella.com/articles/Findbugs/article.html

Wall, T. (kein Datum). *sourceforege.net*. Abgerufen am 23. 06 2013 von http://abbot.sourceforge.net/doc/overview.shtml

Walton, L. (kein Datum). *sourceforge*. Abgerufen am 21. 06 2013 von http://eclipse-metrics.sourceforge.net/

wikipedia. (4. 4 2013). Von http://de.wikipedia.org/wiki/%C3%84quivalenzklassentest abgerufen

wikipedia. (19. 5 2013). Von
http://de.wikipedia.org/wiki/Android_%28Betriebssystem%29#Architektur abgerufen

11 Abbildungsverzeichnis

12 Tabellenverzeichnis

13 Anhang

A Installation der ImkerApp

Im folgenden wird erläutert, wie die *ImkerApp* ohne Probleme genutzt werden kann.

Getting Started

Um das Projekt starten zu können werden folgende Anwendungen benötigt:

- Eclipse Indigo (3.7)
- Android SDK

Download

Zunächst einmal muss das Android SDK[25] heruntergeladen werden. War dieses erfolgreich, findet man im Ordner *eclipse* die entsprechende *eclipse.exe*, welche dann mit einem Doppelklick gestartet werden kann. Für den Emulator müssen noch die notwendigen Android Pakete heruntergeladen werden. Über den Menüpunkt *Window → Android SDK Manager* können diese Pakete heruntergeladen werden. Für *ImkerApp* wird mindestens die Android Version 2.2 (API8) benötigt.

AVD

Zur Ausführung der *ImkerApp* muss ein Android Simulator (Android Virtual Device) erstellt werden, sofern kein externes Gerät verwendet wird. Über den Menüpunkt *Window → Android Virtual Device Manager* kann ein neues AVD erstellt werden. Mit einem Klick auf *New* gelangt man in ein Dialogfenster, in welchem man die entsprechenden Einstellungen vornehmen kann.

[25] http://developer.android.com/sdk/index.html

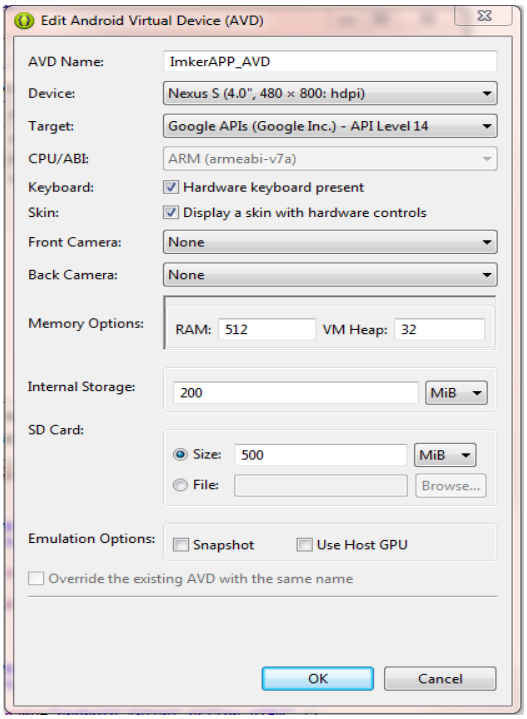


Abbildung 13-1 Erstellung AVD

In der Abbildung 13-1 sind die Einstellungen eines AVD für *ImkerApp* zu sehen. Neben dem Namen des AVD muss das Target, also die Android Version, angegeben werden. Diese muss mindestens der minimalen Version entsprechen. Zudem ist es wichtig, dass eine SD Card mit mindestens 500 MiB eingerichtet wird.

Ausführung der Anwendung

Nachdem alle Plugins erfolgreich installiert wurden und ein AVD eingerichtet wurde, kann der Sourcecode des Projektes *ImkerApp* mit Hilfe des Android SDK als eigenständiges Projekt importiert werden.

Die Anwendung kann unter *Run As → Android Application* gestartet werden.

Ausführung der Testprojekte

Damit die Testprojekte erfolgreich ausgeführt werden können, müssen Testdaten vorhanden sein. Daher müssen in der Klasse *DatabaseManager* aus dem Package *imkerapp.database* folgende Variablen auf *true* gesetzt werden:

```
private static boolean debug = true;
private boolean debugdata = true;
```

B Entwicklungsumgebung

Die Applikation *ImkerApp* wurde mit Hilfe der Entwicklungsumgebung eclipse und dem Android Development Kit (ADT) Plugin entwickelt. Zusammen bieten sie alle Funktionen, welche man zum Entwickeln benötigt.

Das ADT erlaubt sehr schnell und einfach Android Applikationen zu erstellen. Es bietet die Möglichkeit Benutzeroberflächen zu erstellen, externe Bibliotheken einzubinden oder die Nutzung eines Debug-Monitors. Zudem ist ein Informationsmonitor, das sogenannte Log, verfügbar, über den Informationen über den Emulator oder das angeschlossene Gerät abgerufen werden.

Zu alle dem existiert noch das Android SDK, welches der Hauptteil der Entwicklung von Applikationen ist. Hier befinden sich alle benötigten API-Bibliotheken und Entwicklungstools, welche zum Erstellen, Debuggen und Testen der Applikation nötig sind.