Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Bachelorarbeit

im Studiengang
BCS

Refaktorisierung eines Smartphone-basierten Content Management Systems

von
Torsten Knauf

Erstprüfer: Prof. Dr. Manfred Kaul
Zweitprüfer: Dr. Leif Oppermann

Eingereicht am: 22.04.2014

Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Unterschrift Sankt Augustin, den

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

Designer	TOTEM.Designer
DRY	Don’t-Repeat-Yourself
Fraunhofer-FIT	Fraunhofer-Institut für Angewandte Informationstechnik FIT
NE	Natural Europe
OCP	Open-Closed-Prinzip
Scout	TOTEM.Scout
SRP	Single-Responsibility-Prinzip
TOTEM	Theories and Tools for Distributed Authoring of Mobile Mixed Reality Games

1 Einleitung

Diese Arbeit wurde im Rahmen meiner Tätigkeit als studentische Hilfskraft beim Fraunhofer-Institut für Angewandte Informationstechnik FIT im Bereich der Mixed Reality geschrieben.

Als erstes wird auf ortsbasierte Spiele eingegangen und auf die Notwendigkeit von Werkzeugen, die beim Erstellungsprozess unterstützen.

Die in dieser Arbeit behandelten TOTEM-Werkzeuge sind solche Werkzeuge. Sie wurden in einem deutsch-französischen Forschungsprojekt entwickelt und bestehen aus dem TOTEM.Designer und dem TOTEM.Scout. Mit ihnen können ortsbasierte Daten gesammelt und strukturiert werden. Der Designer ist ein webbasiertes Content Management System. Der Scout ist eine Android App und dient als mobiler Datensammler für den Designer. Die Verbindung einer mobilen Komponente mit einer Desktop-Umgebung in einem Werkzeug ist ein Alleinstellungsmerkmal der TOTEM-Werkzeuge in diesem Kontext.

Im Rahmen des mehrjährigen iterativen Rapid-Prototyping-Erstellungsprozesses wurde der Source Code der TOTEM-Werkzeuge allerdings immer schwerer zu warten und zu erweitern. Dies gilt insbesondere für die kompliziertere Android App, den Scout.

Die Motivation dieser Arbeit ist es daher, den Source Code des Scouts zu verbessern. Dafür wird zuerst darauf eingegangen, was guter Source Code ist, und anschließend das Refaktorisieren vorgestellt. Das Refaktorisieren hat das Ziel, den Source Code zu verbessern, ohne dabei sein nach außen zu beobachtendes Verhalten zu verändern.

Eine Ist-Zustands-Analyse des Source Codes auf der Basis der zuvor erarbeiteten Theorie kommt jedoch zu dem Fazit, dass ein Neuschreiben einfacher ist als ein Refaktorisieren. Daher wurde der Source Code in dieser Arbeit neu geschrieben.

Der neugeschriebene Source Code umfasst anschließend nur noch gut 5.000 Zeilen statt wie zuvor über 15.000. Es wird des Weiteren gezeigt, dass die bemängelten Schwachpunkte im neugeschriebenen Source Code behoben wurden. Dazu ist es wichtig, dass die TOTEM-Werkzeuge zukünftig leicht um neue Datentypen erweitert werden können. Deswegen wurde für die Datentypen auf das Open-Closed-Prinzip geachtet. Eine anschließend einfach durchgeführte Erweiterung um ein Indoor-Raummodell bekräftigt die Qualität des neugeschriebenen Source Codes.

In der abschließenden Diskussion wird auf aufgetretene Schwierigkeiten mit Unit-Tests eingegangen, sowie verschiedene Verbesserungsvorschläge für die TOTEM-Werkzeuge vorgeschlagen.

2 Kontext der Arbeit

In diesem Kapitel werden die TOTEM-Werkzeuge vorgestellt. Sie sind Autoren-Werkzeuge um ortsbasierte Daten zu erstellen und zu strukturieren. Sie wurden im Kontext von ortsbasierten Spielen entwickelt. Daher werden zum Einstieg zuerst drei ortsbasierte Spiele vorgestellt. Anschließend wird die Notwendigkeit von Werkzeugen, die beim Erstellungsprozess von ortsbasierten Spielen unterstützen, erörtert.

2.1 Drei Beispiele für ortsbasierte Spiele

Ortsbasierte Spiele sind eine Form der Mixed Realitiy. Ortsbasiert bedeutet, dass die Spiele von der realen Position eines Spielers, meist einer GPS-Koordinate, abhängig sind. Mixed Reality bedeutet, dass die reale Umgebung mit einer Virtuellen vermischt wird (Milgram und Kishino 1994). Dies wird in Abbildung 1 verdeutlicht, in welcher der Bereich der Mixed Reality zu sehen ist.

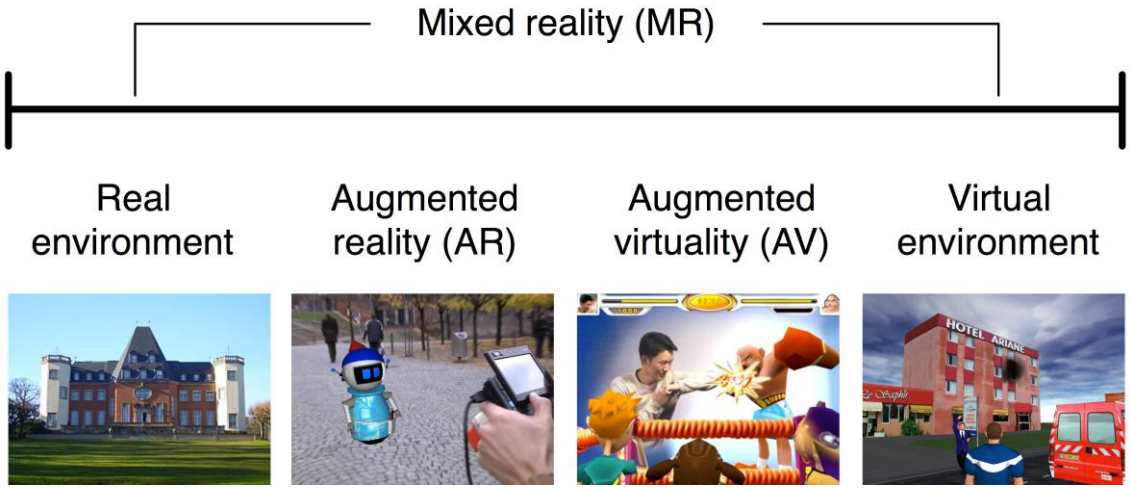

Abbildung 1. Mixed Reality (Feng 2011, S. 2)

Die Mixed Reality setzt auf der realen Umgebung auf, umspannt die Augmented Reality und die Augmented Virtuality und geht fast bis zur rein virtuellen Umgebung. In der Augmented Reality werden virtuelle Objekte in die reale Umgebung projiziert. In der Augmented Virtuality werden reale Objekte in die virtuelle Umgebung integriert.

Folgend werden nun drei Beispiele für ortsbasierte Spiele gegeben.

2.1.1 Geocaching

Ein bekanntes Beispiel für ein ortsbasiertes Spiel ist Geocaching. Es gibt zurzeit über 2 Millionen Geocaches und über 6 Millionen Spieler[1].

[1] (Groundspeak)

Abbildung 2. Geocache in Dänischenhagen

Ziel des Spiels ist es, einen an einer GPS-Koordinate versteckten Schatz zu finden. Zum Spielen wird nur ein GPS fähiges Gerät sowie die GPS-Koordinate des Schatzes benötigt. Abbildung 2 zeigt einen Geocache-Schatz, der in einem Straßenschild versteckt ist. Er besteht aus einem Logbuch, in das sich eingetragen werden kann.

Um einen neuen Geocache zu erstellen, das heißt, einen Schatz für andere zu verstecken, wird nur eine GPS-Koordinate benötigt. Das Teilen des Geocaches mit der Community erfolgt dann durchs Hochladen der GPS-Koordinate auf eine Webseite. Das Hochladen wird von einem einfachen Formular zum Eintragen der GPS-Koordinate und optional einige weitere Beschreibungstexte unterstützt.

2.1.2 Epidemic Menace

Ein Beispiel für ein sehr komplexes (Prototyp-) Spiel ist Epidemic Menace. Es wurde im Rahmen des EU-Forschungsprojekts IPCity[2] am Fraunhofer-FIT entwickelt und im abgeschirmten Bereich auf dem Fraunhofer Campus in Sankt Augustin gespielt.

[2] („IPCity“)

Abbildung 3. Epidemic Menace (Fischer, Lindt, und Stenros 2006, S. 2)

Abbildung 3 zeigt einige Ausschnitte, die während eines Spiels aufgenommen wurden. Im Spiel treten zwei Teams gegeneinander an, die versuchen eine sich ausbreitende Epidemie zu bekämpfen und dabei möglichst viele Punkte sammeln wollen. Die Epidemie breitet sich in Abhängigkeit von Wind und Temperatur aus und ist an bestimmten GPS-Koordinaten zu bekämpfen. Das Spiel besteht aus einem Zusammenspiel zwischen verschiedenen Geräten wie Handys, Computer, und speziellen Augmented Reality Geräten. (Montola, Stenros, und Waern 2009, S. 159ff.)

Das Erstellen und Verändern eines Spiels, sowie das Synchronisieren der Daten zwischen den verschiedenen Geräten während eines Spiels, ist dementsprechend sehr aufwendig.

2.1.3 Time Warp

Time Warp[3] wurde ebenfalls im Rahmen des EU-Forschungsprojekts IPCity am Fraunhofer-FIT entwickelt und ist ein Beispiel für ein später entwickeltes ortsbaisertes Spiel.

[3] („IPCity » Time Warp“)

Abbildung 4. Time Warp („IPCity » Time Warp“)

Es wird in Köln gespielt. In Anlehnung an die Heinzelmännchen[4] wurden in der Zukunft kleine Roboter erschaffen, die den Menschen bei der alltäglichen Arbeit helfen sollen. Die Roboter entwickeln jedoch ein Selbstbewusstsein und fliehen in verschiedene Zeiten an verschiedene Orte in Köln. Die Aufgabe der Spieler ist es, die Roboter in den Zeiten an den verschiedenen Orten zu finden und wieder einzufangen. Als Alternative können die Roboter, welche stets um Mitleid bitten, auch befreit werden. Abbildung 4 zeigt ein Foto der Mixed Realität aus Time Warp.

Abbildung 5. Time Warp mit Ultramobile PC („IPCity » Time Warp“)

Zum Spielen wird ein Ultramobile PC benötigt, wie in Abbildung 5 zu sehen ist. Die virtuellen Objekte, beispielsweise der Roboter aus Abbildung 4, sind durch die Kamera des Ultramobile PCs sichtbar.

[4] („Wikipedia - Heinzelmännchen“)

Das Erstellen von Time Warp ist ebenfalls deutlich aufwändiger als das Erstellen eines Geocaches. Es müssen Grafiken erstellt werden, wie beispielsweise die von den virtuellen Heinzelmännchen, diese mit GPS-Koordinaten verknüpft werden und eine Spiellogik implementiert werden.

2.2 Vereinfachung des Erstellungsprozesses ist nötig

Die Technologien, die ortsbasierte Spiele ermöglichen, entwickeln sich rasant weiter. So mussten zum Beispiel für Epidemic Menace (2006) benötigte Hardware teilweise auf dem Rücken mitgetragen werden, wie in Abbildung 3 unten rechts zu sehen war. Für Time Warp (2010) war bereits ein Ultramobile PC ausreichend, wie in Abbildung 5 zu sehen war. Heutzutage bieten sogar alltägliche Smartphones viele Möglichkeiten.

Die Nutzbarkeit und Bedeutung von neuen Technologien entwickeln sich im Allgemeinen daran, wie die Nutzer sie adoptieren und verwenden. Deswegen ist es wichtig, viele ortsbasierte Spiele zu entwickeln, die von Spielern getestet und ausprobiert werden. (Hull, Clayton, und Melamed 2004)

Wie unter anderem die Spiele Epidemic Menace und Time Warp gezeigt haben, ist das Erstellen solcher Spiele jedoch sehr aufwändig. Daher ist eine Vereinfachung des Erstellungsprozesses nötig.

2.2.1 Demokratisierung des Erstellungsprozesses

Durch Werkzeuge wie WordPress[5] kann heutzutage ohne großen Aufwand und Fachwissen eine Webseite erstellt und veröffentlicht werden. Dadurch werden dezentralisiert und unabhängig voneinander viele Webseiten erstellt und erprobt, die von Anwendern auf ihren Wert getestet werden. Die genaue Anzahl von aktuell verfügbaren Webseiten ist nicht bestimmbar, aber es gibt auf jeden Fall mehrere Milliarden Webseiten[6]. Dass jeder Inhalt für das World Wide Web erstellen kann, wird auch als *Demokratisierung des World Wide Webs* bezeichnet.

Idealerweise ließe sich durch Werkzeug Unterstützung des Erstellungsprozesses von ortsbasierten Spielen ebenfalls eine *Demokratisierung* erreichen, so dass jeder ortsbasierten Spiele entwickeln und veröffentlichen kann. Dadurch könnten ortsbasierte Spiele, genau wie Webseiten, parallel und unabhängig voneinander erstellt und von Anwendern auf ihren Wert getestet werden.

Die Idee dieser Analogie stammt aus (Hull, Clayton, und Melamed 2004). Sie entwickelten dazu Prototyp-Werkzeuge, die das Erstellen von ortsbasierten Spielen unterstützen. Bei der Evaluation dieser Werkzeuge kommen sie unter anderem zu dem Ergebnis, dass die

[5] („WordPress“)
[6] („The size of the World Wide Web“)

Spielersteller 75% der Zeit mit dem Sammeln, Bearbeiten und Strukturieren von Inhalt beschäftigt waren.

2.2.2 Inhalte vor Ort sammeln

Die meiste Zeit beim Erstellen von ortsbasierten Spielen wird also für das Erstellen von Inhalt benötigt. Das Sammeln und Bearbeiten von Inhalt für ortsbasierte Spiele ist jedoch teuer und dauert lange, da viele verschiedene Experten benötigt werden. So werden nicht nur Domain-Experten für das Spiel benötigt, sondern auch Experten für das Design, für die Technologie, sowie ggf. professionelle Schauspieler, welche alle bezahlt und koordiniert werden müssen. (Weal u. a. 2006)

Im Sinne der zuvor thematisierten Demokratisierung des Erstellungsprozesses sind daher Werkzeuge nötig, die beim Erstellen von Inhalt unterstützen. Wenn die Spielersteller ihren Inhalt alleine erstellen können, hat dies folgende Vorteile:

- Es ermöglicht ein schnelleres Erstellen von Inhalten.
- Die Inhalte können von den Spielentwicklern jederzeit sofort überarbeitet, verändert und ergänzt werden.
- Es reduziert nicht nur die Zeit sondern auch die Kosten da weniger Experten notwendig sind.

Bereits (Hull, Clayton, und Melamed 2004) kam des Weiteren zu dem Ergebnis, dass Werkzeuge notwendig sind, die vor Ort einsetzbar sind. Denn es muss unter anderem auf GPS-Ungenauigkeiten eingegangen werden und geschätzt werden, wie lange es dauert von einem Ort zum Nächsten zu gehen. Dies lässt sich in einer Desktop-Umgebung nicht realistisch berücksichtigen.

(Weal u. a. 2006) verstärkt diese Notwendigkeit, denn nur vor Ort können echte Eindrücke, welche die Spieler ja nachempfinden sollen, gesammelt und aufgenommen werden. Auch kommen oft spontan vor Ort die besten Ideen.

Die Desktop-Umgebung ist jedoch unersetzlich, da zum Beispiel das Text Eingeben auf kleinen mobilen Geräten mühselig ist und sich Bilder, Videos und Audios auch besser in einer Desktop-Umgebung bearbeiten lassen. Daher sollten die mobilen Komponenten die Desktop-Umgebung nicht ersetzen, sondern unterstützen.

2.2.3 Anforderungen an Werkzeuge

Im folgenden Kasten sind abschließend die Anforderungen zusammengefasst, die Werkzeuge zur Unterstützung des Erstellungsprozesses erfüllen sollten.

Anforderungen an Werkzeuge zum Erstellen von Inhalten:

- Die Bedienung der Werkzeuge muss leicht und ohne Programmiererfahrung funktionieren.
- Es kann eine Ortsbestimmung erfolgen.
- An den bestimmten Orten können virtuelle Daten platziert werden, wie Texte, Bilder oder Audios.
- Zusätzlich können Notizen zu bestimmten Orten erstellt werden.
- Der gesammelte Inhalt kann kategorisiert und anderen zur Verfügung gestellt werden.
- Der gesammelte Inhalt kann ggf. in einem neuen Kontext wiederverwendet werden.
- Es können Inhalte vor Ort gesammelt und anschließend in einer Desktop-Umgebung überarbeitet und ergänzt werden.

2.3 Iterative Rapid-Prototyping-Entwicklung der TOTEM-Werkzeuge

Fraunhofer-FIT hat, wie bereits erwähnt, die Spiele Time Warp und Epidemic Menace entwickelt. Ein Fazit aus der sehr aufwändigen Entwicklung dieser Spiele ist, dass Werkzeuge zur Unterstützung des Erstellungsprozesses benötigt werden.

In einer ersten Iteration wurden dann zunächst für die konkret zu entwickelnden Spiele Tidy City[7] und Portal Hunt[8] in einem Rapid-Prototyping-Prozess die jeweils spezifische Tidy City- und Portal Hunt-Autoren-Werkzeuge erstellt. Mit diesen konnte Inhalt für die jeweiligen Spiele gesammelt und strukturiert werden.

In einer zweiten Iteration wurden diese Werkzeuge zu den TOTEM-Werkzeugen generalisiert, damit sie auch außerhalb der konkreten Spiele eingesetzt werden können. Generalisiert bedeutet, dass sich mit den Werkzeugen beliebige Datenstrukturen sammeln lassen.

In weiteren Projekten wurden dann Spiele wie Zwergenwelten[9], diverse Summer School Games[10] und Natural Europe[11] entwickelt. Dazu wurden die generalisierten TOTEM-Werkzeuge ggf. um benötigte Funktionalität, wie beispielsweise eine WLAN-basierte

[7] („Tidy City“)
[8] („Portal Hunt“)
[9] („Zwergenwelten“)
[10] („TOTEM Summer School“)
[11] („Natural Europe“)

Ortsbestimmung[12], erweitert. Anschließend wurden die so gewonnenen spezialisierten und erweiterten Werkzeuge wieder in die TOTEM-Werkzeugen generalisiert.

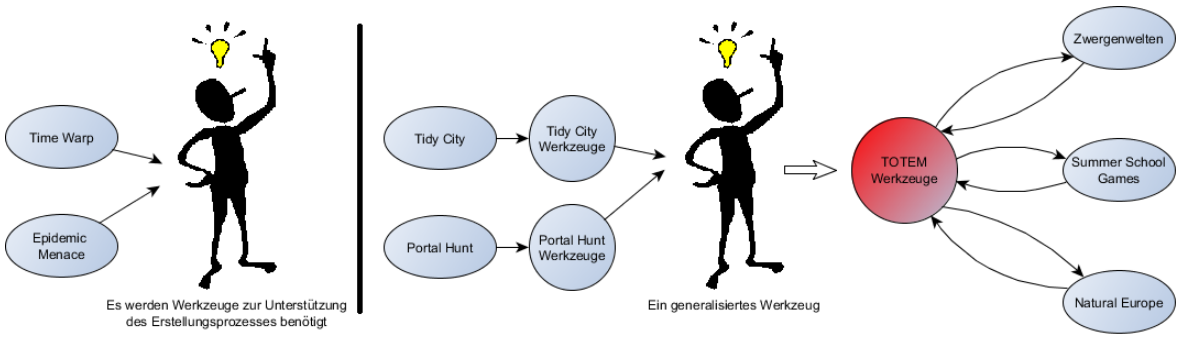

Abbildung 6. Drei Stufen der Entwicklung der TOTEM-Werkzeuge

Die Entwicklung der Spiele lässt sich somit in drei Stufen zusammenfassen, die in Abbildung 6 zu sehen sind. Im ersten Schritt wurden Spiele ohne die Unterstützung von Werkzeugen erstellt. Da dies sehr aufwendig war, wurden Werkzeuge zur Unterstützung erstellt, was der zweite Schritt ist. Um nicht für jedes Spiel aufwendig ähnliche Werkzeuge erstellen zu müssen, ist dann der dritte Schritt diese Werkzeuge zu einem generalisierten Werkzeug zusammenzufassen, welches flexibel eingesetzt werden kann. Dieser dritte Schritt wird in den TOTEM-Werkzeugen umgesetzt.

2.4 Die TOTEM-Werkzeuge

Diesem Kapitel liegt das Paper (Jurgelionis u. a. 2012) zugrunde, welches die TOTEM-Werkzeuge vorstellt.

Die TOTEM-Werkzeuge (Theories and Tools for Distributed Authoring of Mobile Mixed Reality Games) sind Werkzeuge, die das Sammeln und Strukturieren von ortsbasierten Inhalten, insbesondere für Spiele, unterstützen und sind ein Content Management System. Unter einem Content Management System wird im Allgemeinen „eine Software zur gemeinschaftlichen Erstellung, Bearbeitung und Organisation von Inhalten“ („Wikipedia - Content-Management-System“) verstanden.

[12] (S. Feng 2011)

In Abschnitt 2.2.3 wurden Anforderungen an Werkzeuge zur Unterstützung des Erstellungsprozesses von ortsbasierten Spielen aufgestellt. Der unten stehende Kasten listet die Hauptfunktionalitäten der TOTEM-Werkzeuge auf, in denen sich diese Anforderungen wiederfinden.

Hauptfunktionalitäten der TOTEM-Werkzeuge:

- Die Bedienung erfolgt komplett über ein Benutzerinterface, wodurch keine Programmierkenntnisse erforderlich sind. So können Spielersteller ohne Hilfe Inhalt erstellen, bearbeiten und strukturieren.
- Durch eine Android App (TOTEM.Scout) kann mit einem Smartphone vor Ort Inhalt gesammelt werden. Durch eine Webseite (TOTEM.Designer) kann in einer Desktop-Umgebung der gesammelte Inhalt bearbeitet und ergänzt werden. Das Synchronisieren der Daten zwischen dem Smartphone und der Webseite erfolgt über das Internet per Knopfdruck.
- Für bestimmte Arten von Inhalten, die sich in ihrer Struktur wiederholen, können Templates erstellt werden, von denen beliebig viele Instanzen erzeugt und mit unterschiedlichen Daten gefüllt werden können.
- Es können auch unstrukturierte Daten als Notizen gesammelt werden.
- Die gesammelten Daten können durch unterschiedliche Technologien, wie durch GPS-Koordinaten oder NFC-Tags, an Orte gekoppelt werden.

Das Besondere der TOTEM-Werkzeuge sind dabei folgende vier Punkte, welche in keinem anderen Werkzeug vereint sind:

1. Die TOTEM-Werkzeuge sind generalisiert, das heißt, dass die Art des zu sammelnden Inhalts beliebig variiert und strukturiert werden kann. Sie können somit vielseitig eingesetzt werden.
2. Es gibt eine Rollenverteilung zwischen Autoren und Inhalt-Sammlern. Autoren können die zuvor genannten Templates erstellen und bearbeiten. Autoren und Inhalt-Sammler können Instanzen zu diesen Templates erstellen und diese mit konkreten Daten füllen. So kann jeder, dem die Autoren die Templates zur Verfügung gestellt haben, Inhalt sammeln.

3. Durch die vollständig in einander integrierten Designer und Scout können mit den gleichen Werkzeugen in einer Desktop-Umgebung und vor Ort Inhalte gesammelt werden, ohne dass manuelle Synchronisation nötig ist.
4. Der gesammelte strukturierte Inhalt kann beim Designer per Knopfdruck in Form von Web-Standards, beispielsweise als JSON-Datei, exportiert werden.

2.4.1 Metapher von Pebbles, Shapes und Marbles

Für das Konzept der Templates und Instanzen, welches für Leute ohne Kenntnisse in der Objektorientierten Programmierung nicht intuitiv zu verstehen ist, gibt es die anschauliche Metapher von Pebbles, Shapes und Marbles. Die Pebbles, welche sich in den gesammelten Notizen widerspiegeln, sind als ungeschliffene Rohdaten zu verstehen. Durch eine Shape können diese zu Marbles geschliffen werden. Die Marbles, wie der Name bereits vermuten lässt, entsprechen dann den Spielobjekten. Durch die Shape (das Template) wird festgesetzt, woraus die Marbles bestehen. In einer Shape kann eine beliebige Anzahl von Properties wie Texte oder Zahlen, Media-Daten, wie Bilder oder Videos und Orte, wie GPS Positionen oder NFC-Tags, enthalten sein. Abbildung 7 zeigt das Konzept, wie aus einer Pebble, durch eine Shape, eine Marble wird.

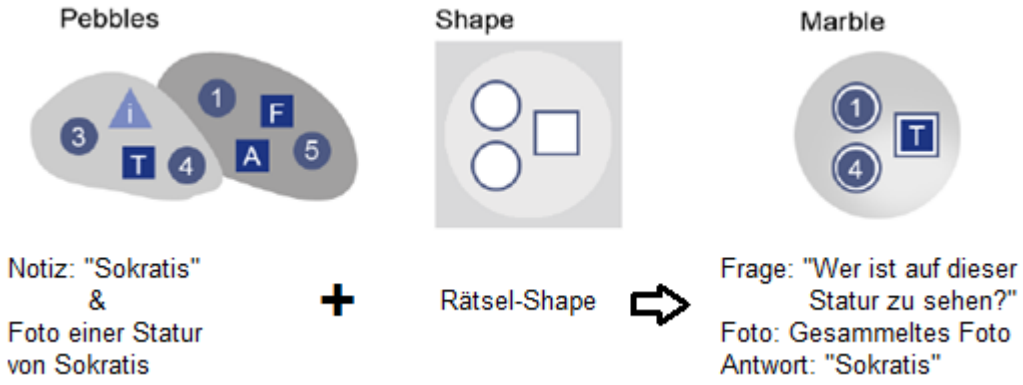

Abbildung 7. Pebbles, Shapes und Marbles (Nach Jurgelionis u. a. 2012, S. 4)

2.4.2 Die Aufgabenverteilung zwischen dem Designer und dem Scout

Abbildung 8 zeigt die Aufgabenverteilung zwischen dem Scout und dem Designer, welche anschließend genauer vorgestellt werden.

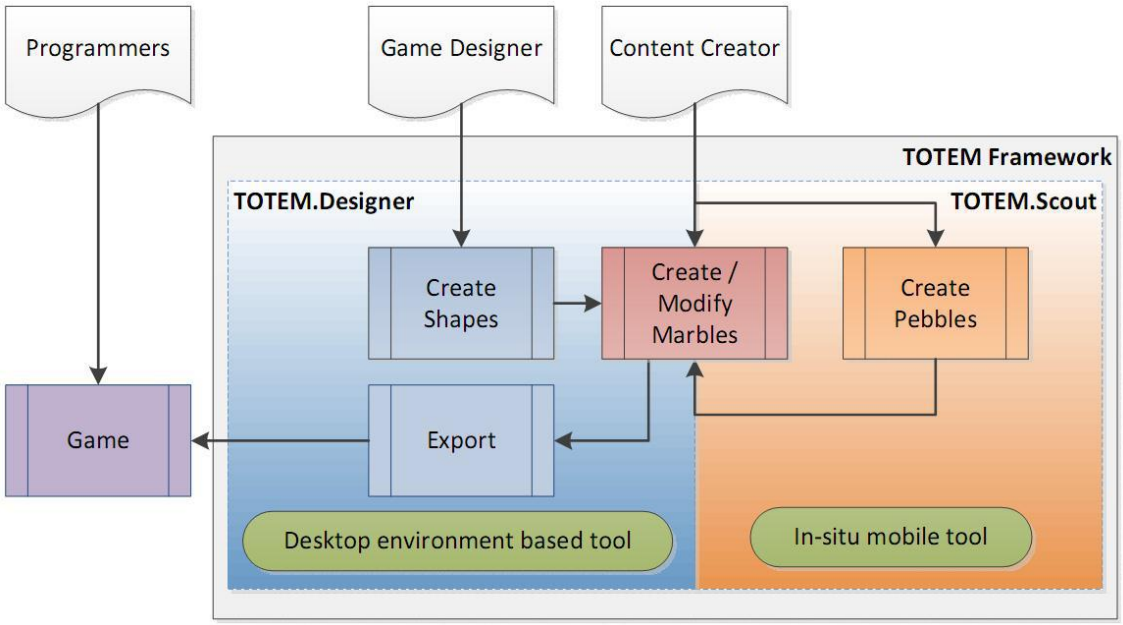

Abbildung 8. Aufgabenverteilung zwischen dem Designer und dem Scout (Jurgelionis u. a. 2012, S. 9)

Die Spiellogik wird von einem Programmierer erstellt und findet außerhalb der TOTEM-Werkzeuge statt. Der Inhalt für die Spiele wird dann durch die TOTEM-Werkzeuge erstellt. Ein Spieldesigner erstellt dazu die Shapes auf dem Designer, die den benötigten Inhalt beschreiben. Die Shapes können dann auf den Scout geladen und vor Ort mit Daten gefüllt werden. Anschließend werden die gefüllten Daten, welche Marbles entsprechen, wieder auf den Designer geladen, wo diese noch bequem aufgearbeitet werden können, bevor sie in das Spiel exportiert werden.

2.4.3 Der TOTEM.Designer

Der Designer ist eine mit dem Webframework Django[13] geschriebene Webanwendung.

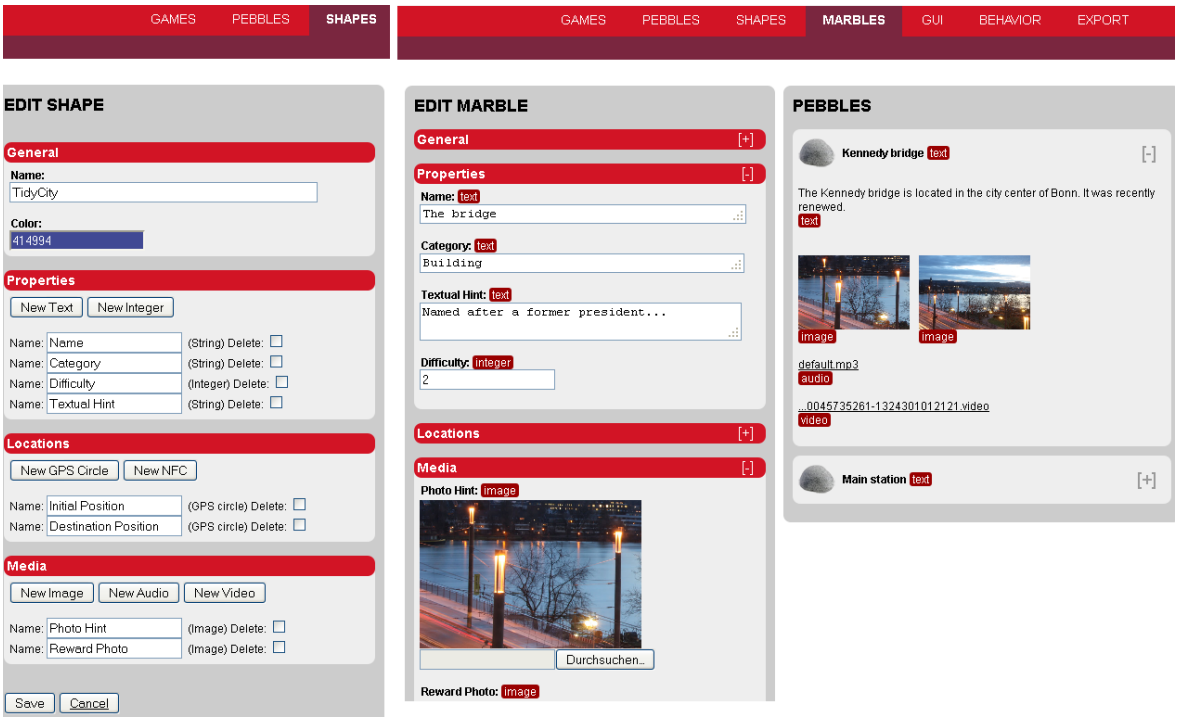

Abbildung 9. Links das Erstellen einer Form, rechts das Füllen einer Form (Jurgelionis u. a. 2012, S. 6)

Abbildung 9 zeigt zwei Screenshots vom Designer anhand des Beispiels von Tidy City[14]. Links wird eine Shape erstellt, welche die Rätsel beschreibt, aus denen das Spiel besteht. Ein Rätsel besteht aus einem Namen, einer Kategorie, einem Tipp und einem Schwierigkeitsgrad. Des Weiteren werden durch GPS-Koordinaten eine Anfangsposition und eine Endposition festgesetzt. Dazu gibt es ein Bild als Tipp und ein Lösungsbild. Wird nun eine Marble nach dieser Shape erstellt, was rechts zu sehen ist, werden alle benötigten Felder angezeigt. Diese können dann einfach gefüllt werden. Rechts werden auch die Pebbles, die per Drag and Drop reingezogen werden können, angezeigt.

2.4.4 Der TOTEM.Scout

Der Scout ist eine Android App, mit der vor Ort Daten gesammelt werden. Sie dient als Datensammler für den Designer. Auf ihr können keine Shapes erstellen werden, um das Sammeln klar von dem Erstellen zu trennen (s. Abbildung 8). Shapes und bereits auf dem Designer erstellte Marbles können vom Designer heruntergeladen werden. Die heruntergeladenen Shapes können dann mit Daten vor Ort durch die App gefüllt werden, oder bereits erstellte Marbles ergänzt bzw. geändert werden. Genauso können Pebbles als Notizen mit beliebigen Daten gespeichert werden. Alle gesammelten Pebbles und

[13] („Django“)
[14] („Tidy City“)

Marbles können abschließend per Knopfdruck wieder auf den Designer hochgeladen werden.

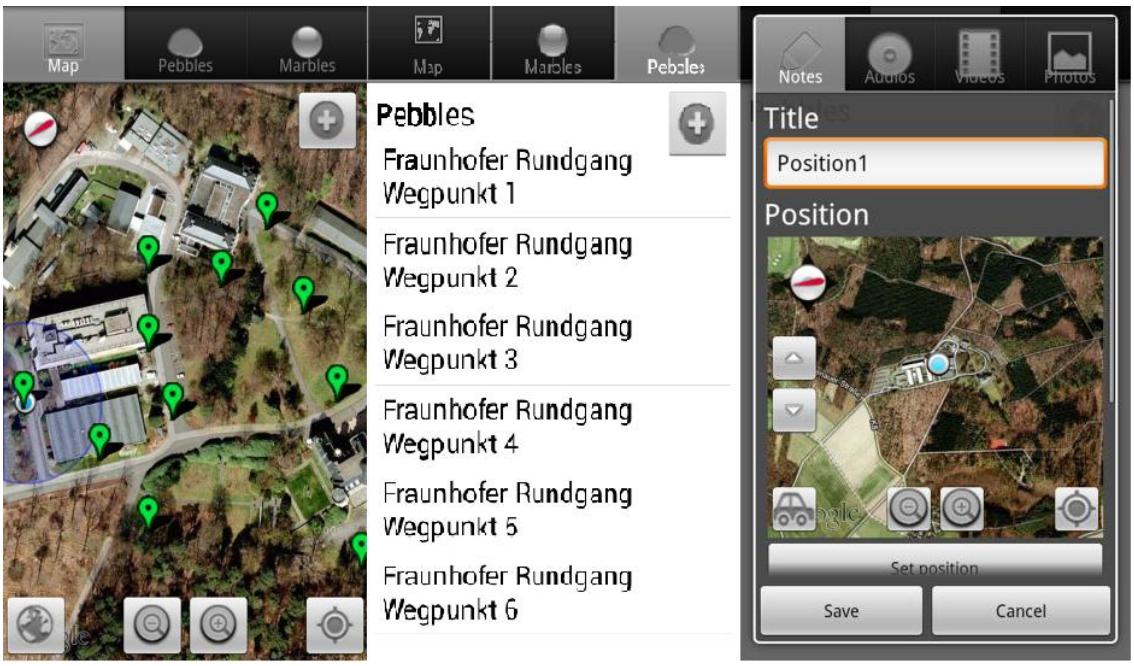

Abbildung 10. Links die Google Karte, in der Mitte die Liste aller gesammelten Pebbles, Rechts das Bearbeiten eines Pebbles (Nach Jurgelionis u. a. 2012, S. 7)

Abbildung 10 zeigt ein Screenshot des Scouts. Links ist die integrierte Google Maps zu sehen, die die eigene Position und die Position der bereits erstellten Datensätze anzeigt. Google Maps kann in der Straßenansicht oder wie hier in der Satellitenansicht angezeigt werden. In der Mitte ist eine Liste mit den erstellten Pebbles zu sehen. Wird eine Pebble angeklickt, führt dies zum rechten Bild, in dem die ausgewählte Pebble editiert werden kann. Die Marbles sind analog zu den Pebbles.

2.5 Motivation für die Refaktorisierung des TOTEM.Scouts

Die iterative Rapid-Prototyping-Entwicklung der TOTEM-Werkzeuge (s. 2.3) wurde von verschiedenen Entwicklern nacheinander vorgenommen. Dazu standen die Entwicklern oft unter Zeitdruck. Deswegen war die Qualität des Source Codes nie oberste Priorität. Es hat auch nie eine Refaktorisierung stattgefunden, um die Qualität des Source Codes zu verbessern.

Der Source Code des Scouts hat darunter aus zwei Gründen besonders gelitten:

1. Im Gegensatz zum Designer ist beim Scout Multithreading erforderlich. Operationen, die länger dauern können, sollten nicht im UI-Thread ausgeführt werden, um die Benutzererfahrung nicht einzuschränken. Aus diesen Gründen

beendet Android sogar eine App, wenn der UI-Thread zu lange beschäftigt war[15]. Threading ist jedoch komplex und benötigt somit auch viel Zeit.

2. Der Scout muss dem Datenmodell des Designers entsprechen, da er als Datensammler für diesen dient. Somit muss beim Scout auch Wissen über den Designer vorhanden sein. Das einarbeiten in den Designer und den Scout erfordert wiederum mehr Zeit.

Dementsprechend war mein Einarbeiten in den Source Code des Scouts sehr mühsam. Ein einfaches Erweitern war mit dem vorhandenen Design auch nicht gut zu bewerkstelligen. Beim Einarbeiten wurden des Weiteren Bugs entdeckt.

Deswegen soll der Scout zuerst refaktorisiert werden, bevor in einer weiteren Iteration die Integration der, für das Natural Europe Projekt spezialisierten, Natural Europe-Werkzeugen umgesetzt wird. Dazu wird im folgenden Kapitel zuerst überlegt, was guter Source Code ist, und anschließend das Refaktorisieren vorgestellt. Im vierten Kapitel wird dann die Verbesserung des Source Codes thematisiert.

[15] (s. „Android Developers - Keeping Your App Responsive“)

3 Theorie

Dieses Kapitel ordnet die Wichtigkeit von gutem Source Code für ein Softwareprodukt ein und definiert, was guter Source Code ist. Im Anschluss wird erläutert, dass auch guter Source Code gewartet und regelmäßig überarbeitet werden muss, damit er gut bleibt. Das Refaktorisieren beschäftigt sich mit diesem Überarbeiten und Strukturieren, um die Qualität vom Source Code sicherzustellen. Für das Refaktorisieren ist das automatische Testen wichtig, welches auch behandelt wird. Somit wird in diesem Kapitel die Theorie für den praktischen Teil im vierten Kapitel vorgestellt.

3.1 Einordnung der Wichtigkeit von gutem Source Code

Bevor konkret auf Kriterien für guten Source Code eingegangen wird, sei hier angeschnitten, dass ein gutes Softwareprodukt aus viel mehr als aus gutem Source Code besteht. In Abhängigkeit von seiner Rolle hat jeder Produktbeteiligter andere Kriterien, die wichtig sind. So ist für einen Benutzer vor allem die User Experience wichtig, für einen Manager die Wirtschaftlichkeit und nur für den Softwareentwickler ist der Source Code primär wesentlich. (Simon, Seng, und Mohaupt 2006, S. 18ff.)

Abbildung 11 zeigt durch die sechs Aspekte von Softwarequalität nach ISO 9126 eine weitere Sicht. Obwohl für alle Aspekte guter Source Code zweifelsfrei die Grundlage bildet, beinhaltet kein Begriff primär das Ziel guten Source Code zu erhalten.

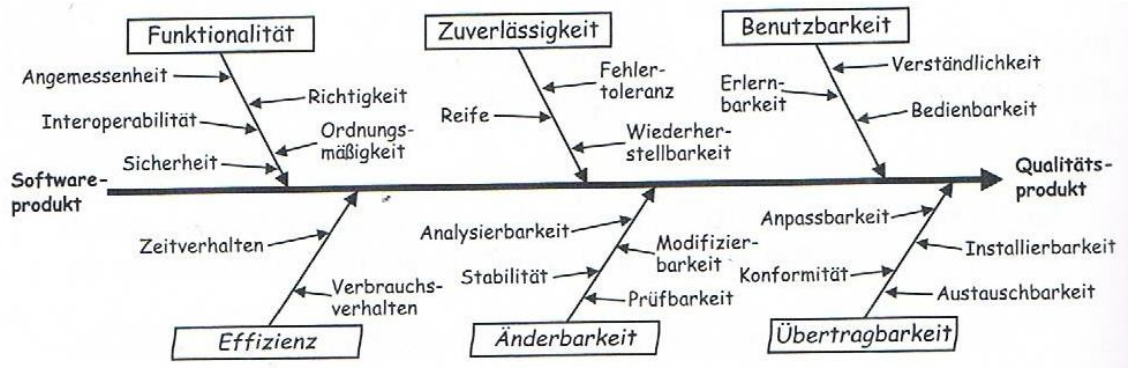

Abbildung 11. Aspekte einer Qualitätssoftware nach ISO 9126 (Vigenschow 2010, S. 18)

Eine weitere Sichtweise auf Softwarequalität ist die ebenfalls in ISO 9126 definierte *quality in use*. Sie bewertet die Qualität eines Softwaresystems auf den drei aufeinander aufbauenden Schichten der Inneren-, Äußeren- und Nutzungsqualität aus Abbildung 12.

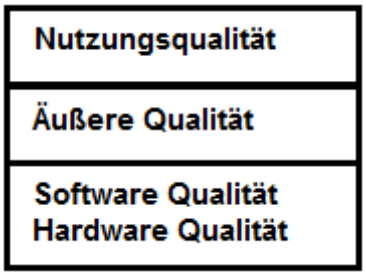

Abbildung 12. Quality in use (Nach Vigenschow 2010, S. 21)

Der Entwickler beschäftigt sich hauptsächlich mit der inneren Qualität. Diese setzt sich aus der Hardware und der Software, welche sich im Source Code widerspiegelt, zusammen und bleibt für den Anwender verborgen. Die äußere Qualität ist das Zusammenspiel der inneren Qualität und lässt sich beispielsweise in der Antwortzeit des Systems messen. Obwohl die Nutzungsqualität auf den anderen beiden Schichten aufbaut, spiegelt sie letztendlich den tatsächlichen Wert wider. In ihr wird gemessen, wie zufrieden alle Stakeholder, insbesondere der Anwender, sind und welcher Mehrwehrt sich durch die Software einstellt. (Vigenschow 2010, S. 20f.)

Somit lässt sich durch den kurzen Einschub feststellen, dass es viel mehr Kriterien für ein gutes Softwareprodukt als den Source Code gibt. Der Source Code wird sogar von den meisten Stakeholdern wie den Kunden und Manager nicht aktiv wahrgenommen. Jedoch bauen fast alle Kriterien auf den Source Code auf, weswegen er ein wichtiger Aspekt für ein gutes Softwareprodukt ist.

3.2 Was ist guter Source Code

Hier werden verschiedene Ansätze, um guten Source Code zu definieren, erläutert. Darauf aufbauend wird zum Schluss eine Definition für guten Source Code gegeben.

3.2.1 Bekannte Programmierer gefragt

In (Martin 2009, S. 32ff.) wurden einige bekannte und erfahrene Programmierer gefragt, was guter Source Code ist. Die am häufigsten genannten Punkte sind:

- Source Code ist gut, wenn er funktioniert wie geplant und erwartet.
- Der Aufgabenbereich des Source Codes ist durch Tests abgedeckt, die alle bestanden werden.
- Er ist einfach zu verstehen.
- Die Abhängigkeiten sind minimal, so dass er leicht zu warten und erweitern ist.

3.2.2 Kriterien für guten Source Code

Die zuvor genannten Punkte sind zwar einleuchtend, aber größtenteils abstrakt. Folgend werden einige konkrete Punkte genannt, durch welche die Ziele der leichten Verständlichkeit, Wartung und Erweiterung erreicht werden können. Sie sind aus (Martin 2009) entnommen.

Für die Verständlichkeit sollte auf die **Formatierung** geachtet werden. Bei der Programmiersprache Java gehören dazu unter anderem eine einheitliche Einrückung, Konventionen für das Platzieren von Klammern, sowie eine maximale Zeilenlänge.

Die **Namenswahl** für Klassen, Funktionen und Variablen ist für die Verständlichkeit ebenfalls sehr wichtig. Aus dem Namen einer Klasse, Funktion oder Variable sollte sofort hervorgehen, warum es sie gibt, was sie ausführt und wie sie zu benutzen ist.

Dazu sollte **alles kurz sein**. Je größer etwas ist, umso größer ist seine natürliche Unübersichtlichkeit und so kleiner die Chance, bestimmte Sachen wiederverwenden bzw. erweitern zu können.

Ein weiteres wichtiges Kriterium ist das **Single-Responsibility-Prinzip** (SRP). Es besagt, dass eine Klasse genau eine Verantwortung haben sollte und somit auch nur genau einen Grund haben sollte, sich zu ändern. Dadurch entstehen viele kleine, aufgeräumte Klassen, in denen der Entwickler nicht lange suchen muss, um zu finden, was er braucht. Des Weiteren erleichtert dies das Verändern und Erweitern dieser Klassen. *Switch*-Anweisungen widersprechen diesem Prinzip naturgemäß, da sie normalerweise *n* mögliche Aufgaben haben.

Genauso gilt für Funktionen, dass sie nur eine Aufgabe erfüllten sollte. Sie erfüllen genau eine Aufgabe, wenn sie eine **einheitliche Abstraktionsebene** haben. Ein Beispiel für eine hohe Abstraktionsebene ist ein Funktionsaufruf wie *getAnObject()*, ein Beispiel für eine mittlere Abstraktionsebene ist *object.setSomething()* und ein Beispiel für eine tiefe Abstraktionsebene ist *.append(„\n“)*. Funktionen mit unterschiedlichen Abstraktionsebenen sind schwieriger zu verstehen, da zwischen den Abstraktionsebenen gewechselt werden muss und eventuell nicht sofort klar ist, ob etwas zu dem allgemeinen Konzept gehört, oder bloß ein Implementierungsdetail ist.

Das **Don’t-Repeat-Yourself-Prinzip** (DRY) sollte ebenso eingehalten sein. Sich wiederholender Source Code oder duplizierte Logik im Source Code bietet mehr Angriffsfläche für potentielle Fehler. Auch wird der Source Code dadurch unnötig größer, worunter wiederum die Übersichtlichkeit leidet. Jede Änderung muss darüber hinaus an allen Stellen durchgeführt werden, was geradezu dazu einlädt, eine Stelle zu vergessen, oder irgendwo einen Fehler einzubauen.

3.2.3 Metriken

Eine Metrik ist eine Maßfunktion *M*, die einem Objekt oder Ereignis *x* eine Zahl *y* zur Beschreibung zuordnet, also $M: x \mapsto y, y \in \mathbb{R}$. Bei einer Metrik für Software sind die Objekte oder Ereignisse *x* Elemente von Source Code wie beispielsweise Softwaremodule.

Als Ideal sollte eine Metrik für Source Code die tatsächliche Qualität des Source Codes widerspiegeln. Laut dem IEEE Standard 1061 wird das dadurch erreicht, dass die Metrik direkt ist. Darunter wird verstanden, dass sie ausschließlich von dem zu messenden Qualitätsaspekt abhängig ist. Die Problematik davon soll an folgendem Beispiel, der als direkte Metrik angegeben *„mean time to failure“*, gezeigt werden. Diese misst die zu erwartende Zeit, bis ein Fehler auftritt. Sie ist offensichtlich davon abhängig, ob als Zeit die aktive Benutzungszeit, die Zeit seit der Einführung der Software oder nur die Zeit, in der unterschiedliche Ausführungspfade der Software benutzt werden, genommen wird. Auch können Fehler in Abhängigkeit von ihrer Auswirkung unterschiedlich gezählt werden, ob zum Beispiel ein Fehler erst bei einem Programmabsturz gezählt wird, oder bereits ein leichtes Flackern in der Benutzungsoberfläche. Dazu wird die Anzahl der Fehler wahrscheinlich in Abhängigkeit der Benutzergewohnheiten und der laufenden Betriebsumgebung variieren. Damit ist diese Metrik von mehreren Kriterien abhängig. Aufgrund dieser Problematik stellt sich die Frage, wie gut Metriken die tatsächliche Qualität widerspiegeln können. (Kaner und Bond 2004)

Als anderes Beispiel, um die Qualität des Source Codes zu messen, sei die allseits bekannte Metrik *„Lines of Code“* genannt. Zum einen stellt sich die Frage, ob Kommentare und Leerzeilen mitgezählt werden sollten. Zum anderen stellt sich die Frage der Interpretation: Sind besonders viele Zeilen ein guter Wert, weil sie von viel Sorgfalt zeugen, oder besonders wenig, da sie ein Indiz für eine elegante Lösung sind? Dementsprechend wäre der Source Code besser, wenn er viel Code-Duplizierung beinhaltet, oder sich sämtliche *Null*-Abfragen spart, was beides offensichtlich nicht der Fall ist.

Fowler geht sogar so weit zu behaupten, dass die menschliche Intuition immer besser als jedes System von Metriken ist. (Fowler 2005, S. 67)

3.2.4 Smells

Smells sind Strukturen im Source Code, die Indizien für schlechten Source Code sind. Sie sind keine mathematisch exakten Kriterien, sondern beruhen auf Intuition und Erfahrungswerten. Sie spiegeln sich in Verletzungen von Kriterien für guten Source Code wider und gelten somit als Startpunkte für Refaktorisierungen. (Fowler 2005, S. 67)

In (Fowler 2005, Kap. 3) und (Martin 2009, Kap. 17) sind über 50 verschiedene Smells angegeben und erläutert. Folgend werden davon einige rausgesucht, die in Abschnitt 4.1.2 bei der Ist-Zustands-Analyse des TOTEM.Scouts aufgefallen sind. In den Klammern hinter den Namen befindet sich dabei ein Literaturverweis für genauere Informationen. Es sei erwähnt, dass eine Stelle, die einen Smell enthält, oft mehrere verschiedene Smells aufweist.

Duplizierter Code (Fowler 2005, S. 68): Dieser Smell ist die „Nummer Eins in der Gestanksparade“ (Fowler 2005, S. 68). Er ist das genaue Gegenteil vom DRY-Prinzip und hat all die Nachteile, die dieses Prinzip zu verhindern versucht.

Faule Klasse (Fowler 2005, S. 77): Jede Klasse muss verstanden und gewartet werden und hat dazu einen Overhead. Nicht alle Klassen leisten genug, um dies zu kompensieren und sollten dementsprechend ggf. in bereits bestehende Klassen integriert werden.

Große Klasse (Fowler 2005, S. 71): Große Klassen sind das Gegenteil von faulen Klassen. Sie haben eine natürliche Unübersichtlichkeit und widersprechen meist dem SRP. Sie sind „hervorragende Brutstätte für duplizierten Code, Chaos und Tod“ (Fowler 2005, S. 71).

Lange Methode (Fowler 2005, S. 69f.): Je Länger eine Methode ist, umso schwerer ist es, diese zu verstehen. Auch ist die Wahrscheinlichkeit der Wiederverwendbarkeit größer, je kleiner seine Funktionalität ist. Durch ein Zerlegen in mehrere kleinere Methoden mit guter Namenswahl lässt sich außerdem oft der folgend erläuterte Smell „Kommentare“ entfernen.

Kommentare (Fowler 2005, S. 82) und (Martin 2009, S. 85ff.): Kommentare sind an sich keine Smells. Jedoch werden sie oft verwendet, wenn der Source Code schlecht und ohne Kommentare nicht verständlich ist. Der Source Code an sich wird durch Kommentare jedoch nicht besser. So führen Kommentare oft zu Stellen, die refaktorisiert werden sollten.

Ein echter Smell hingegen ist auskommentierter Source Code. Programmierer mögen denken, dass er noch wichtig sein könnte und ihn deshalb niemals Löschen, obwohl er den Source Code aufbläht und somit unübersichtlich macht. Durch ein Versionskontrollsystem kann der entsprechende Source Code, statt auskommentiert, einfach gelöscht und ggf. wenn nötig wieder hergestellt werden.

Zur Abgrenzung dazu gute Kommentare können welche mit juristischer Natur sein. Durch Kommentare können auch bestimmte Aspekte, die eventuell sonst nicht angemessene Beachtung finden, hervorgehoben werden. So sorgen zum Beispiel

TODO-Kommentare dafür, dass nötige Änderungen, die gerade nicht umgesetzt werden können, nicht verloren gehen.

Allerdings müssen auch diese Kommentare gewartet werden. Veraltete oder irreführende Kommentare erschweren das Arbeiten mit dem Source Code, anstatt es zu vereinfachen. Sie blähen ihn des Weitern auf, was wiederum eine natürliche Unübersichtlichkeit mit sich zieht.

Divergierende Änderungen (Fowler 2005, S. 72f.): Nach dem SRP sollte jede Klasse nur einen Grund haben, sich zu ändern. Hat sie dazu mehrere Gründe, führt dies oft zu Komplikationen. Dieser Smell ist verwandt mit dem Smell „Große Klasse“ und tritt so oft in Kombination mit diesem auf.

Schrotkugel herausoperieren (Fowler 2005, S. 73): Dieser Smell ist in gewisser Weise das Gegenteil vom Smell „Divergierende Änderungen“ und beschreibt das Phänomen, dass eine Änderung viele weitere verstreute Änderungen zur Folge hat. Es ist dabei leicht eine Stelle zu übersehen, was ein Fehlverhalten des Programms zur Folge hat.

***Switch*-Befehl** (Fowler 2005, S. 76): *Switch*-Befehle haben oft Duplikation zur Folge. Sie müssen meist an verschiedenen Stellen wiederholt werden und wenn eine neue Bedingung hinzugefügt wird, muss dies an allen Stellen geschehen. Für aneinander gereihte *if-else*-Konstruktionen gilt genau das gleiche.

Falsche Zuständigkeit (Martin 2009, S. 349): Ist der Source Code nicht dort, wo ihn der Leser erwartet, muss er ihn erst suchen und oft anschließend den Kontext zwischen verschiedenen Stellen wechseln. Die Java Packages sind zur Organisation ein sehr grobes Mittel. Erwartet der Leser eine Funktionalität bereits im falschen Package, führt dies zu langem Suchen.

3.2.5 Patterns

In einem Kapitel über guten Source Code dürfen Patterns nicht fehlen. Patterns beruhen genauso wie Smells auf Erfahrungen. Sie spiegeln allerdings guten Source Code wieder und können als *how to* gelesen werden. Ein Pattern beschreibt ein Problem, welches in einem Kontext erfahrungsgemäß regelmäßig vorkommt, sowie ein Beispiel für eine gute Lösung des Problems. So entsteht eine eigene Sprache für Problemlösung und erfahrene Programmierer können ihr Wissen durch das Niederschreiben von Patterns, zu immer wieder auftretenden Problemen, weitergeben. (Gamma u. a. 2007, S. 2ff.)

Wichtig dabei ist, dass Patterns als *how to* oder auch Lösungsangebote zu verstehen sind, durch die guter Source Code und eine gute Software-Architektur erreicht werden können. Das blinde Umsetzten ist jedoch nicht vorteilhaft. Anhang 1 zeigt ein „Hello

Word“-Programm, welches dieses an sich simple Programm mittels dem Factory Method[16]-, dem Singelton[17]- und dem Strategy[18]-Pattern umsetzt, was offensichtlich absurd ist.

Patterns haben keinen Selbstzweck, sondern dienen, um guten Source Code zu erzeugen. Wird kein Kriterium von gutem Source Code, wie beispielsweise Vermeidung von Code-Duplizierung, durch ein Pattern verbessert, so hat das Pattern auch keinen Mehrwert. Eventuell macht es den Source Code sogar unnötig kompliziert. Auch gibt es nicht die eine exakte Muster-Implementierung von einem Pattern. Ein Pattern beschreibt das Konzept, aber kann auf viele individuell angepasste Weisen umgesetzt werden. Manchmal ist sogar ein minimalisiertes Pattern vorzuziehen. Bringt die volle Umsetzung von einem Pattern keinen Mehrwert, so ist dies auch nicht nötig. (Kerievsky 2004, S. 24ff.)

3.2.6 Abschließende Definition zu gutem Source Code

Basierend auf diesem Kapitel komme ich zur folgenden simplen, aber dennoch umfassenden Definition zu gutem Source Code:

Source Code ist gut, wenn er keine unnötigen Erschwernisse beinhaltet, um seine Aufgaben zu erledigen.

„Seine Aufgaben“ ist dabei sehr differenziert zu betrachten. Wird der Source Code nur einmalig benutzt, reicht es, wenn er einmal funktioniert, denn alles andere wäre Ressourcenverschwendung. Soll er allerdings öfters und wahrscheinlich zukünftig benutzt und eventuell auch weiterentwickelt werden, so sind viel mehr Kriterien zu beachten. So sollte er einfach veränder- und erweiterbar sein. Alles was das erschwert, ist dann Zeugnis für einen schlechten Source Code und lässt sich meist in Smells wiederfinden (s. Abschnitt 3.2.4). Aspekte die dies unterstützen sind Bausteine für gutem Source Code (s. Abschnitt 3.2.2). Patterns sind beispielsweise solche Bausteine, die jedoch auch falsch verbaut sein können (s. Abschnitt 3.2.5).

Abschließend lässt sich noch festhalten, dass es keine mathematisch exakten Kriterien für guten Source Code gibt (s. Abschnitt 3.2.3). Eine gute Annäherung bilden eine gute Intuition sowie vor allem Erfahrungswerte. Deswegen sind für das später behandelte Refaktorisieren auch nicht Metriken sondern Smells die Startpunkte.

3.3 Source Code altert

Jeder Programmierer wird den Vorsatz kennen, bei einem neuen Programm alles von Anfang an sauber und ordentlich gestalten zu wollen. Genauso wird jeder Programmierer

[16] (Gamma u. a. 2007, S. 315ff.)
[17] (Gamma u. a. 2007, S. 127ff.)
[18] (Gamma u. a. 2007, S. 107ff.)

das Gefühl kennen, dass es dann in der Realität nicht funktioniert hat, was in Abbildung 13 sehr schön karikiert wird.

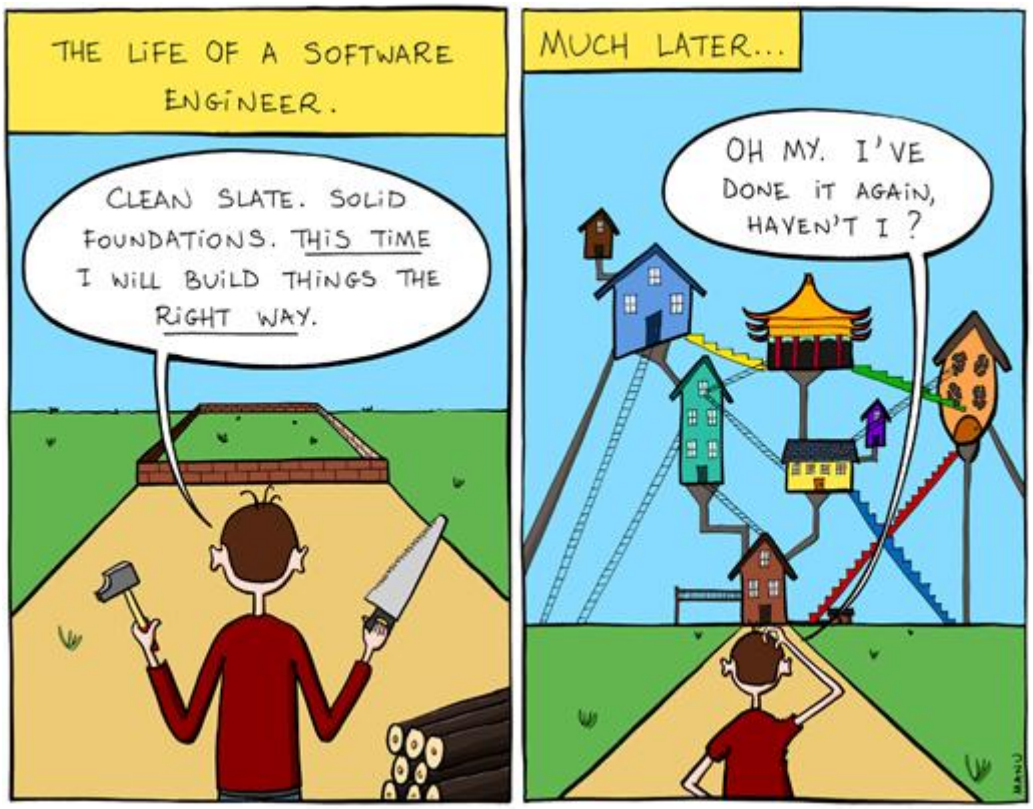

Abbildung 13. Source Code altert („Code-Inside Blog“)

Dieses Phänomen wird auch oft als *historisch gewachsenes System bezeichnet*. Der Grund für dieses Phänomen ist, dass Source Code altert. Früher wurde davon ausgegangen, dass Software in einem sogenannten *Big Design Upfront* vollständig vor dem Source Code Schreiben entworfen werden kann und anschließend nur noch umgesetzt werden muss. Moderne agile Entwicklungsprozesse gehen hingegen davon aus, dass dies nicht möglich ist. Stattdessen wird von einem evolutionären Lernprozess ausgegangen. Es gibt nicht den einen Entwurf, sondern es werden immer neue Erfahrungen und Ansätze gemacht, wie ein Problem elegant gelöst werden kann. Darüber hinaus soll die Software Aufgaben in einem bestimmten Kontext lösen. Die Aufgaben werden jedoch oft zu Projektbeginn nicht exakt verstanden und können auch mit der Zeit variieren. Dazu kann sich der Kontext, durch beispielweise Gesetzesänderungen, wandeln. Des Weiteren passieren Änderungen der Schnittstellen von Drittanbietern, auf die die eigene Software aufbaut und dementsprechend reagiert werden muss. All dies hat nicht vorgeplante Änderungen zur Folge, die in der Regel mit einem Strukturverlust verbunden sind. Dadurch werden Änderungen immer weniger gradlinig und somit teurer. Gibt es keine Gegenwirkung, führt dies zu dem Phänomen aus Abbildung 13. (Lippert und Roock 2004, S. 11ff.)

Man könnte meinen, dass es ausreicht, die in Abschnitt 3.2.4 vorgestellten Smells einfach zu vermeiden. In der Praxis ist dies jedoch nicht realistisch. Entwickler stehen oft unter Zeitdruck, so dass eine schnelle und oft nicht Architektur-gerechte Änderung stattfindet. Bei mehreren Entwicklern passiert es dazu oft, dass einer getreu zu dem entwickelt, was er verstanden hat, ein anderer Entwickler das Design jedoch anders ausgelegt hatte. Des Weiteren kann die bereits im vorherigen Abschnitt angesprochene Problematik, dass die Architektur nicht einmalig entworfen wird, sondern kontinuierlich angepasst werden muss, sowie das Technologieänderungen von außen auftreten, zwangsweise Smells erzeugen. (Lippert und Roock 2004, S. 76f.)

3.4 Refaktorisieren

Hier wird sich mit der Restrukturierung von Source Code beschäftigt, welche das Ziel hat, die Qualität des Source Codes zu verbessern. Das Restrukturieren ist unter anderen wegen der zuvor erläuterten Alterung von Source Code nötig.

3.4.1 Definition Refaktorisierung

Der Refaktorisierung liegt folgende Definition zu Grunde:

> **„Eine Änderung an der internen Struktur einer Software, um sie leichter verständlich zu machen und einfacher zu verändern, ohne ihr beobachtbares Verhalten zu ändern.“** (Fowler 2005, S. 41)

Die „interne Struktur“ ist dabei mit dem Source Code gleichzusetzten. Es wird also der Source Code geändert. Dabei werden keine neuen Funktionalitäten eingebaut, oder Änderungen vorgenommen, die von einem Anwender der Software bemerkt werden könnten. Das Ziel davon ist es, den Source Code verständlicher und einfacher änderbar zu machen.

In größeren Projekten, die über einen längeren Zeitraum gehen und von verschiedenen Programmierern entwickelt werden, ist es für den Entwicklungsprozess offensichtlich hinderlich, wenn der Source Code nicht leicht zu verstehen und nur schwer zu ändern ist. Damit ist mit der Definition von gutem Source Code aus Abschnitt 3.2.6 ersichtlich, dass das Refaktorisieren in größeren Projekten das Ziel von gutem Source Code hat.

3.4.2 Wann sollte refaktorisiert werden

Dieser Abschnitt beruht auf (Fowler 2005, S. 46f.).

Es sollten keine Extraphasen für das Refaktorisieren eingebaut werden. Genau wie Patterns hat das Refaktorisieren keinen Selbstzweck, sondern es wird refaktorisiert, um einen konkreten Mehrwert zu erreichen.

Refaktorisieren ist zum Beispiel gut beim Einarbeiten und Verstehen von Source Code. Wenn über eine Stelle im Source Code länger nachgedacht werden muss, um zu verstehen, was diese erledigt, kann sofort refaktorisiert werden, damit er leichter verständlich ist. Dadurch werden zukünftige Änderungen leichter und der Nächste wird sich schneller in den Source Code einarbeiten können.

Erschwert das verwendete Design eine geplante Erweiterung, so wird dann auch zuerst das Design refaktorisiert, und anschließend kann die Erweiterung leicht eingebaut werden. Dies vermeidet Design-ungerechte Änderungen, die den Source Code schwerer verständlich machen und oft lässt sich die Erweiterung auf diese Weise schneller realisieren.

Wenn Fehler auftreten, ist dies auch ein Zeichen dafür, dass refaktorisiert werden sollte. Der Source Code war in diesem Fall nämlich offensichtlich nicht gut genug, um zu erkennen, dass er nicht fehlerfrei ist.

Um den schlimmsten Smell „Code Duplizierung“ zu verhindern, gilt die folgende Dreierregel: Wird etwas zum ersten Mal programmiert ist alles gut. Wird dieses oder etwas Ähnliches ein zweites Mal gebraucht, ist eine Wiederholung der Programmierung noch in Ordnung. Tritt es jedoch zum dritten Mal auf, sollte unbedingt refaktorisiert werden, so dass es anschließend nur an einer Stelle implementiert ist.

3.4.3 Wann sollte NICHT refaktorisiert werden

Es gibt nach (Fowler 2005, S. 56) nur zwei Gründe, wieso nicht refaktorisiert werden sollte.

Der erste Grund ist ein anstehender Fertigungstermin. Von Refaktorisierungen wird langfristig profitiert, was bei einem anstehenden Fertigungstermin zu spät sein kann.

Der zweite Grund ist, wenn der vorhandene Source Code so schlecht ist, dass ein Neuschreiben einfacher ist als ein Refaktorisieren. Ein eindeutiges Indiz dafür ist, wenn der Source Code viele Bugs enthält, die nur sehr mühselig zu beheben sind. Denn wenn ein Programm nicht funktioniert, macht ein Refaktorisieren wenig Sinn. Anschließend wäre die innere Struktur zwar besser, aber da das Refaktorisieren nicht das äußere Verhalten ändert, funktioniert es danach genauso wenig.

Ein Kompromiss zwischen Neuschreiben und Refaktorisieren ist das Aufteilen von Source Code in Komponenten. So können funktionierende Komponenten refaktorisiert werden und nur die anderen müssen neu geschrieben werden.

3.4.4 Richtlinien

Beim Refaktorisieren besteht stets die Gefahr neue Fehler in die Software einzubauen, oder das gar bereits funktionierende Software hinterher nicht mehr funktioniert (Lippert und Roock 2004, S. 19). Um diesen Super-GAU zu vermeiden, gibt es Richtlinien, an die sich beim Refaktorisieren gehalten werden sollte.

Als erstes gilt das **automatisierte Testen**, welches im Abschnitt 3.5 noch behandelt wird, als Vorbedingung für das Refaktorisieren. Die automatisierten Tests ermöglichen ohne großen Aufwand nach einer Änderung unbeabsichtigte Seiteneffekte und Fehler zu finden und geben dem Programmierer so die nötige Sicherheit, dass sie keine neuen Fehler eingebaut haben. (Fowler 2005, S 83ff.)

Als weiteren Punkt gibt es die **Metapher der zwei Hüte** des Programmierers und des Refaktorierers, die sich ein Entwickler nacheinander aufzieht. Hat er den Hut des Programmierers auf, konzentriert er sich ausschließlich darauf, dass eine neue Funktionalität funktioniert. Anschließen tauscht er den Hut gegen den des Refaktorisierers und refaktorisiert. Die Metapher der zwei Hüte verdeutlicht, dass entweder ausschließlich darauf geachtet werden sollte, dass etwas funktioniert, oder ausschließlich auf das Refaktorisieren. (Fowler 2005, S. 42f.)

Beim Refaktorisieren sollte dabei in möglichst **kleinen Schritten** vorgegangen werden. Das hat vor allem folgende zwei Vorteile: Zu einem wird ein neuer Fehler dann sofort entdeckt und kann auch sofort behoben werden, da die Stelle wo er liegt, sehr wahrscheinlich an der Stelle ist, die als letztes geändert wurde. Zum anderen ist so stets eine lauffähige Version vorhanden und andere Entwickler müssen nicht auf das Abschließen der Refaktorisierung warten, bzw. ihre Änderung an die der Refaktorisierung anpassen. (Lippert und Roock 2004, S. 19ff.)

Die in Abschnitt 3.2.5 erwähnten **Patterns sind dabei oft Ziele für Refaktorisierungen**, da sie auf bewehrte Lösungen beruhen, was guten Source Code impliziert. (Fowler 2005, S. 103)

Für Anregungen und als Nachschlagwerk, falls mit einer Refaktorisierung nicht weiter gekommen wird, hat Fowler aus seinen Notizen ein Katalog mit Refaktorisierungen erstellt (Fowler 2005, S. 98ff.). Wie Patterns beruhen sie auf seinen Erfahrungen und beschreiben die einzelnen Schritte, wie eine bestimmte Refaktorisierung durchgeführt werden kann. Ein ähnliches Nachschlagwerk hat Kerievsky erstellt. Er beschreibt in seinem Katalog von Refaktorisierungen, wie Pattern in den Source Code eingebaut werden können (Kerievsky 2004, S. 47ff.).

3.4.5 Problematische Stellen

In einigen Fällen ist das Refaktorisieren mit Schwierigkeiten verbunden. Solche Fälle werden hier angeschnitten:

Da das Testen eine Vorbedingung für eine Refaktorisierung ist, sind im Allgemeinen **Stellen, die schwer zu testen sind**, nur mit besonderer Vorsicht zu refaktorisieren.

Refaktorisierungen, die eine **Datenbank** tangieren, sind oft schwierig. Dies liegt zum einen daran, dass die ganze Software oft auf das Datenbankschema aufbaut und sich eine Änderung somit durch den ganzen Source Code zieht. Zum anderen ist die Migration von Daten sehr mühselig und meist mit manueller Arbeit verbunden, was den Aufwand wesentlich erhöht. (Fowler 2005, S. 53) Dazu weisen relationale Datenbanken wesentliche Unterschiede zu objektorientierten Programmen auf, welche unter anderem in Abschnitt 5.2.3 thematisiert werden. Diese Unterschiede erhöhen die Komplexität und erschweren somit das Refaktorisieren. (Lippert und Roock 2004, S. 152ff.)

Grundlegende Design-Entscheidungen zu ändern, kann sich ähnlich wie Datenbank-Änderungen, durch den ganzen Source Code ziehen. Ist dies der Fall, so sollte dies mit äußerstem Bedacht geschehen, da dies der Richtlinie der kleinen Schritte widerspricht und viel Zeit beanspruchen kann. Allgemein ist **bei allem Vorsicht geboten, was sich nicht in kleine Schritte zerlegen lässt**.

Ein weiteres Problem stellt das Ändern von **veröffentlichten Schnittstellen** dar. Veröffentlicht heißt, dass Drittanbieter die Schnittstellen nutzen. Werden solche geändert, müssen die Drittanbieter, auf die oft kein direkter Einfluss ausgeübt werden kann, ihren Source Code an die Refaktorisierungen anpassen. Dementsprechend sollte die alte Schnittstelle nicht ungültig werden, was oft zu Code-Duplizierung führt. (Fowler 2005, S. 53ff.)

Des Weiteren sei erwähnt, dass die Performance oft erst einmal durch das Refaktorisieren leidet. Es werden neue Objekte eingeführt und durch die kleineren Funktionen sind mehr Funktionsaufrufe nötig. Damit verbunden ist allerdings heutzutage nur noch ein marginaler Leistungsverlust. Demgegenüber steht, dass die meiste Zeit in einem kleinen Bereich des Source Codes verbraucht wird. Bei einem gut strukturierten Source Code, lässt sich dieser Bereich, beispielsweise mit der Hilfe eines Profilers, leicht finden. Nur dort wird dann die Performance optimiert. So werden keine personellen Ressourcen verschwendet, um Bereiche auf Kosten der Source Code-Verständlichkeit zu optimieren, die nur minimalen Effekt haben. Stattdessen können gezielt die kritischen Stellen optimiert werden, die am meisten Auswirkung haben. Dies führt im Endeffekt sogar oft zu besseren Ergebnissen. (Fowler 2005, S. 60ff.)

3.4.6 Werkzeugunterstützung

Die Kosten des Refaktorisierens können durch Werkzeugunterstützung reduziert werden. Hier werden zwei verschiedene Arten von Werkzeugunterstützung erörtert.

Werkzeuge können Stellen finden, die wahrscheinlich refaktorisiert werden sollten. Metrics[19] ist ein Plug-In für Eclipse, welches den Source Code nach verschiedenen Metriken analysieren kann, wie beispielsweise die Länge der einzelnen Methoden. Da allerdings Abschnitt 3.2.3 zu dem Schluss gekommen ist, dass Metriken mit Vorsicht zu genießen sind, sind auch die vorgeschlagenen Stellen mit Vorsicht zu betrachten. Natürlich hat ein automatisches Tool keine menschliche Intuition, welche zu bevorzugen ist. Aus diesem Grund wird in dieser Arbeit auf den Einsatz solcher Tools verzichtet.

Die zweite Art von Werkzeugen findet nicht mögliche Stellen für das Refaktorisieren, sondern unterstützt den Entwickler beim Umsetzten bereits gefundener Stellen. Damit solche Werkzeuge in der Praxis eingesetzt werden, sollten sie in die IDE integriert sein, um ein umständliches hin und her Wechseln zu vermeiden. Auch sollten sie schnell sein. Muss der Entwickler warten, so führt er in der Regel die Änderungen lieber manuell aus. Ein wichtiges Feature ist dabei das Zurücknehmen einer Änderung. Bei automatischen Änderungen können auch ungewollte Effekte entstehen. Daher ist eine Rückgängig-Funktion wünschenswert. Auch fördern sie das Refaktorisieren, da der Entwickler sich die Auswirkung von einer Refaktorisierung anschauen kann und wenn sie ihm nicht gefallen, kann er diese einfach zurück nehmen. (Fowler 2005, S. 422f.)

So ermöglicht Eclipse beispielweise das Umbenennen einer Variable oder Klasse und ändert automatisch alle Referenzen mit. Auch wird das automatische Extrahieren von Methoden und Interfaces angeboten. Eine ausführlichere Liste, was Eclipse automatisiert unterstützt, ist in („Refactoring-Tools in Eclipse“) zu finden. Zukünftig könnte sogar das Einbauen von Patterns automatisiert werden. (Lippert und Roock 2004, S. 28)

Des Weiteren kann die Formatierung, die für die Lesbarkeit sehr wichtig ist, durch den Code Formatter in Eclipse individualisiert und automatisch bei jedem Speichern ausgeführt werden[20].

Eine weitere Unterstützung bietet eine Versionsverwaltung, mit der notfalls eine frühere funktionierende Version zurückgeholt werden kann, falls trotz aller Vorsicht nicht reparierbare Fehler eingebaut wurden.

Es sei erwähnt, das auch das automatische Refaktorisieren zu Fehlern führen kann. So kann zum Beispiel das Ändern vom Namen einer Methode zu Problemen führen, obwohl alle direkten Referenzen mit geändert wurden. Ein String der den Methodennamen enthält

[19] („Metrics 1.3.6“)
[20] („Eclipse documentation“)

wird beispielsweise nicht mit geändert. Versucht nun die Reflection API von Java mit diesem String die Methode aufzurufen, so kann sie diese nicht mehr finden. (Fowler 2005, S. 421)

3.5 Automatisierte Tests

Hier wird das automatisierte Testen behandelt. Es wird das JUnit-Test-Framework vorgestellt sowie die Erweiterung für Android. Des Weiteren wird auf Stellen eingegangen, die schwierig zu testen sind. Anschließend wird die Wichtigkeit von Tests für das Refaktorisieren erörtert, und erwähnt, dass auch Tests refaktorisiert werden sollten.

3.5.1 Einführung in das automatisierte Testen

Ein sehr oft genanntes Beispiel für die Notwendigkeit des Testens ist der Jungfernflug der Ariane 5. Sie war eine Trägerrakete, deren Gesamtentwicklungskosten sich auf 5,5 Mrd. € beliefen. In ihr wurden Softwareelemente der Ariane 4 wiederverwendet. Die Ariane 5 war jedoch deutlich Leistungsstärker, weswegen die Flugbahn aufgrund eines Variable-Überlaufs falsch berechnet wurde. Das Sicherheitssystem löste deswegen die Selbstsprengung aus. (Vigenschow 2010, S. 10ff.)

Somit hat die Wiederverwendung von eigentlich bewährtem Source Code zu einer Katastrophe geführt. Die Moral davon ist, dass Software kontinuierlich getestet werden muss.

Unter Software Tests wird folgendes Verstanden:

> **„Unter dem *Test* von Software verstehen wir die stichprobenartige Ausführung eines Testobjekts, die zu dessen Überprüfung dienen soll. Dazu müssen die Randbedingungen für die Ausführung des Tests festgelegt sein. Über einen Vergleich zwischen Soll- und Ist-verhalten wird bestimmt, ob das Testobjekt die geforderten Eigenschaften erfüllt.“** (Vigenschow 2010, S. 26)

Die Verwendung des Begriffes „stichprobenartig“ verdeutlicht, dass durch Tests nicht die Fehlerfreiheit eines Programmes bewiesen werden kann, sondern dass durch diese nur ein Indiz dafür geliefert wird. Dementsprechend sollten zusätzlich manuelle Tests durchgeführt werden (Vigenschow 2010, S. 89f.). Es kann demonstrativ getestet werden, was das Ziel hat, den normalen Programmablauf zu testen, oder destruktiv. Beim destruktiven Testen wird nicht versucht, das System funktional zu nutzen, sondern gezielt Fehler zu finden (Vigenschow 2010, S. 26). Fowler nennt dies „die Rolle des Gegners des Codes spiele[n]“ (Fowler 2005, S. 96).

3.5.2 Das JUnit Test Framework

JUnit gehört der Familie der xUnit Test Frameworks an. 1999 veröffentlichte Beck ein Unit Test Framework für Smalltalk. Beck und Gamma portierten dieses Framework für Java (JUnit). Mittlerweile ist das Framework für viele Sprachen, wie C++ (CppUnit) und Python (PyUnit) verfügbar. Da alle auf die gleiche Architektur aufbauen, sind sie auch als xUnit Test Familie bekannt.

Fowler sagt über die Bedeutung dieses Test Frameworks: *„Never in the field of software development have so many owed so much to so few lines of code"* (Meszaros 2007, S. XIX).

Durch JUnit können Unit-Tests für einzelne Methoden automatisiert werden. Einer der Hauptziele dabei ist die klare Trennung von Produktions- und Testcode, was in Abbildung 14 verdeutlicht wird. Dies kommt dem SRP nach. So wird der Produktionscode nicht zusätzlich aufgebläht und die Tests können unabhängig vom eigentlichen Programm laufen. (Hamill 2005, S. 1f.)

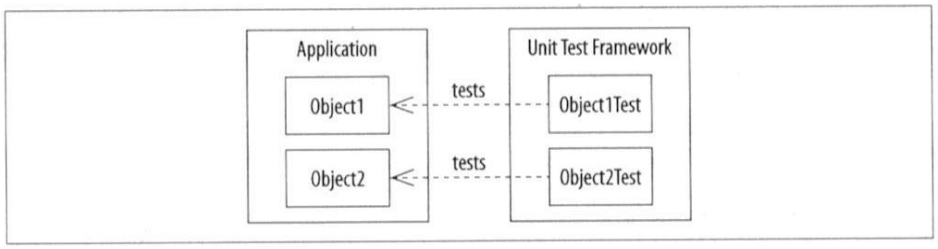

Abbildung 14. Produktions- und Testcode (Hamill 2005, S. 1)

Abbildung 15 zeigt die Architektur des xUnit Frameworks, die auch JUnit zugrunde liegt. Ein Test-Runner führt einen Test aus. Ein Test ist entweder eine Ansammlung von Tests, oder ein konkreter Test. Dies wird durch das Composite Pattern[21] umgesetzt. Jeder konkreter Test ist von einer Fixture abgeleitet. Die Fixture ermöglicht es, für jede Testmethode innerhalb einer Klasse eine *setUp*-Methode, die vor dem Test ausgeführt wird, sowie eine *tearDown*-Methode, die nach dem Test ausgeführt wird, zu definieren. Das Ergebnis eines Tests ist entweder bestanden oder nicht bestanden. Wenn ein Test nicht bestanden wurde, werden genauere Details dazu ausgegeben. (Hamill 2005, S. 20ff.)

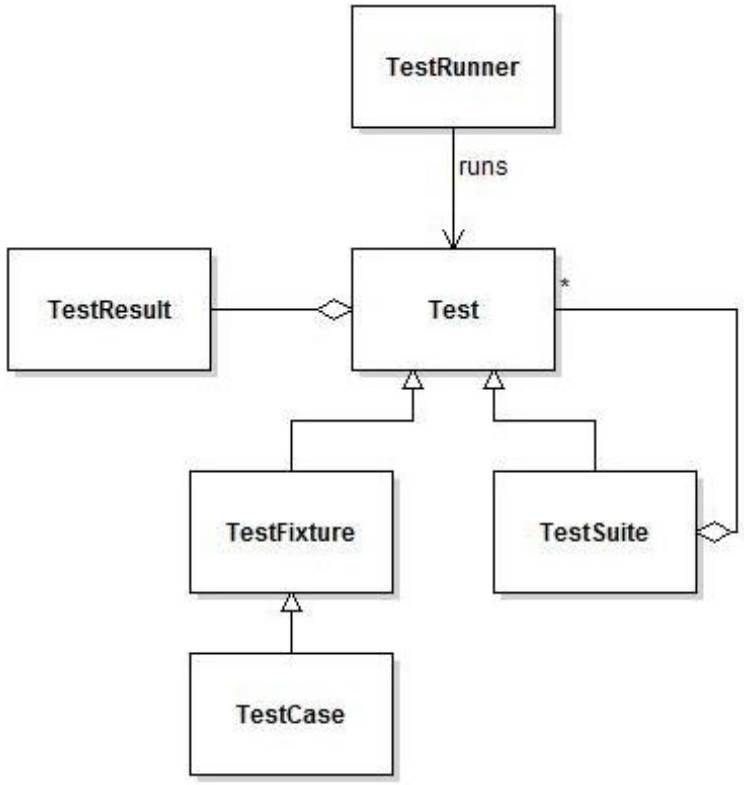

Abbildung 15. xUnit Architektur (Nach Hamill 2005, S. 31)

Testklassen in JUnit erben von der Klasse *TestCase*. Die Fixture besteht aus den leeren Methoden *setUp()* und *tearDown()*, die optional überschrieben werden können. Per Konvention beginnen Testmethoden mit „*test*“, damit sie vom Runner gefunden werden. Die Auswertung der Tests geschieht durch *assert*-Methoden. Einer *assert*-Methode wird das erwartete und das berechnete Ergebnis übergeben. Stimmen die beiden nicht überein, so wird die Testmethode nicht weiter ausgeführt und diese Testmethode wurde nicht bestanden. Optional kann zusätzlich ein String übergeben werden, der angezeigt wird, wenn der Assert nicht bestanden wurde. Die anderen Testmethoden werden weiter ausgeführt. Mehrere Tests können in einer von *TestSuite* abgeleiteten Klasse zusammen-

[21] (Gamma u. a. 2007, S. 163ff.)

geführt werden, indem einzelne Testklasse in der überschriebenen Methode *suite()* hinzugefügt werden. (Hamill 2005, S. 63ff.)

Seit JUnit 4 werden auch Annotations unterstützt. Namenskonventionen können so ersetzt werden. Zum Beispiel müssen Testmethoden nicht mehr per Konvention mit „*test*“ anfangen, sondern können stattdessen auch durch *@Test* annotiert werden. Eine Auflistung der Hauptunterschiede ist in („Junit 3 vs Junit 4 Comparison“) zu finden. JUnit 4 ist dabei abwärtskompatibel zu JUnit 3. Da die Android Erweiterung, auf die im folgenden Abschnitt eingegangen wird, jedoch JUnit 4 nicht unterstützt, wird JUnit 4 hier nicht weiter vertieft und stattdessen JUnit 3 verwendet.

Ab der Version 3 von Eclipse ist JUnit per Default integriert. Zum Build Path muss nur die JUnit-Bibliothek hinzugefügt werden und anschließend können JUnit-Tests aus Eclipse gestartet werden.

Ein Beispiel für einen JUnit-Test ist in Abbildung 16 zu sehen, welcher durch eine *assert*-Methode zwei Strings miteinander vergleicht.

```
public class TOTEMServerTest extends TestCase {

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        // Do something for each test
    }

    @Override
    protected void tearDown() throws Exception {
        // Do something after each test
        super.tearDown();
    }

    public void testXmlrpcFunctionEnum() {
        assertEquals("The enum to String method does not give the right result",
                "addMarbles", TOTEMServer.XmlrpcFunction.ADD_MARBLES.toString());
    }
}
```

Abbildung 16. JUnit-Test-Beispiel

3.5.3 Android JUnit Erweiterung

Android-Komponenten wie Views, die eng mit dem Betriebssystem auf dem Gerät verbunden sind, lassen sich mit JUnit nicht testen, da JUnit reine Java-Klassen sind. Um Android-Komponenten zu testen, bietet Android ein Test-Framework, welches die JUnit Test Cases erweitern. Abbildung 17 zeigt das Konzept dieser Erweiterung.

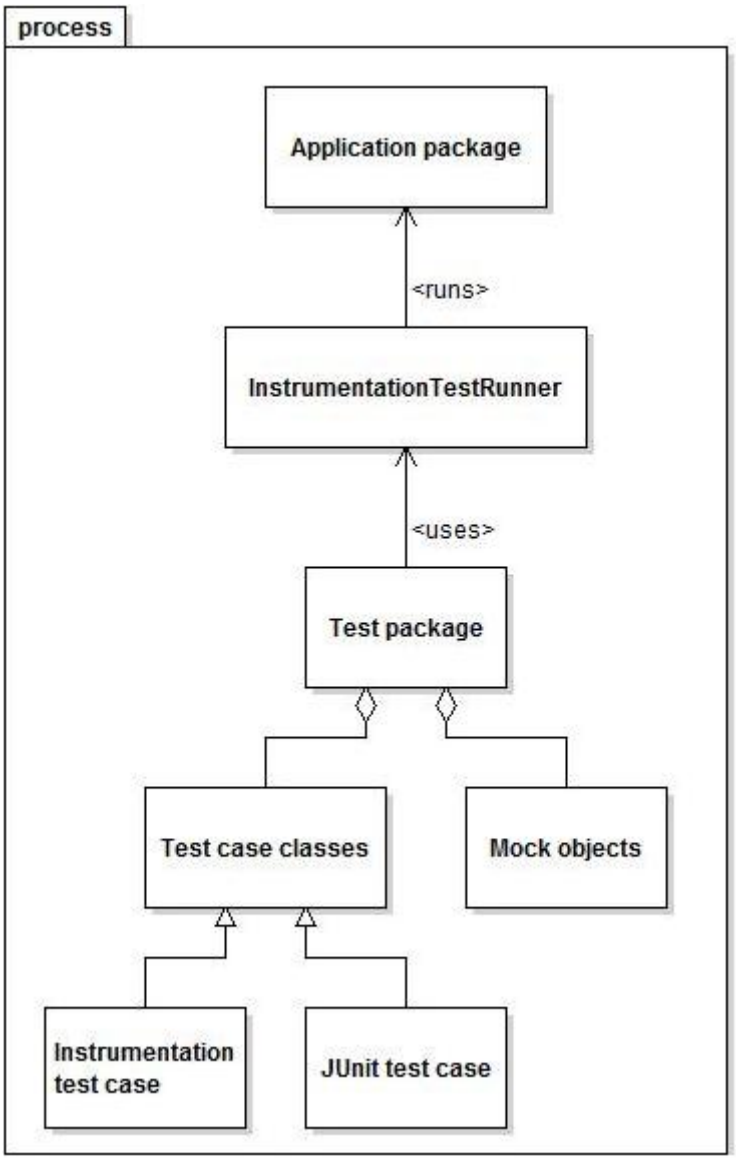

Abbildung 17. Android JUnit Test-Framework (Nach „Android Developers - Testing Fundamentals“)

Ein Testprojekt, welches ein vollständiges Android-Projekt ist, besteht aus Testklassen sowie Mock-Objekten. Die Testklassen sind entweder reine JUnit-Klassen oder Android spezifische JUnit-Klassen, die mittels Instrumentation Zugriff auf Android-Komponenten haben. Ausgeführt wird ein Testprojekt von einem Instrumentation Test Runner. Dieser startet die Android-Anwendung, die getestet wird, auf ein Gerät und kann durch Hooks die Anwendung manipulieren. Zum Beispiel kann so der Lifecycle einer Activity manuell beeinflusst werden. Somit können auch Android-Komponenten automatisiert getestet werden.

Das Testprojekt kann dabei von Werkzeugen der Android SDK erstellt werden. Diese konfigurieren das Testprojekt zu einem Projekt mit allen benötigten Utensilien, wie ein Android Manifest, und aktualisieren automatisch alle Referenzen zum Projekt, so dass dafür kein manuelles Konfigurieren nötig ist.

Abbildung 18 zeigt ein Beispiel für einen Android-Test, welcher die *get*-Methoden der User Klasse testet. Der für diesen Test benötigte Android Context ist durch die *AndroidTestCase*-Methode *getContext()* verfügbar.

```
public class UserTest extends AndroidTestCase {

    SharedPreferences sharedPreferences;
    long savedUserId;
    String savedUserName;
    String savedUserPassword;

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        sharedPreferences = PreferenceManager.getDefaultSharedPreferences(getContext());
        savedUserId = sharedPreferences.getLong(User.USER_ID, -1);
        savedUserName = sharedPreferences.getString(User.USER_NAME, "");
        savedUserPassword = sharedPreferences.getString(User.USER_PASSWORD, "");

        User.login(PseudoLogin.user, PseudoLogin.password, PseudoLogin.id);
    }

    public void testGetUserId() {
        assertEquals(PseudoLogin.id, User.getUserId());
    }

    public void testGetUserName() {
        assertEquals(PseudoLogin.user, User.getUserName());
    }

    public void testGetUserPassword() {
        assertEquals(PseudoLogin.password, User.getUserPasswaord());
    }

    @Override
    protected void tearDown() throws Exception {
        sharedPreferences.edit().putLong(User.USER_ID, savedUserId).commit();
        sharedPreferences.edit().putString(User.USER_NAME, savedUserName).commit();
        sharedPreferences.edit().putString(User.USER_PASSWORD, savedUserPassword).commit();
        super.tearDown();
    }
}
```

Abbildung 18. Beispiel Android-Test

3.5.4 Kriterien für gute Tests

Hier werden Kriterien für gute Tests genannt. Sie sind aus (Martin 2009, S. 161ff.) entnommen.

Tests sollten, wie normaler Source Code, **leicht verständlich** sein. Wenn sie nicht leicht verständlich sind, schleichen sich leicht Fehler ein. Das ist bei Tests besonders ärgerlich, da die Fehler im Test oder im Programm liegen können, was die Fehlersuche erschwert.

Außerdem erhöht es den Aufwand beim Beheben der Fehler, wenn ein Test fehlschlägt, aber nicht verständlich ist, was dieser Test überhaupt testet.

Aus den gleichen Gründen sollten Tests **einfach und primitiv** sein.

Des Weiteren ist **F.I.R.S.T** ein bekanntes Prinzip für Tests, das für **Fast, Independent, Repeatable, Self-Validating** und **Timely** steht.

Die Tests sollen schnell (**Fast**) sein, damit sie regelmäßig ausgeführt werden können und so unter anderem eine regelmäßige Wiederholbarkeit ermöglichen.

Wenn Tests voneinander abhängig sind, kann es sein, dass ein Test aufgrund eines anderen Tests scheitert, wodurch die Fehlersuche erschwert wird. Umgekehrt kann es auch sein, dass dadurch ein Fehler verborgen wird, was eine trügerische Sicherheit gibt. Deswegen sollen Tests unabhängig (**independent**) voneinander sein.

Bei der Wiederholbarkeit (**Repeatable**) ist es wichtig, dass die Tests jedes Mal das gleiche Ergebnis liefern. Tritt ein Fehler nur manchmal auf, ist er schwer zu lokalisieren. Die Wiederholbarkeit bezieht sich auch auf die Umgebung. Die Tests sollten in jeder Umgebung lauffähig sein und nicht nur am lokalen Computer eines Entwicklers, damit überall und jeder von ihnen profitieren kann.

Für die Automatisierung ist die Selbstvalidierung (**Self-Validating**) wichtig. Manuelle Validierung hat den Nachteil, dass sie zeitaufwendig ist und subjektiv interpretiert werden kann.

Die Tests sollten rechtzeitig (**Timely**) geschrieben werden. Werden Tests sogar vor dem Produktionscode geschrieben, kann kein nicht testbarer Source Code geschrieben werden und die Tests können sofort ausgeführt werden.

Wenn die Tests selber Source Code sind, wie bei JUnit, sind für Tests dazu die **gleichen Kriterien wie für guten Source Code** (s. Abschnitt 3.2.2) anwendbar.

Abschließend stellt sich noch die Frage, wie gut die Tests sind und wie sie getestet werden können. Wenn die Tests Fehlschlagen, obwohl alles in Ordnung ist, so erzeugen sie unnötige Arbeit. Schlagen sie nicht fehl, obwohl ein Fehler vorliegt, erfüllen sie nicht ihren Zweck. Dies wird auch als „*False Positive*“ und „*False Negative*“ bezeichnet. Auf jeden Fall geben schlechte Tests keine Sicherheit. Um die Tests zu testen, können mit Absicht Fehler in den Source Code eingebaut werden. Idealerweise finden die Tests alle eingebauten Fehler, ansonsten müssen die Tests entsprechend nachgebessert werden. Über ein Versionskontrollsystem können die Fehler anschließend ohne Aufwand wieder

entfernt werden. Durch beispielweise Jester[22] kann das Fehlereinbauen mittels Fehlerinjektion auch automatisiert werden. (Vigenschow 2010, S. 260f.)

3.5.5 Problematische Stellen

Genau wie beim Refaktorisieren gibt es auch beim Testen mit JUnit schwierige Stellen. Auf einige wird hier kurz eingegangen:

GUI Elemente bilden das graphische User Interface und beinhalten meist Fenster, Eingabefelder, Buttons, etc. Die Userinteraktion zu simulieren und evaluieren erfordert meist komplexes GUI Event Scripting. Es muss überprüft werden, ob die richtigen Fenster geöffnet und diese auch richtig angezeigt werden. Dazu können während dem Auswerten vom User zufallsmäßig Klicks eingestreut werden, die den Kontrollfluss variieren. Ein Ansatz, um diese Schwierigkeiten zu umgehen ist die *Humbling Dialog Box*. Sinngemäß ist das ein einfacher Dialog. Alle Daten die er anzeigen soll, werden durch eine *set*-Methode gesetzt, und alle eingegebenen Daten durch eine *get*-Methode gelesen. Sämtliche Logik wird, soweit es geht, separat implementiert. Die Idee ist, dass die Logik dann so wie jede andere Klasse getestet werden kann. Die GUI hingegen ist dadurch so einfach geworden und meist auf gut getestete Frameworks aufgebaut, so dass auf automatisierte Tests verzichtet werden kann. (Hamill 2005, S. 51ff.)

Eine weitere Schwierigkeit bilden **externe Schnittstellen**, wie **Webservices** oder **Datenbanken**. Sie sind oft in der Entwicklungsphase noch nicht verfügbar. Wenn sie verfügbar sind, kann das Aufrufen zeitaufwendig sein, was dazu führt, dass die Tests seltener durchgeführt werden. Oft ist auch zeitaufwendiges Einrichten von Testdaten nötig und es besteht die Gefahr, dass zum Beispiel bei einem Verbindungsfehler, der immer auftreten kann, Testdaten über den Test hinaus verbleiben. Als Lösung können externe Schnittstellen durch Mock-Objekte simuliert werden. Sie sind Objekte, deren einziger Zweck es ist, für die Tests die externen Schnittstellen mit ihren Funktionen, Eingabeparameter und zu erwarteten Output zu simulieren, so dass der eigene Source Code unproblematisch getestet werden kann. (Hamill 2005, S 41ff.)

Da JUnit auf die zu testende Objekte wie jede andere Klasse zugreift, kann sich auch ein **Sichtbarkeitsproblem** ergeben. Auf private Variablen und Methoden kann aus der Testklasse nicht zugegriffen werden. Zwei mögliche Lösungen wären die Kapselung aufzuweichen und die Sichtbarkeit der Variablen zu erhöhen, oder auf JUnit-Tests für solche zu verzichten. (Vigenschow 2010, S. 225f.) Als dritte Möglichkeit kann durch die Reflection API[23] von Java zur Laufzeit die Sichtbarkeiten von Methoden und Variablen

[22] („Jester“)
[23] („The Java™ Tutorials - The Reflection API“)

gesetzt werden. Dabei ist jedoch zu berücksichtigen, dass dies die einfache Verständlich- und Lesbarkeit der Tests vermindert.

Threads sind aufgrund ihrer Nebenläufigkeit oft komplex und fehleranfällig. Abbildung 19 zeigt einen sehr einfachen Source Code-Schnipsel. Wird in diesem *lastIdUsed* der Wert *0* zugeordnet und zwei verschiedene Threads rufen *getNextId()* auf, so kann anschließend das Ergebnis *2* sein, was zu erwarten ist, oder auch *1*, wenn die beiden Threads in Konflikt miteinander geraten. Das liegt daran, dass bei *++lastIdUsed* zuerst der Wert von *lastIdUsed* gelesen wird und in einem zweiten Schritt dieser um eins erhöht und zurückgeschrieben wird. Wird ein Thread T_A nach dem ersten Schritt unterbrochen und ein anderer Thread T_B verändert den Wert von *lastIdUsed*, so führt dies zum Konflikt. Da T_A bereits den Wert von *lastIdUsed* gelesen hatte, rechnet er mit dem alten Wert weiter und schreibt das Ergebnis zurück, so dass die Operation von T_B keinen Effekt hatte.

```
public class X {
  private int lastIdUsed;
  public int getNextId() {
    return ++lastIdUsed;
  }
}
```

Abbildung 19. Source Code für zwei Threads (Martin 2009, S. 220)

Der entsprechend generierte Byte-Code zu Abbildung 19 hat 12870 Ausführungspfade[24] für zwei Threads innerhalb ihrer *getNextId()*-Methode, von denen nur ganz wenige das überraschende Ergebnis *1* liefern würden. Dementsprechend würden JUnit-Tests diesen möglichen Fehler nur sehr selten finden und nicht sicher reproduzieren können, was die Fehlersuche erschwert.

Eine weitere Schwierigkeit Threads mit JUnit zu testen, ist die Tatsache, dass die JUnit-Tests nicht auf die gestarteten Threads warten. Um ein Ergebnis eines Threads zu testen, muss der Test manuell durch *wait*- oder *sleep*-Befehle auf diesen warten. Dies ist aufwändig und ebenfalls fehleranfällig.

```
AtomicBoolean atomic = new AtomicBoolean(false);
// Do some async stuff that eventually updates the atomic boolean
await().untilTrue(atomic);
```

Abbildung 20. Awaitility Beispiel („Awaitility“)

Awaitility[25] ist eine Bibliothek, die diese Arbeit vereinfacht. Der Source Code in Abbildung 20 hält zum Beispiel den aufrufenden Thread so lange wach, bis die Variable *atomic* auf *true* gesetzt wurde, oder wirft nach zehn Sekunden eine *ConditionTimeoutException*. Die Zeit, bis diese Exception geworfen wird, kann ggf. beliebig eingestellt werden.

[24] (Martin 2009, S. 220)
[25] („Awaitility“)

Abschließend sei darauf hingewiesen, dass JUnit-Tests nicht alles abdecken. So haben zum Beispiel Usability oder Performance Tests andere Belange, die nicht von den JUnit-Tests abgedeckt werden. (Hamill 2005, S. 4f.)

3.5.6 Testen und Refaktorisieren

Wie bereits erwähnt, verändert das Refaktorisieren den Source Code, ohne dabei sein zu beobachtendes Verhalten zu verändern. Genau darin liegt ein Problem. Denn wie kann sichergestellt werden, dass sich das gesamte Programm nachher genauso verhält wie vorher? Es können sich bei jeder Änderung simple Fehler einschleichen, oder unerwartete Seiteneffekte zur Folge haben.

Automatische Tests, die vor und nach jeder Refaktorisierung das gleiche Ergebnis liefern, bieten eine Sicherheit, dass genau das nicht passiert. Sie werden deswegen als Vorbedingung für das Refaktorisieren benötigt.

Bevor eine Stelle im Source Code refaktorisiert wird, wird ein Test für dessen Verhalten geschrieben. Wenn nach der Refaktorisierung dieser Test bestanden wird, hat sich wahrscheinlich kein Fehler dabei eingeschlichen. Wenn nur durch Tests abgedeckter Source Code verändert wird, werden also Fehler wahrscheinlich sofort entdeckt. Dabei ist wahrscheinlich, dass die Fehler in dem gerade geänderten Abschnitt liegen, da die Tests vorher noch bestanden wurden. Somit wird die Fehlersuche auf ein Minimum reduziert. Dies verdeutlicht nebenbei, dass die automatisierten Tests während einem Projektverlauf immer mehr an Wert gewinnen, da in der Regel später umfangreicher refaktorisiert wird. (Vigenschow 2010, S. 195ff.)

3.5.7 Tests Refaktorisieren

Wenn der Produktionscode weiterentwickelt wird und sich ändert, müssen auch die Tests angepasst werden. Die JUnit-Tests sind dabei, genauso wie der Produktionscode, Source Code. Schlecht geschriebene Tests zu verändern, ist somit genauso zeitaufwändig und teuer, wie normalen Source Code zu ändern. So können auch schlecht geschriebene Tests zum Scheitern eines Projektes führen. (Martin 2009, S. 161f.)

Für Tests gelten die gleichen Kriterien wie für guten Source Code (s. Abschnitt 3.2). Die Smells aus Abschnitt 3.2.4 gelten im Allgemeinen auch für Tests. In (Kerievsky 2004) wird dazu konkret auf Test-spezifische Smells eingegangen. Es werden auch Test Patterns, ähnlich wie Design Patterns (s. Abschnitt 3.2.5) vorgestellt.

Da JUnit-Test-Code normaler Source Code ist, muss er dementsprechend genauso gewartet und regelmäßig refaktorisiert werden. Beim Refaktorisieren der Tests, dient dann der Produktionscode als Test. Vor dem Refaktorisieren wurde er als fehlerfrei getestet.

Wird er es nachher nicht mehr, so wurde wahrscheinlich beim Refaktorisieren der Tests Fehler eingebaut. (Lippert und Roock 2004, S. 25)

Für genauere Betrachtungen sei auf (Meszaros 2007) verwiesen.

4 Das Verbessern des Source Codes vom TOTEM.Scout

Im Abschnitt 2.5 wurde motiviert, dass der Source Code des Scouts verbessert werden soll. Im 3. Kapitel wurde dazu definiert, was guter Source Code ist, und das Refaktorisieren vorgestellt. Dieses Kapitel beschäftigt sich nun mit der Umsetzung der Verbesserung des Source Codes.

4.1 Analyse des Ist-Zustands

Um die Überarbeitung vom Scout vorzubereiten, wird hier zunächst der Ist-Zustand vom Source Code des Scouts analysiert.

4.1.1 Die Package-Struktur

Die Package-Struktur besteht aus den vier Packages *de.fraunhofer.fit.cvae.totem .locationModel*, *de.fraunhofer.fit.cvae.totem.totemscout* *de.fraunhofer.fit.cvae.totem .totemscoutmarbles* und *net.technologichron.android.control*, die in Abbildung 21 zu sehen sind. Im Folgenden werden sie mit *.locationModel*, .totem*scout*, *.totemscout marbles* und *.control* abgekürzt.

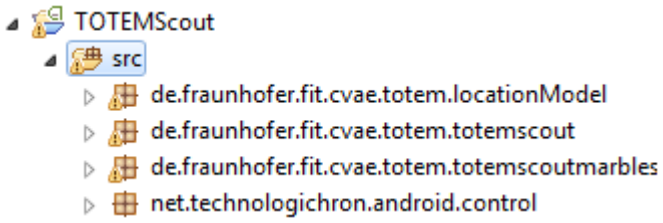

Abbildung 21. TOTEM.Scout Packages

Abbildung 22 zeigt die Packages mit den ihnen zugehörigen Klassen.

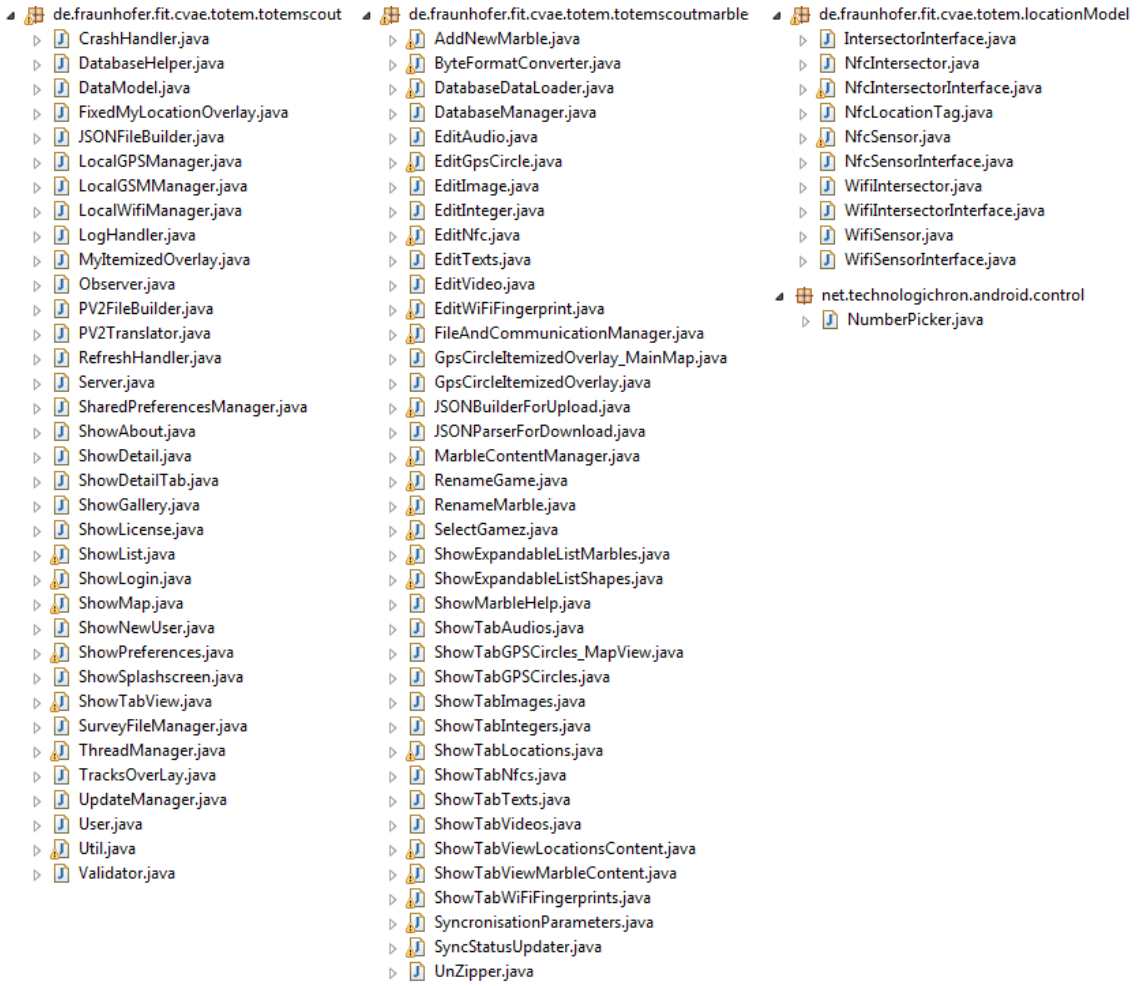

Abbildung 22. TOTEM.Scout Packages mit Klassen

Es fällt sofort auf, dass sich im Package *.control* nur die Klasse *NumberPicker.java* befindet, weswegen dieses Package kritisch hinterfragt werden sollte. Seit dem Android API Level 11[26] gibt es einen vorgefertigten NumberPicker[27] von Android, womit diese Klasse mittlerweile einfach überflüssig ist.

Im Package *.totemscoutmarbles* wird alles erwartet, was Marbles behandelt. Folgende Klassen widersprechen dem jedoch:

- *UnZipper.java*, die eine gezippte Datei entzipped.
- *ByteFormatConverter.java*, die eine Byte Zahl in einen String mit der Maßeinheit MB oder KB formatiert.
- *FileAndCommunicationManager.java*, die später noch genauer erörtert wird.
- *ShowExpandableListShapes.java*, die eine Liste von allen verfügbaren Formen zeigt.

[26] Android API Level 11 entspricht der Android Version 3.0 „Honeycomb“
[27] („Android Developers - NumberPicker“)

Des Weiteren würden die Klassen *LocalGPSManager.java*, *LocalGSMManager.java* und *LocalWifiManager.java* eher im Package *.locationModell* als im Package *.totemscout* vermutet werden.

Die vier Klassen *DatabaseHelper.java*, *DataModel.java*, *DatabaseDataLoader.java* und *DatabaseManager.java* verteilen sich auf die Packages *.totemscout* und *.totemscoutmarbles* und lassen sich vom Namen her nur schwer in ihrem Sinn und Zweck differenzieren. Es überrascht jedoch auf jeden Fall, sie nicht im gleichen Package zu finden.

4.1.2 Vorhandene Smells

Zu jedem in Abschnitt 3.2.4 erläuterten Smell wird hier beispielhaft ein Vorkommen im Scout angeführt.

Duplizierter Code: Der Code in Abbildung 23 kommt acht Mal in der Klasse *SurveyFileManager.java* im Package *.totemscout* vor und könnte problemlos in eine Methode verpackt werden.

```
if (!initialised) {
    Util.logErrorMessage("SurveyFileManager not initialised",
            this.getClass());
    return null;
}
```

Abbildung 23. Code-Duplizierung

Faule Klasse: Die Klasse *RenameMarble.java* im Package *.totemscoutmarbles* erledigt nichts anderes, als den Namen einer Marble zu ändern. Diese von Activity abgeleitete Klasse hat einen großen Overhead. Stattdessen könnte dies durch einen schlankeren Android AlertDialog erledigt werden.

Große Klasse: Mit über 1800 Zeilen kann die Klasse *FileAndCommunication Manager.java* im package *.totemscoutmarbles* zweifelsfrei dem Smell „Große Klasse“ zugeordnet werden.

Lange Methode: Die Methode *parseGamesJSONFileData(String path, String GameJsonFileName)* aus der Klasse *JSONParserForDownload.java* im Package *.totemscoutmarbles* lässt sich mit ihren 168 Zeilen dem Smell „Lange Methode“ zuordnen.

Divergierende Änderungen: Da dieser Smell oft in der Kombination mit dem Smell Große Klasse vorkommt, lässt er sich auch in der Klasse *FileAnd CommunicationManager.java* im Package *.totemscoutmarbles* finden. So behandelt die Klasse unter anderem:

- Das Erstellen und Löschen von Dateien

- Das Überprüfen ob Internet vorhanden ist
- Verschiedene User Dialoge
- Verschiedene asynchrone Threads zum Hoch- und Herunterladen von Daten mit dem Designer.

Dass die Klasse von *Activity.java* erbt, obwohl sie keine visuellen Elemente hat, ist des Weiteren auch kritisch zu hinterfragen.

Schrotkugel herausoperieren: Im Source Code des Scouts kommt der String „*integers*“ 24 Mal in zehn verschiedenen Klassen vor. Wird dieser String verändert, muss er an all diesen Stellen mit geändert werden. Eine Konstante *INTEGERS* mit dem Wert „*integers*“ wäre eine erste offensichtliche Verbesserung. Soll aber zum Beispiel dieser String entfernt werden oder ein neuer hinzugefügt werden, müsste dies immer noch an allen 24 Stellen geschehen.

***Switch*-Befehl**: Das Beispiel von dem vorherigen Smell Schrotkugel herausoperieren, ist das Resultat davon, dass der Datentyp einer Property durch eine *Switch*-Abfrage festgestellt wird und kann somit auch als Beispiel für diesen Smell herangezogen werden.

```
switch (marbleProperties) {

    case AUDIOS:
        result = (List)gameMarblePropertiesList.get("audios");
        break;

    case IMAGES:
        result = (List)gameMarblePropertiesList.get("images");
        break;

    case INTEGERS:
        result = (List)gameMarblePropertiesList.get("integers");
        break;

    case LOCATIONS:
        result = (List)gameMarblePropertiesList.get("locations");
        break;

    case TEXTS:
        result = (List)gameMarblePropertiesList.get("texts");
        break;

    case VIDEOS:
        result = (List)gameMarblePropertiesList.get("videos");
        break;

}
```

Abbildung 24. Smell *Switch*-Befehl

Abbildung 24 zeigt so einen *Switch*-Befehl, welcher den Datentyp einer Property feststellt. In der Klasse kommen 8 ähnliche Switch-Befehle vor, so dass diese auch ein Beispiel für den Smell „Duplizierter Code“ sind. Dieser *Switch*-Befehl

könnte nebenbei offensichtlich durch eine Methode *getList(String type)* ersetzt werden.

Kommentare: Als Beispiel für diesen Smell dient auch hier die Klasse *FileAndCommunicationManager.java*. Sie beinhaltet 22 Zeilen auskommentierten Source Code.

Falsche Zuständigkeit: Für diesen Smell kann der gesamte Abschnitt 4.1.1 herangezogen werden. Wird eine Klasse in einem anderen Package vermutet, so ist das gegenwärtige Package nicht für diese Klasse zuständig.

Mentales Mapping (Nach Martin 2009, S. 54): Dieser Smell wurde zwar nicht im Abschnitt 3.2.4 eingeführt und wird im Kontext von guter Namenswahl verwendet, er ist jedoch ein treffender Name für den im Source Code aus Abbildung 25 enthaltenen Smell.

```
    //select a particular game by number in the List
Map game = SyncronisationParameters.getParsedGameList().get(groupPosition);

    //retrieve a Marble List for the selected game
List gameMarblesList = (List) game.get("marbles");

    //select a particular Marble in the Marble List
Map gameMarble = (HashMap) gameMarblesList.get(childPosition);

    // set the selection for the onResume focus in the ShowExpandableListMarbles
ShowExpandableListMarbles.selectedGame = groupPosition;
ShowExpandableListMarbles.selectedMarble = childPosition;

    //retrieve the Marble properties
Map gameMarblePropertiesList = (Map) gameMarble.get("properties");

    //retrieve the Marble audios, videos, etc.. list
List audiosList = (List)gameMarblePropertiesList.get("audios");
List imagesList = (List)gameMarblePropertiesList.get("images");
List integersList = (List)gameMarblePropertiesList.get("integers");
List locationsList = (List)gameMarblePropertiesList.get("locations");
List textsList = (List)gameMarblePropertiesList.get("texts");
List videosList = (List)gameMarblePropertiesList.get("videos");
```

Abbildung 25. Smell „Mentales Mapping“

Dieser Source Code liest die Properties einer Marble. Dazu muss wie folgt vorgegangen werden: Eine Liste enthält alle Spiele. Ein Spiel ist ein Map-Objekt. Jedes dieser Spiele hat einen Eintrag, welcher ein Map-Objekt enthält, das seinerseits alle Marbles dieses Spiels beinhaltet. Die gesuchte Marble hat dann einen Eintrag für ihre Properties, welche wiederum ein Map-Objekt ist. Aus diesem Properties-Map-Objekt kann nun die Listen aller Properties gelesen werden.

Die Notwendigkeit, dass vor jedem Schritt ein Kommentar steht, der die aktuelle Abstraktionsebene beschreibt, zeigt, dass der Leser hier ein mentales Mapping

vollführen muss. Die Kommentare sind nebenbei ein Beispiel für den Smell „Kommentare“, wenn dieser versucht, schlechten Source Code zu kompensieren. Abgesehen von der schweren Lesbarkeit und der nicht intuitiven Zugriffsart, die wahrscheinlich jedes Mal nachgeschaut werden muss, ist diese Lösung auch sicherlich nicht die effizienteste.

4.1.3 Entdeckte Bugs

Zusätzlich wurden im Scout folgende Bugs gefunden:

Fehlende Datentypen: Der Scout ignoriert alle Daten beim Herunterladen vom Designer des Typs Float und Boolean. Werden dann nach Shapes, die Float und Boolean-Werte beinhalten, neue Marbles auf dem Scout erstellt und anschließend hochgeladen, überprüft der Designer die Daten nicht. So werden diese Marbles nicht nur im Scout unvollständig erzeugt, sondern erhalten auch, inkonsistent zu ihrer Shape, Einzug ins System.

Default Marbles: Im Scout werden die Default Marbles mit in der Liste der normalen Marbles angezeigt. Im Designer sind diese jedoch nur über die Shape zu setzen und nicht in der Liste der Marbles sichtbar. Dass sie im Scout angezeigt werden und den Namen „DefaultMarble“ tragen, ist verwirrend. Dazu verletzt es das Konzept, dass auf dem Scout keine Shapes editiert werden können.

Ewige temporäre ID: Im Scout erzeugte Marbles erhalten eine temporäre ID. Werden diese hochgeladen, gibt der Designer ihnen eine echte ID. Auf dem Scout behalten sie jedoch ihre temporäre ID. Werden sie dann auf dem Scout editiert und noch einmal hochgeladen, werden somit nicht die entsprechenden zuvor hochgeladenen Marbles auf dem Designer geändert, sondern stattdessen Neue angelegt.

Fehlende Fehlerausgabe beim Upload: Beim Hochladen der Marbles wird nicht über eventuelle Fehler, wie beispielsweise ein Connection Time out, informiert. Stattdessen wird immer der Toast „upload successful“ ausgegeben.

Leere Namen: Im Gegensatz zum Designer erlaubt es der Scout, Marbles einen leeren Namen zu geben und hochzuladen. Marbles ohne Namen können auf dem Designer jedoch nicht editiert werden, da der Name ein Link ist, der dann nicht angezeigt wird.

Google Maps: Vermutlich auf Grund einer API-Änderung seitens Google wird die integrierte Google Maps nicht mehr gerendert, wie in Abbildung 26 zu sehen ist. Überlappen sich dazu mehrere Marker, ist nur der oberste selektierbar. Zurzeit

wird die alte Google Maps Android API v1 verwendet wird, so dass beim Beheben gleich auf die aktuelle Version[28] umgestiegen werden kann.

Abbildung 26. Links wie die Google Maps einmal aussah (Jurgelionis u. a. 2012, S. 7), rechts zum Zeitpunkt der Ist-Zustands-Analyse

4.1.4 Zwei Hauptschwachstellen

Bei genaueren Betrachtungen sind zwei grundlegende Probleme aufgefallen, die das Warten und Erweitern des Source Codes erschweren. Sie werden folgend als zwei Hauptschwachstellen vorgestellt.

Das Datenmodell ist doppelt implementiert: Für die persistente Speicherung wird eine SQLite Datenbank verwendet. Wird zur Laufzeit auf die Datenbank zugegriffen, sollte dies stets in einem eigenen Thread geschehen, da eine Anfrage an eine Datenbank, unter anderem aufgrund von Hintergrundprozessen, länger dauern kann und es so zu einem *Application Not Responding*-Error[29] kommen kann. Um dieses Threading zu umgehen, werden beim Programmstart sämtliche Daten in ein *HashMap*-Objekt geladen, auf das ohne Threading direkt zugegriffen werden kann. Dies hat jedoch den Nachteil, dass das Datenmodell zum einen in der Datenbank und zum anderen in dem *HashMap*-Objekt implementiert ist. Dies hat erhebliche Code-Duplizierung zur Folge und widerspricht dem DRY-Prinzip. Das *HashMap*-Objekt führt außerdem zu dem Smell „Mentales Mapping“, welcher in Abbildung 25 dargestellt war.

[28] („Google Maps Android API v2“)
[29] („Android Developers - Keeping Your App Responsive“)

Schwere Erweiterbarkeit: Wie in Abschnitt 2.4 erwähnt, werden Marbles nach einer vordefinierten Shape geformt. Die Shape gibt vor, welche Datensätze in der Marble enthalten sein müssen. Diese Datensätze bestehen aus verschiedenen Datentypen wie Integers oder Booleans. Die unterschiedlichen Datentypen werden im Source Code über *Switch*-Befehle und Strings verarbeitet, wodurch die Smells Schrotkugel und *Switch*-Befehle entstanden sind. Das erschwert es, neue Datentypen hinzuzufügen, da in jedem *Switch*-Befehl ein neues *case*-Statement hinzugefügt werden muss. Die in Abbildung 25 zu sehende direkte String-Verwendung ist außerdem fehleranfällig, da der Entwickler sich leicht vertippen kann. Darüber hinaus ist es mühselig einen String zu ändern, da er an jeder Stelle geändert werden müsste.

4.2 Refaktorisieren oder Neuschreiben

Nachdem zuvor viele Stellen gefunden wurden, die refaktorisiert werden sollten, stellt sich die Frage, ob ein Neuschreiben nicht eine einfachere Alternative zum Refaktorisieren ist (s. auch Abschnitt 3.4.3). Dass Bugs entdeckt wurden, verstärkt diese Frage. Deswegen werden hier weitere Überlegungen dazu angestellt.

In Abschnitt 4.1.4 wurden zwei schwerwiegende Schwachstellen gefunden. Erstens wurde das Datenmodell doppelt implementiert, nämlich zum einen als Datenbankmodell und zum anderen in einem einzigen HashMap-Objekt. Als Datensammler für den Designer ist der Scout eine rein datengetriebene App. Die Implementierung des Datenmodells zu ändern, zieht sich somit durch die gesamte Anwendung durch und ist eine sehr umfangreiche Refaktorisierung. Große Refaktorisierungen sind jedoch riskant. Die Stellen können zwar nacheinander einzeln refaktorisiert werden, dennoch ist die gesamte Refaktorisierung sehr umfangreich und kommt einem Redesign nahe.

Die zweite schwerwiegende Schwachstelle der schweren Erweiterbarkeit beruht darauf, dass beim schnellen iterativen Rapid-Prototyping-Erstellungsprozesses Smells wie *Switch*-Befehle und direkte String-Vergleiche akzeptiert wurden, um schnell einsatzfähige Prototypen zur Verfügung zu haben. (Buxton und Sniderman 1980) beschreiben das Akzeptieren von Smells für die schnelle Einsatzfähigkeit als „*The trap of the "path of least resistance"* ", den es zu vermeiden gilt. Wenn die Wartbarkeit und Erweiterbarkeit durch den "*path of least resistance*" gelitten hat, nehmen sie dies als Indiz dafür, dass ein Redesign notwendig ist.

Folgend werden noch einmal alle erörterten Punkte zusammengefasst:

- Die Package-Struktur ist nicht passend (s. Abschnitt 4.1.1).
- Es wurden viele Smells gefunden (s. Abschnitt 4.1.2).

- Es wurden Bugs entdeckt (s. Abschnitt 4.1.3). Nach einem reinen Refaktorisieren, welches das von außen beobachtbare Verhalten nicht verändert, wären diese Bugs immer noch vorhanden.
- Eine Änderung der Implementierung des Datenmodells zieht sich durch die gesamte Anwendung und kommt somit einem Redesign bereits nahe (s. Abschnitt 4.1.4).
- Dass die Wartbarkeit und Erweiterbarkeit gelitten haben, ist ein Indiz dafür, dass ein Redesign notwendig ist (s. Abschnitt 4.1.4).
- All diese Punkte bekräftigen meine Intuition, welche für das Refaktorisieren sehr wichtig ist, dass ein Neuschreiben beim Scout die bessere Alternative ist.

Somit wurde zu dem Schluss gekommen, dass nicht wie ursprünglich geplant eine Refaktorisierung durchzuführen ist, sondern stattdessen ein Neuschreiben.

4.3 Verbesserungen der neuen Version

Bei der Neuimplementierung wurde wie folgt vorgegangen:

1. Es wurde überlegt, wie das Datenmodell umgesetzt wird, auf welches der Scout aufbaut. Diese Überlegungen wurden umgesetzt und bei neuen Erkenntnissen ggf. angepasst und refaktorisiert.
2. Es wurde versucht, den Scout in funktionierende Komponenten aufzuteilen, die übernommen werden können und nicht neu geschrieben werden müssen. Die XML-Layout-Dateien, die das graphische Userinterface erzeugen, konnten dabei größtenteils übernommen werden. Der Java Source Code wurde hingegen größtenteils neu geschrieben.
3. Beim Neuschreiben lag ein Hauptaugenmerk darauf, die in der Ist-Zustands-Analyse gefundenen Schwachpunkte zu verbessern.

Folgend wird auf die Beseitigung der zwei gefundenen Hauptschwachstellen und auf die neue Package-Struktur eingegangen. Des Weiteren werden die Smells thematisiert. Da nicht refaktorisiert wurde, konnten auch gleich kleinere Änderungen im nach außen sichtbaren Verhalten vorgenommen werden, welche zum Schluss vorgestellt werden.

4.3.1 Datenmodell an einem Ort

Der Smell „Code Duplizierung“ ist der schlimmste aller Smells. Somit sollte das Datenmodell auf jeden Fall nur an einer Stelle implementiert sein. Da der Scout eine persistente Speicherung seiner Daten benötigt, wurde die relationale Datenbank beibehalten und das *HashMap*-Objekt entfernt.

Dadurch galt es, die Komplexität durch das benötigte Threading, um nicht im UI-Thread auf die Datenbank zuzugreifen, zu minimieren. Zu diesem Zweck wird in zwei Arten des Datenbankzugriffs unterschieden. Zum einen ein ausschließlich **schreibender Datenzugriff**, der komplett im Hintergrund ausgeführt werden kann. Zum anderen ein **lesender Datenbankzugriff**, der für den Anwender Daten liest und somit der UI-Thread auf das Ergebnis des Datenbankzugriffs warten sollte.

Beim **schreibenden Datenbankzugriff** sind stets folgende Schritte nötig:

1. Einen neuen Thread erzeugen und die Schritte zwei bis vier in diesem ausführen.
2. Eine Datenbankverbindung öffnen.
3. Die entsprechenden Daten in die Datenbank schreiben.
4. Die Datenbankverbindung wieder schließen.

Bei diesen vier Schritten, die auch als Algorithmus bezeichnet werden können, muss dabei nur der dritte Schritt individualisiert werden. Dementsprechend sollten sich die anderen drei Punkte verallgemeinert implementieren lassen, so dass stets nur der dritte Punkt implementiert werden muss.

Das Template Method Pattern[30] erfüllt genau diese Anforderung. In diesem gibt eine abstrakte Klasse eine Template-Methode an. Diese definiert den Algorithmus. Alle gemeinsamen Aspekte des Algorithmus werden dann in der Template-Methode implementiert und die individuellen Aspekte werden durch abstrakte Methoden, die auch *primitive operations* genannt werden, an die richtige Stelle eingeführt. Eine konkrete Klasse erbt von der abstrakten Klasse und muss nun nur noch seine individuellen Aspekte implementieren, in dem sie die *primitive operations* überschreibt. So ist der konkrete Algorithmus von der Komplexität des Algorithmus ein Stück weit befreit, da er nur seine individuellen Aspekte implementieren muss. Darüber hinaus wird so Code-Duplizierung vermieden.

[30] (Gamma u. a. 2007, S. 325ff.)

Abbildung 27 zeigt diese Template-Methode und Abbildung 28 eine beispielhafte konkrete Implementierung, um im Hintergrund die Werte eines GPS Circles zu aktualisieren.

```
public abstract class DatabaseThread extends Thread {

    protected abstract void run(SQLiteDatabase db);

    @Override
    public void run() {
        SQLiteDatabase db = TOTEMDatabase.getInstance().getWritableDatabase();
        run(db);
        db.close();
    }
}
```

Abbildung 27. Database Thread Template-Methode

```
public static void saveGPSCircle(final GPSCircleHelper gpsCircle) {
    new DatabaseThread() {
        @Override
        protected void run(SQLiteDatabase db) {
            ContentValues cv = new ContentValues();
            cv.put(COLUMN.LATITUDE, gpsCircle.latitude);
            cv.put(COLUMN.LONGTITUDE, gpsCircle.longitude);
            cv.put(COLUMN.RADIUS, gpsCircle.radius);
            SQLCommands.update(db, TABLE.GPS_CIRCLE_PROPERTY, gpsCircle.id, cv);
        }
    }.start();
}
```

Abbildung 28. Konkreter Database Thread

Beim **lesenden Datenbankzugriff** sind stets folgende Schritte nötig:

1. Einen neuen Thread erzeugen.
2. Dem User eine Progress Circle anzuzeigen, damit er weiß, dass er auf ein Ergebnis warten soll und die App nicht abgestürzt ist[31].
3. Eine Datenbankverbindung öffnen.
4. Die gewünschten Daten aus dieser abfragen.
5. Die Datenbankverbindung wieder schließen.
6. Mit dem Ergebnis der Abfrage im UI-Thread weiterarbeiten.
7. Den Progress Circle wieder entfernen.

[31] („Android Developers - Progress & Activity“)

Dabei sind nur der vierte und der sechste Schritt individuell. Somit kann dies durch das Template Methode Pattern wieder unterstützt werden. Eine beispielhafte Umsetzung ist in Abbildung 29 zu sehen.

```
public abstract class DatabaseBackgroundOperation<T> extends AsyncTask<Void, Void, T> {
    protected Activity activity;
    private CircleProgressBar cp;

    protected abstract T executeInBackground(SQLiteDatabase db);
    protected abstract void afterExecuteOnUIThread(T result);

    public DatabaseBackgroundOperation(Activity activity) {
        this.activity = activity;
    }

    protected void onPreExecute() {
        super.onPreExecute();
        cp = new CircleProgressBar(activity);
    }

    protected final T doInBackground(Void... voids) {
        SQLiteDatabase db = TOTEMDatabase.getInstance().getWritableDatabase();

        T result = executeInBackground(db);

        db.close();

        return result;
    }

    protected void onPostExecute(T result) {
        super.onPostExecute(result);

        afterExecuteOnUIThread(result);
        cp.removeProgressCircle();
    }
}
```

Abbildung 29. Database Background Operation

Die Klasse *AsyncTask.java*[32] ist dabei eine Android-spezifische Klasse, die es ermöglicht Hintergrundoperationen einfach auslagern zu können und mit dem Ergebnis im Main-Thread weiterarbeiten zu können. Ihre generische Methode *doInBackground(Typ… params)*, wird automatisch in einem neuem Thread ausgeführt. Der Methode *onPostExecute(Typ result)*, welche wieder im Main-Thread ausgeführt wird, wird der Rückgabewert von *doInBackground(Typ… params)* übergeben, mit dem dann im Main-Thread weitergearbeitet werden kann. Dementsprechend ist die Klasse *AsyncTask.java* selber eine abstrakte Template-Methode.

Für einen Datenbankzugriff muss nun nur noch von der Klasse *DatabaseBackgroundOperation.java* geerbt und ihre *primitive operations* *executeInBackground* und *afterExecuteOnUIThread* überschrieben werden, um die individuellen Punkte vier und sechs zu implementieren.

[32] („Android Developers - AsyncTask“)

Durch die zwei Template-Methoden aus Abbildung 27 und Abbildung 29 wurde somit die Komplexität des Threadings in den Template-Methoden versteckt und gleichzeitig Code-Duplizierung vermieden. Daher kann nun auf das *HashMap*-Objekt verzichtet werden.

4.3.2 Erweiterbarkeit

Um die Erweiterbarkeit des Scouts zu verbessern, wurde auf zwei Prinzipien geachtet. Zu einem auf das **Open-Closed-Prinzip**[33] (OCP) und zum anderen auf **„Program to an interface, not an implementation“**[34].

Das **Open-Closed-Prinzip** besagt, dass der Source Code offen für Erweiterungen und geschlossen für Veränderungen sein soll. Das heißt, dass das Erweitern durch Hinzufügen von neuem Source Code geschieht und dabei kein bereits vorhandener Source Code geändert werden muss. Muss bereits vorhandender Source Code geändert werden, besteht die Gefahr, dass in funktionierenden Source Code Bugs eingebaut werden. Eventuell müssen sogar mehrere Stellen geändert werden, was aufwendig sein kann, und es besteht dann Gefahr, dass eine Stelle dabei übersehen wird.

Dadurch, dass der Datentyp im Scout bisher über *Switch*-Befehle und Strings bestimmt wurde, verletzte er das OCP. Sollte ein neuer Datentyp hinzugefügt werden, müsste nämlich bei jedem *Switch*-Befehl ein neuer Fall hinzugefügt werden.

„Program to an interface, not an implementation“ besagt, dass ein Client nur das benötigte Interface kennen sollte und nicht die konkreten Klassen. Dadurch können die konkreten Objekte ausgetauscht und erweitert werden, ohne dass sich der Client daran anpassen muss. „Program to an interface, not an implementation“ unterstützt somit das OCP.

In der neuen Version des Scouts wurde deswegen ein Interface Property angelegt, welche alle Methoden definiert, die eine Property erfüllen muss, wie beispielsweise *edit(Activity activity, int id)*. Jede Property muss dieses Interface implementieren. Wird eine Property nun bearbeitet, muss nicht über einen *Switch*-Befehle sein Typ festgestellt werden, sondern es kann stattdessen auf dem Property-Objekt die gewünschte Methode aufgerufen werden.

[33] (Martin 1996)
[34] (Gamma u. a. 2007, S. 17f.)

Abbildung 30 zeigt dies mit einem Ausschnitt aus der Klasse *PropertyList.java*. Sie enthält ein Property-Objekt und eine Liste von IDs der anzuzeigenden Properties. Wenn ein Eintrag angeklickt wird, wird die *edit*-Methode mit der entsprechenden ID aufgerufen. Wird nun eine neue Property hinzugefügt, durchs hinzufügen einer neuen Klasse, die das Property Interface implementiert, muss weder die Klasse *PropertyList.java* noch irgendeine andere bereits bestehende Property geändert werden. Somit können nun einfach neue Properties hinzugefügt werden, was dem OCP entspricht.

```
public void onListItemClick(ListView l, View v, int position, long id) {
    property.edit(getActivity(), ids[position]);
}
```

Abbildung 30. Polymorpher Methodenaufruf

4.3.3 Die neue Package-Struktur

Die neue Package-Struktur wurde an das Model/View/Controller Pattern[35] angelehnt. Es wurde 1979 zuerst in Smalltalk umgesetzt und ist mittlerweile sehr bekannt. Das Model enthält die Daten einer Anwendung. Die View zeigt diese Daten an und der Controller verbindet das Model und die View. Diese logische Aufteilung fördert ein flexibles Design, in dem idealerweise das Model ausgetauscht werden kann, ohne dass sich etwas in der View ändern muss und umgekehrt genauso.

Der Hauptgrund, wieso sich an diese Struktur angelehnt wurde, ist, dass es sehr bekannt ist und es kaum Unklarheiten gibt, in welches Package eine Klasse gehört. So erleichtert es den Einstieg für neue Programmierer und jede Klasse lässt sich intuitiv anhand der Package-Struktur finden.

[35] (Gamma u. a. 2007, S. 4ff.)

Abbildung 31 zeigt die umgesetzte Package-Struktur mit ihren jeweiligen Klassen.

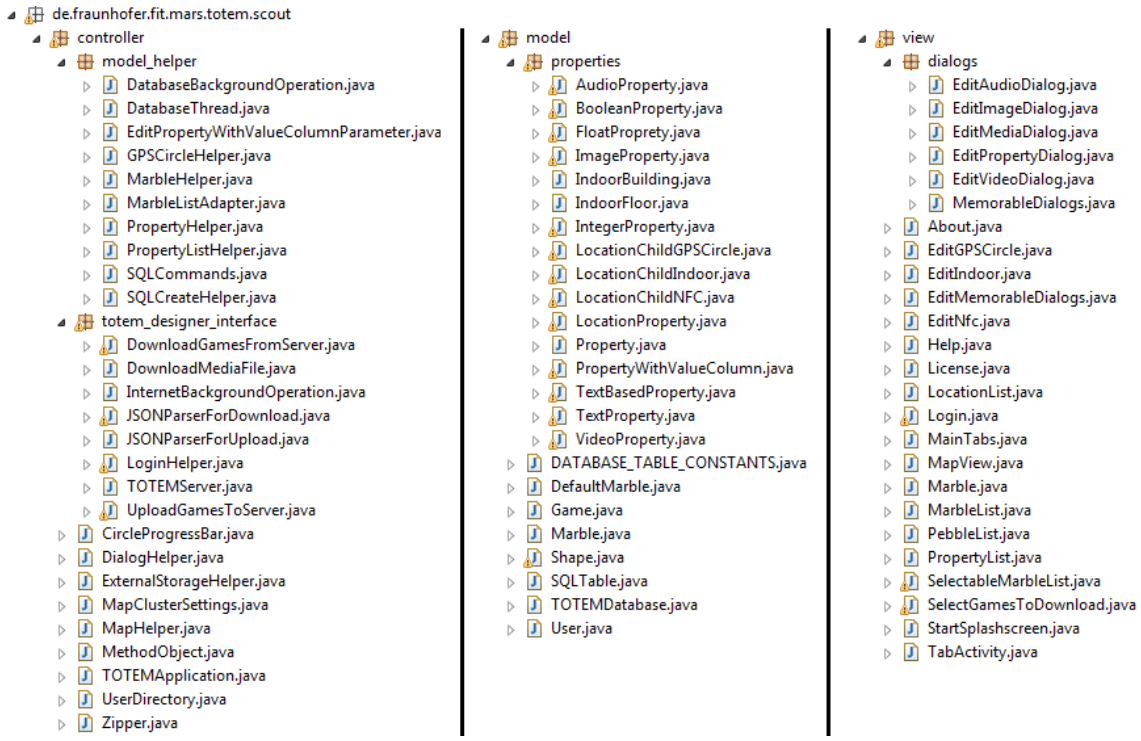

Abbildung 31. Die neue Package-Struktur

4.3.4 Behobene Smells

Hier wird erörtert, dass die in Abschnitt 4.1.2 gefundenen Smells nicht wiederholt wurden. Angaben über Zeilen des Source Codes, wie zum Beispiel die längste Methode, wurden durch das Eclipse Plugin Metrics[36] gemessen. Es sei erwähnt, dass die Smells nicht von Anfang an vermieden wurden, sondern dass, wenn ein Smell entdeckt wurde, dieser Refaktorisiert wurde.

Duplizierter Code: Um diesen Smell nicht zu wiederholen, wurde die Dreierregel aus Abschnitt 3.4.2 berücksichtig. Das heißt, dass wenn etwas Ähnliches zum dritten Mal implementiert wurde, refaktorisiert wurde, so dass die Gemeinsamkeiten an einem Ort gebündelt sind. Beispiele dafür sind die Template-Methoden um den Datenbankzugriff zu vereinheitlichen aus Abbildung 27 und Abbildung 29. Nachdem zum dritten Mal die gleichen Schritte nötig waren, unter anderem das Öffnen und Schließen der Datenbank, wurden diese in den Template-Methoden verallgemeinert.

Faule Klasse: Dieser Smell bezog sich auf die Dialoge, die im alten Scout eigene Activities waren. Stattdessen wurden im neuen Scout Dialoge verwendet. So wurde ein Package *view.dialogs* angelegt und im *controller*-Package eine Klasse *DialogHelper.java*. Ein weiterer Vorteil davon ist, dass die Dialoge nun an einem

[36] („Metrics 1.3.6“)

Ort zu finden sind und damit leichter wiederverwendet und auch im Aussehen vereinheitlicht werden können.

Große Klasse: Mit 352 Zeilen ist die Klasse *SelectGamesToDownlaod.java* nun die größte Klasse im neuem Scout. Sie ist die einzige Klasse, die größtenteils aus der alten Version übernommen wurde. In weiteren Refaktorisierungen könnten zum Beispiel die enthaltenen inneren Klassen ausgelagert werden, um die Klasse weiter zu verschlanken.

Im Vergleich dazu hatten in der alten Version 21 Klassen mehr als 352 Zeilen. Fünf davon hatten sogar mehr als 1000 Zeilen.

Lange Methode: Die längste Methode im neuem Scout ist *getSelectShape(SQLiteDatabase db, final Activity activity)* in der Klasse *DialogHelper.java* mit 51 Zeilen. Bei einer Refaktorisierung des neugeschriebenen Source Code könnte dies sicherlich verbessert werden. Es ist aber auf jeden Fall eine klare Verbesserung zum alten Source Code, in dem die längste Methode in der Klasse *ShowTabView.java* 230 Zeilen lang ist. Nun 7 Zeilen als durchschnittliche Länge einer Methode statt 14, ist ebenfalls, im Sinne dass alles klein sein sollte, eine klare Verbesserung.

Divergierende Änderungen: Sobald bemerkt wurde, dass eine Klasse divergierende Zuständigkeiten hat, wurde diese in mehrere separate Klassen zerlegt, was dem SRP folgt. Die Aufgaben der Klasse *FileAndCommunicationManager.java* aus dem alten Scout lassen sich zum Beispiel nun in den folgenden Klassen wiederfinden: *UserDirectory.java*, *ExternalStorageHelper.java*, dem gesamtem Package *view.dialogs* sowie *DialogHelper.java*, *TOTEMServer.java, DownloadGames FromServer.java*, *DownlaodMediaFiles.java*, *InternetBackgroundOperation.java* und *UploadGamesToServer.java*.

Schrotkugel herausoperieren und *Switch*-Befehl: Die Datentypen der Properties werden nun über Polymorphismus zur Laufzeit bestimmt. Dadurch fallen die entsprechenden *Switch*-Befehle weg und die Zuständigkeiten der einzelnen Properties bündeln sich in ihren Klassen (s. Abschnitt 4.3.2). Des Weiteren wurde darauf geachtet, dass ähnliche *Switch*-Befehle nie an unterschiedlichen Orten vorkommen.

Kommentare: Der neue Source Code des Scouts enthält keinen auskommentierten Source Code mehr. Des Weiteren wurde in der neuen Version jeder hinzugefügte Kommentar kritisch hinterfragt, ob er durch ein Refaktorisieren des entsprechenden Source Codes nicht überflüssig gemacht werden kann.

Falsche Zuständigkeit: Für eine klare Zuständigkeit wurde die Package-Struktur an das Model/View/Controller Pattern angelehnt (s. Abschnitt 4.3.3). Im Allgemeinen wurde darauf geachtet, dass der Source Code da ist, wo er gebraucht wird. So war zum Beispiel die Methode *getAllMediaUrls()* als statische Methode zuerst in der Klasse *PropertyHelper.java*. Später wurde bemerkt, dass diese Methode ausschließlich in der Klasse *DownloadGamesFromServer.java* verwendet wird und somit wurde die Methode dorthin verschoben.

Mentales Mapping: Das mentale Mapping ist durch das riesige *HashMap*-Objekt entstanden (s. Abschnitt 4.1.2). Da dieses im neuem Scout nicht mehr nötig ist (s. Abschnitt 4.3.1), ist dieser Smell automatisch weggefallen. Stattdessen gibt es für jede Property eine eigene Klasse, die intuitiv zu finden ist.

4.3.5 Veränderungen im Verhalten des Scouts

Beim Neuschreiben wurde auch das nach außen sichtbare Verhalten des Scouts mit geändert. Dies ist vor allem durch die in Abschnitt 4.1.3 gefunden Bugs motiviert. Des Weiteren wurde ein *isSynchronized-Feld* für die Properties hinzugefügt, was im ersten Abschnitt dieses Kapitels behandelt wird. Im zweiten Abschnitt werden die verwendeten Android Dialoge, welche teilweise irreführend waren, überarbeitet.

Fehlende Datentypen: Da für die Properties auf das OCP geachtet wurde, konnten die fehlenden Boolean- und Float-Datentypen einfach hinzugefügt werden. Dafür wurde eine Klasse *BooleanProperty.java* und eine Klasse *FloatProperty.java* angelegt, die beide das Property-Interface implementieren.

Default Marbles: Für die Default Marbles wurde eine eigene Datenbanktabelle angelegt. So können sie direkt von den normalen Marbles unterschieden werden und werden für die Liste der Marbles nicht mit geladen. Diese Unterscheidung in zwei Tabellen, statt über ein Datenbankfeld *isDefaultMarble*, festzuhalten wird auch für das Konzept des Designers empfohlen. Dies wird später in Abschnitt 5.4.2 weiter diskutiert.

***IsSynchronized*-Feld:** Des Weiteren wurde für jede Property ein *isSynchronized*-Feld hinzugefügt, durch welches sich bestimmen lässt, ob eine Property auf dem Scout geändert wurde. Beim Hochladen der Daten an den Designer können so ausschließlich die Daten hochgeladen werden, welche geändert wurden. Besonders bei Bildern und Videos, die größer sein können, entlastet dies den benötigten Datenverkehr, welcher bei mobilen Anwendungen minimiert werden sollte.

Fehlende Fehlerausgabe beim Upload: Statt nach dem Hochladen immer „upload successfull“ auszugeben, wird nun die Fehlernachricht vom Designer ausgelesen. Wenn diese nicht leer ist, wird sie dem Benutzer in einem Dialog angezeigt. Dies ist beispielsweise in Abbildung 32 zu sehen.

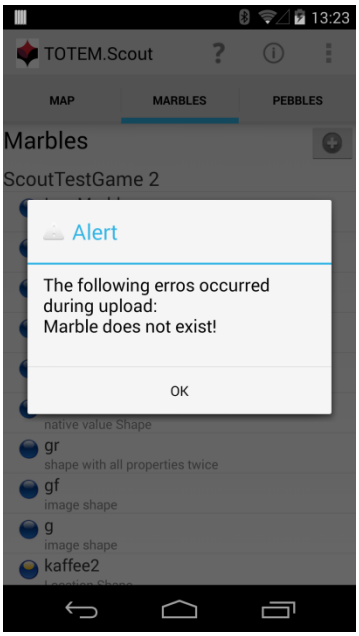

Abbildung 32. Fehlernachricht beim Upload

Die Fehlernachrichten, die zurzeit vom Designer ausgegeben werden, sollten jedoch noch überarbeitet werden, damit sie aussagekräftiger für den Benutzer sind. Der Grund für den Fehler aus Abbildung 32 ist der Folgende: Es wurde zuerst eine Marble auf den Scout heruntergeladen. Auf dem Designer wurde diese dann gelöscht. Im Scout wurde die Marble jedoch verändert und anschließend versucht hochzuladen. Der Designer versucht nun die Änderungen dieser Marble zu übernehmen und stellt fest, dass es die zu ändernde Marble nicht mehr gibt.

Leere Namen: Wird versucht einen leeren Namen zu speichern, erscheint nun eine Fehlermeldung, die besagt, dass ein Wert eingeben werden muss, wie in Abbildung 33 zu sehen ist.

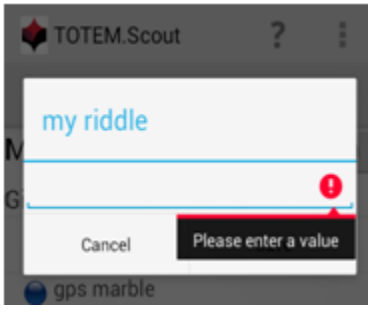

Abbildung 33. Fehlerüberprüfung leerer Name

Google Maps: Für die integrierte Google Maps wurde bei der neuen Version die Android-Maps-Extensions[37] verwendet, welche die Google Maps Android API v2 erweitert. Die Android-Maps-Extensions ermöglichen ein einfaches Clustering für sich überlappende Marker, wie in Abbildung 34 (links) zu sehen ist. Wird auf einen geclusterten Marker geklickt, öffnet sich ein Dialog (rechts), in dem einer der geclusterten Marker ausgesucht werden kann. Wird näher herein gezoomt, werden die geclusterten Marker sofort separat dargestellt, sofern im höheren Zoomlevel keine Überlappung mehr vorliegt. So sind nun alle Marker sichtbar und selektierbar.

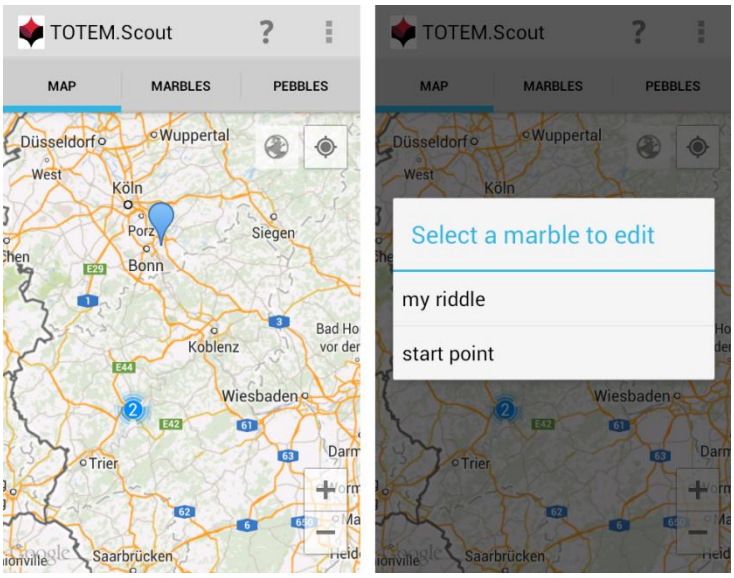

Abbildung 34. Die neue integrierte Google Maps

[37] („android-maps-extensions“)

Folgend werden noch die im alten Scout verwendeten **Android Dialoge thematisiert**. Direkt nach der Installation des Scouts öffnete sich der Dialog aus Abbildung 35, der darauf hinweist, dass beim Ausführen des Downloads alle Daten auf dem mobilen Gerät gelöscht werden. Gemeint sind jedoch alle im Scout gesammelten Daten. Direkt nach der Installation hat der Nutzer des Weiteren garantiert noch keinen Download gestartet. Bei mir hat dieser Dialog auf jeden Fall für einen kleinen Schock gesorgt.

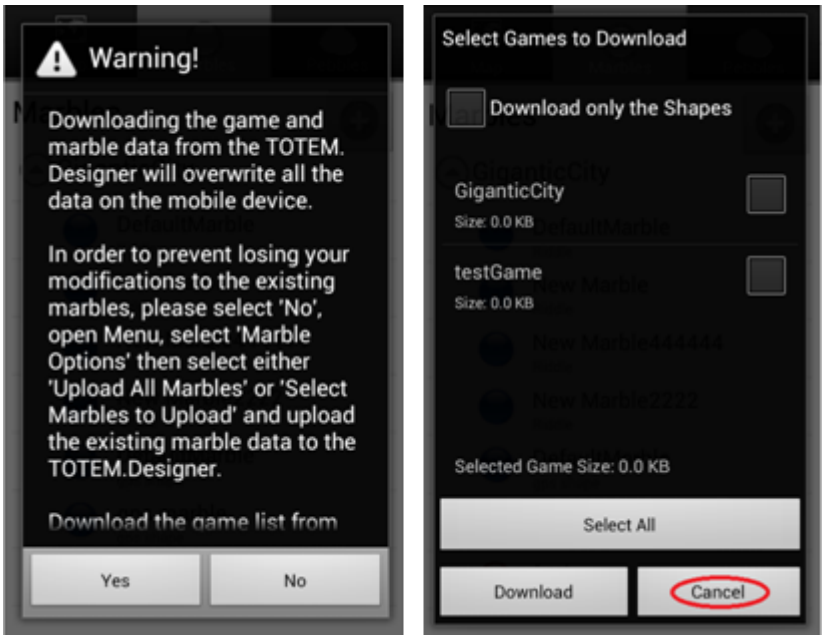

Abbildung 35. Irreführende Dialoge

Dieser Dialog erschien jedes Mal nach App-Start, sowie bevor Daten vom Designer heruntergeladen wurden. Wurde auf „Yes“ geklickt, wurde zum Dialog rechts weitergeleitet. Wurde dort jedoch auf „Cancel“ geklickt, so erschien wieder der Dialog links, was mit Sicherheit nicht erwartet wurde. Ich empfinde es auch als störend, diese Warnung jedes Mal zu erhalten und wegklicken zu müssen, bevor ein Spiel heruntergeladen werden kann.

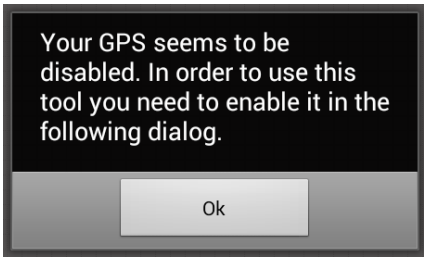

Abbildung 36. Enable GPS Dialog

Einen anderen Dialog, den ich kritisch sehe, ist in Abbildung 36 zu sehen. Dieser erschien nach jedem App-Start, wenn GPS im Handy ausgeschaltet ist, und leitet beim Klicken des OK-Buttons zu den Standortdiensten des Handys weiter. Wenn GPS-Koordinaten gesammelt werden sollen, muss natürlich GPS eingeschaltet sein. Wenn GPS erst

eingeschaltet wird, wenn es benötigt wird, für die integrierte Google Map oder für das Editieren eine GPS-Circle Property, so muss bis zu mehreren Minuten gewartet werden, bis das Handy ein GPS Signal gefunden hat.

Demgegenüber steht jedoch, dass aktuell sämtliche zu bekommende Daten systematisch ausgewertet werden, wie beispielsweise durch die NSA-Abhör-Affäre[38]. Daher sollte dem Anwender überlassen werden, ob er sein GPS dauerhaft an haben will, oder nur wenn es akut gebraucht wird und er dafür eine kurze Wartezeit akzeptiert.

In der neuen Version wurde daher für manche Dialoge die Möglichkeit hinzugefügt, diese nicht erneut anzuzeigen. So enthalten die Dialoge für die Download-Warnung und GPS-Meldung nun eine Checkbox, durch welche sie optional nicht wieder angezeigt werden. Dies ist in Abbildung 37 zu sehen ist.

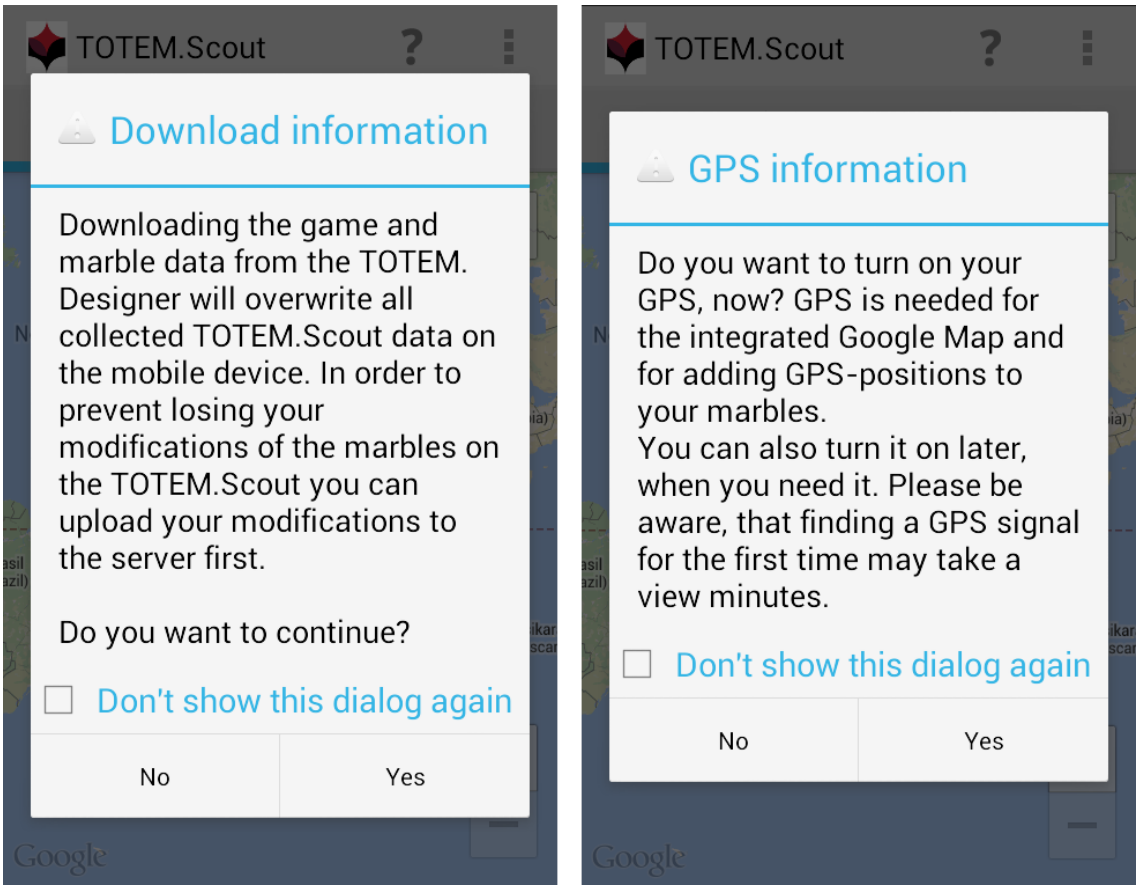

Abbildung 37. Dialoge mit Don't show again-Option

[38] („Wikipedia - Globale Überwachungs- und Spionageaffäre“)

Sollen sie eventuell doch wieder angezeigt werden, können die Häkchen wieder in den Preferences der App entfernt werden, wie in Abbildung 38 zu sehen ist.

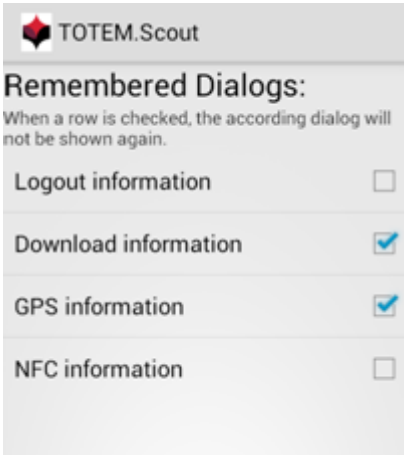

Abbildung 38. Remembered Dialog Preferences

4.4 Tests

Dieser Arbeit liegt ein beschränkter Zeitrahmen einer Bachelorarbeit von drei Monaten zugrunde. Vor allem da nicht, wie ursprünglich geplant, eine Refaktorisierung durchgeführt wurde, sondern ein Neuschreiben, ist dieser Zeitrahmen sehr knapp bemessen. Deswegen wurde zuerst auf ein schnelles Neuschreiben Wert gelegt, da es für wichtiger befunden wurde, nach drei Monaten eine vollständige, statt eine gut Test-abgedeckte, Anwendung zu haben. Es wurde also in gewisser Weise beim Testen der Pfad des geringsten Widerstands genommen. Das Testen wird noch in Abschnitt 5.1 genauer diskutiert.

Hier wird nur noch die Grundlage für das Testen geschaffen. Es wurde dazu ein Android-Testprojekt angelegt. In diesem wird gezeigt, wie Datenbank-Tests realisiert werden können und wie beispielsweise die fehleranfälligste Komponente, das Parsen der JSON-Dateien, getestet werden kann.

Diese Tests sind definitiv weiter auszubauen. Eine Strategie dafür wäre, vor jedem Ändern oder Erweitern des Source Codes für die entsprechenden Stellen zuerst Tests zu schreiben. Des Weiteren gilt, für den Fall das Bugs entdeckt werden, dass jeder Bug ein fehlender Test ist, der ergänzt werden muss.

4.4.1 Testen der Datenbank

Beim Testen der Datenbank ist es wichtig, dass die Datenbank nach den Tests genau den gleichen Inhalt hat, wie vor den Tests. Es könnte argumentiert werden, dass dies beim Scout nicht nötig ist, da seine Datenbank bloß als persistenter Zwischenspeicher fungiert. Es könnten alle Daten auch zum Designer hochgeladen werden und nach den Tests

wieder heruntergeladen werden. Da dies aber zum einen aufwändig ist und zum anderen vergessen werden kann, ist dies keine gute Argumentation.

```
public abstract class DatabaseTestCase extends AndroidTestCase {

    final String TAG = "DatabaseTest";
    protected static volatile TOTEMDatabase testDb;
    protected SQLiteDatabase db;

    @Override
    public void setUp() throws Exception {
        super.setUp();

        testDb = getIsolatedDatabase();
        db = testDb.getWritableDatabase();
        TOTEMDatabase.deleteTablesContent(db);
    }

    @Override
    public void tearDown() throws Exception {
        db.releaseReference();
        super.tearDown();
    }

    private TOTEMDatabase getIsolatedDatabase() {
        if (testDb == null) {
            synchronized (DatabaseTestCase.class) {
                if (testDb == null) {
                    try {
                        RenamingDelegatingContext isolatedContext = new RenamingDelegatingContext(getContext(), "test_");

                        Class[] totemDbConstructorParameters = new Class[]{Context.class};
                        Constructor<TOTEMDatabase> totemDBConstructor = TOTEMDatabase.class
                                .getDeclaredConstructor(totemDbConstructorParameters);
                        totemDBConstructor.setAccessible(true);

                        testDb = totemDBConstructor.newInstance(isolatedContext);

                    } catch (Exception e) {
                        e.printStackTrace();
                        fail("could not create an instance of the database");
                    }
                }
            }
        }
        return testDb;
    }
}
```

Abbildung 39. *DatabaseTestCase.java*

Stattdessen wird ein Test Case für Datenbanken erstellt. Dieser erzeugt eine separate Testdatenbank mit dem gleichem Schema wie die Produktionsdatenbank. Vor jedem Test wird diese Testdatenbank geleert, so dass alle Tests unabhängig von den zuvor getätigten Datenbankoperationen laufen. In Abbildung 39 ist dieser Test Case zu sehen.

Da der Konstruktor für die Datenbank als *private* deklariert wurde, kann er nicht direkt verwendet werden, sondern muss unter Zuhilfenahme von Reflektion zuerst temporär sichtbar gemacht werden.

```
public class TestIfAllTableWereCreated extends DatabaseTestCase {

    public void testTableGame() {
        testIfTableExists(TABLE.GAME);
    }

    public void testTableShape() {
        testIfTableExists(TABLE.SHAPE);
    }

    public void testIfTableExists(String tableName) {
        try {
            Cursor c = db.rawQuery("SELECT * FROM " + tableName, null);
        } catch (SQLiteException e) {
            if (e.getMessage().toString().contains("no such table")) {
                fail("table '" + tableName + "' does not exist");
            } else {
                e.printStackTrace();
                fail("SQLiteException");
            }
        }
    }
}
```

Abbildung 40. Test if all tables were created

Um nun beispielsweise zu testen, ob alle Tabellen erzeugt wurden, kann dieser Test Case verwendet werden. Aufgrund der Tatsache, dass die SQL-Create-Befehle aus Strings erzeugt wurden, ist dies sicherlich die fehleranfälligste Stelle der Datenbank. Der Test könnte wie in Abbildung 40 aussehen. Für jede Tabelle wird dann eine eigene Methode benötigt. Alternativ könnte auch über die Liste von allen Tabellen aus der Klasse *TOTEMDatabase.java* iteriert werden. Der Nachteil davon ist jedoch, dass nicht mehr alle Tabellen separat getestet werden, und nach einem Fehlschlagen die restlichen Tabellen nicht mehr getestet werden.

4.4.2 Testen des Parsens der JSON-Datei

Um den JSON-Parser zu testen, der den Inhalt der JSON-Datei in die Datenbank speichert, wurde eine JSON-Datei, die alle verschiedenen Properties mindestens einmal enthält, vom Designer heruntergeladen und in einen Testordner gelegt. Ein Ausschnitt dieser JSON-Datei ist im Anhang 2 zu sehen.

```
public class JSONParserForDownloadTest extends DatabaseTestCase {

    public void testParser() {
        String testJSONSaveLocation = Environment.getExternalStorageDirectory() + File.separator
                + UserDirectory.APPLICATION_ROOT_DIRECTORY + File.separator + "test";
        File jsonFile = new File(testJSONSaveLocation + File.separator + "games.json");
        assertTrue("There is no file 'games.json' in the directory 'TOTEM.Scout/test', which will be parsed",
                jsonFile.isFile());

        JSONParserForDownload.putIntoDatabase(db, testJSONSaveLocation, false);

        countRowTest(1, TABLE.GAME);
        countRowTest(4, TABLE.SHAPE);
        countRowTest(4, TABLE.DEFAULT_MARBLE);
        countRowTest(4, TABLE.MARBLE);

        countRowTest(8, TABLE.TEXT_PROPERTY);
        countRowTest(6, TABLE.INTEGER_PROPERTY);
        countRowTest(6, TABLE.FLOAT_PROPERTY);
        countRowTest(6, TABLE.BOOLEAN_PROPERTY);
        countRowTest(6, TABLE.IMAGE_PROPERTY);
        countRowTest(4, TABLE.AUDIO_PROPERTY);
        countRowTest(4, TABLE.VIDEO_PROPERTY);

        countRowTest(14, TABLE.LOCATION_PROPERTY);
        countRowTest(8, TABLE.GPS_CIRCLE_PROPERTY);
        countRowTest(6, TABLE.NFC_PROPERTY);
    }

    public void countRowTest(int expected, String tableName) {
        Cursor c = SQLCommands.getNamesAndIds(db, tableName);
        assertEquals(tableName, expected, c.getCount());
    }

}
```

Abbildung 41. JSON-Parser-Test für den Download

Der Test aus Abbildung 41 parst diese JSON-Datei und überprüft, dass die richtige Anzahl der jeweiligen Einträge anschließend in der Datenbank vorhanden sind. Ein weiterer Test sollte noch überprüfen, dass nicht nur die richtige Anzahl an Einträgen gespeichert wurden, sondern auch die richtigen Werte.

Da das Parsen hierarisch aufeinander aufbaut, das heißt, dass zum Beispiel das Parsen der Shapes nicht stattfindet, wenn das Parsen der Games fehlgeschlagen ist, ist es in diesem Fall sinnvoll, dass die rechtlichen assert-Methoden dann auch nicht mehr ausgeführt werden.

Der Parser, der den Inhalt der Datenbank in eine JSON-Datei schreibt, welche im Designer hochgeladen wird, ist genauso fehleranfällig wie der Parser für den Download und sollte dementsprechend auch getestet werden. Allerdings gibt es zurzeit kein Validierungskonzept für die JSON-Datei zum Upload. Die JSON-Datei hochzuladen und anschließend manuell zu überprüfen, ob alle Daten im Designer richtig integriert wurden, wäre eine Möglichkeit. Dies ist jedoch nicht empfehlenswert, da es zu einem zeitaufwendig ist und zum anderen die Produktionsdaten im Designer mit Testdaten vermischt. Deswegen sollte sich zuerst ein Konzept überlegt werden, wie die JSON-Datei automatisiert validiert werden kann, ohne dabei die Daten in den Designer hochzuladen.

4.5 Bewertung der neuen Version

Nachdem zuvor die Verbesserungen beim Neuschreiben des Source Codes beschrieben wurden, wird hier der neue Source Code abschließend bewertet.

Die wichtigste Verbesserung ist, dass nun leicht neue Properties hinzugefügt werden können, wie auch im nächsten Abschnitt gezeigt wird. Dafür wurde sich an das OCP und „Program to an interface, not an implementation“ gehalten (s. Abschnitt 4.3.2).

Eine weitere wichtige Verbesserung ist, dass die Verständlichkeit und die Wartbarkeit erhöht wurden. Dafür wurde eine neue Package-Struktur eingeführt (s. Abschnitt 4.3.3) und auf eine einheitliche Formatierung geachtet. Des Weiteren wurde darauf geachtet, Code-Duplizierung zu vermeiden. Dass darauf geachtet wurde, die vorhandenen Smells nicht zu wiederholen (s. Abschnitt 4.3.4), fördert ebenfalls die Wartbarkeit.

In Abschnitt 3.2.3 wurde zwar über die Qualität von Metriken gestritten, aber dass die Anzahl aller Zeilen im Source Code von über 15.000 um zweidrittel auf gut 5.000 reduziert wurde, ist trotzdem bemerkenswert. Im Sinne, dass alles klein sein sollte, stellt dies definitiv eine Verbesserung dar.

Des Weiteren wurden viele Bugs beseitigt und für eine bessere Benutzererfahrung die Dialoge überarbeitet (s. Abschnitt 4.3.5).

Der Bug Ewige temporäre ID (s. Abschnitt 4.1.3) besteht hingegen weiterhin. Da der Designer die neuen IDs nicht zurück gibt, muss zuerst der Designer überarbeitet werden, was jedoch nicht Teil dieser Arbeit ist.

Es wurde auch, genau wie in der alten Version, auf eine akzeptable Testabdeckung verzichtet. Dies ist ein Mangel am Source Code, den es im weiteren Projektverlauf zu beheben gilt, damit zukünftig verstärkt Refaktorisiert werden kann.

Des Weiteren wurden die Pebbles noch nicht implementiert und auf die WLAN-Features, welche noch nicht im Designer integriert sind, erst einmal verzichtet.

Somit lässt sich abschließend festhalten, dass der Source Code erheblich verbessert wurde, im Bereich der Tests jedoch noch nachgebessert werden muss, sowie sekundäre Features noch zu implementieren sind.

4.6 Integration der Natural Europe-Werkzeuge

Die Motivation für diese Arbeit war, den Source Code des Scouts zu verbessern, damit er anschließend leicht erweitert werden kann. Konkret sollten die spezialisierten Natural

Europe-Werkzeuge[39] (NE-Werkzeuge) wieder in die TOTEM-Werkzeuge integriert werden.

Von den spezialisierten NE-Werkzeugen soll das neue Indoor-Raummodell in die TOTEM-Werkzeuge generalisiert werden. Im Rapid-Prototyping-Prozess der NE-Werkzeuge wurde dazu bewusst das simple Modell aus Abbildung 42 verwendet. Ein Gebäude hat demnach einen Namen und besteht aus beliebig vielen Fluren. Ein Flur hat ebenfalls einen Namen und optional dazu einen Flur-Plan. Jeder Flur besteht wiederum aus beliebig vielen Räumen.

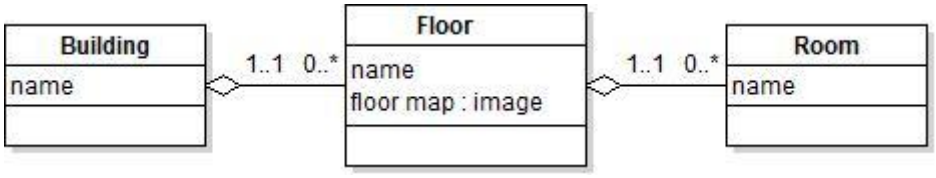

Abbildung 42. Indoor-Raummodell

Für dieses simple Raummodell wurde im Scout nun eine Indoor-Property (*LocationChildIndoor.java*) hinzugefügt, die das Property Interface implementiert. Es ist ein *Location Child*, da in den TOTEM-Werkzeugen ein Ort aus beliebig vielen Orten bestehen kann. So kann zum Beispiel ein Ort aus einer GPS-Koordinate und einem Indoor-Raum bestehen.

Die *edit*-Methode der Indoor-Property startet die Activity *EditIndoor.java*, in der eine Indoor-Property bearbeitet werden kann. Abbildung 43 zeigt zwei Screenshots dieser Activity.

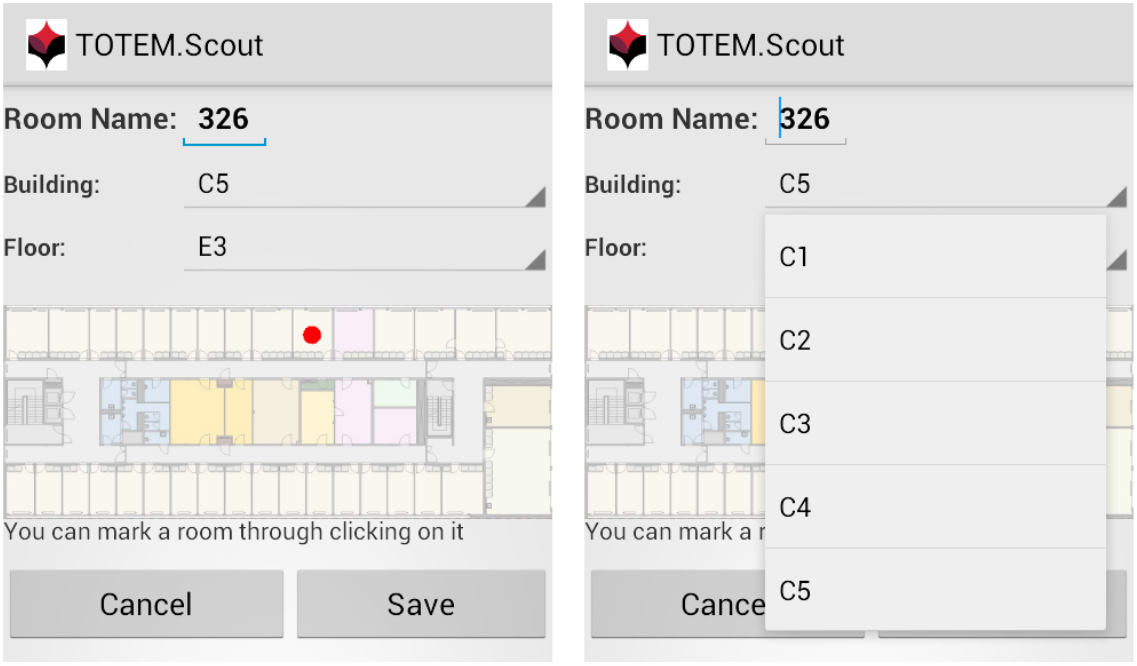

Abbildung 43. Edit Indoor-Property

[39] („NaturalEurope.Designer“)

Um die Indoor-Property dem Datenmodell des Scouts hinzuzufügen, musste noch eine Instanz von ihr in der Klasse *TOTEMDatabase.java* erzeugt werden. Somit besteht die einzige Änderung im bereits bestehenden Source Codes in einem Hinzufügen von *new LocationChildIndoor()*. Ansonsten mussten nur die Indoor-Property-Klasse und die Activity zum editieren dieser Property hinzugefügt werden.

Es sei erwähnt, dass der Designer natürlich ebenfalls erweitert werden muss, da der Scout als Datensammler für ihn dient. Die Erweiterung des Designers ist jedoch nicht Teil dieser Arbeit.

5 Diskussion

In diesem Kapitel wird auf Aspekte eingegangen, die über den praktischen Teil dieser Arbeit hinaus gingen. Zuerst wird das Testen noch einmal thematisiert. Anschließend werden Alternativen zum persistenten Speichern der Daten zur relationalen Datenbank diskutiert. Danach wird zur Dokumentation der TOTEM-Werkzeuge ihr Datenmodell zusammengefasst. Als Nächster Punkt werden Vorschläge zur Überarbeitung der TOTEM-Werkzeuge gemacht sowie mögliche zukünftige Erweiterungen erwähnt. Abschließend folgt eine Selbstreflexion über diese Arbeit.

5.1 Testschwierigkeiten

Wie bereits in Abschnitt 4.4 erwähnt, wurde beim Neuschreiben auf automatisierte Unit-Tests zunächst verzichtet. In der Theorie zum Refaktorisieren wurde erwähnt, dass automatisierte Tests zum Refaktorisieren eine Vorbedingung sind. Da jedoch nicht refaktorisiert wurde und in dem zeitlich begrenzten Rahmen die Priorität zuerst auf das Neuschreiben gelegt wurde, erschien das sinnvoll. Bei ersten Überlegungen, wie nachträglich automatisierte Unit-Tests integriert werden können, sind jedoch Schwierigkeiten aufgefallen, die hier erörtert werden.

5.1.1 Der TOTEM.Scout ist schwer durch Unit-Tests zu testen

Der Scout ist eine reine Datensammler-App. Somit besteht er aus einer Datenbank und View-Elementen, welche die Daten anzeigen. Für die Kommunikation zwischen der Datenbank und der View wird des Weiteren Multithreading benötigt. Datenbanken, Views und Threading sind jedoch schwer durch Unit-Tests zu testen (s. Abschnitt 3.5.5).

Des Weiteren ist die Ortsbestimmung ein wichtiges Feature des Scouts. Diese lässt sich jedoch nicht akkurat simulieren und sich somit auch nicht gut automatisiert testen. Auch Features wie das Foto- oder Video-Aufnehmen benötigen menschliche Interaktionen und lassen sich somit nicht akkurat automatisiert testen.

Das Hoch- und Herunterladen von Daten beim Designer ist dazu ein wesentlicher Bestandteil des Scouts. Jedoch lässt sich auch dies nicht optimal testen, da das Hoch- und Herunterladen lange dauert und somit dem Prinzip widerspricht, dass Tests schnell sein sollten (s. Abschnitt 3.5.4). Dadurch dass die Unit-Tests jedes Mal durch das Android-Test-Framework zuerst auf ein Gerät installiert werden müssen, dauern die Tests bereits alleine durch das Installieren mehrere Sekunden.

Beim Testen des Hochladens zum Designer ist des Weiteren darauf zu achten, dass keine Testdaten im Designer verbleiben.

5.1.2 Unit-Tests sollten zuerst geschrieben werden

Wie der Name des Prinzips F.I.R.S.T (s. Abschnitt 3.5.4) und das „T“ in diesem, welches für Timely steht, aussagen, sollten Tests vor dem Source Code oder zumindest sehr zeitnah zum Source Code geschrieben werden. Das hat unter anderem das Ziel, schwer zu testenden Source Code zu vermeiden.

Da sich zunächst auf das Neuschreiben konzentriert wurde, und nicht auf die Tests, wurde auf die Testbarkeit nicht geachtet. Stattdessen wurde zum Beispiel eine strenge Kapselung umgesetzt und viele private Methoden angelegt. Zum Testen müssen diese durch die Reflection API zur Laufzeit sichtbar gesetzt werden, wie in Abbildung 39. *DatabaseTestCase.java* zu sehen war. Das erhöht jedoch die Komplexität der Tests und vermindert somit ihren Wert.

Anhand der Testklasse *DatabaseTestCase.java* lässt sich auch eine weitere aufgetretene Schwierigkeit beim Testen erläutern. Um eine Datenbank zu erzeugen, reicht der Application Context. Diesen hat die Datenbankklasse *TOTEMDatabase.java* verwendet, um die Datenbank zu erzeugen. Somit gab es keinen Konstruktor, welchem ein Context-Objekt übergeben werden kann, anhand dessen die Datenbank erzeugt werden soll. Um jedoch eine isolierte Datenbank zu erzeugen, wird auch ein isoliertes Context-Objekt benötigt. Deswegen musste der Konstruktor refaktorisiert werden, so dass er auf einem übergebenen Context-Objekt die Datenbank erzeugt.

5.1.3 Was getestet werden kann

Um die Datenbank isoliert testen zu können, wurde der Test Case *TOTEM Database.java* in Abbildung 4.19 vorgestellt. Damit kann schon einmal die Datenbank getestet werden. Des Weiteren können die *getter*- und *setter*-Methoden der Controller-Klassen damit getestet werden, welche aus der Datenbank lesen und schreiben. Wenn in den Views nun, in Anlehnung an die *Humbling Dialog Box* aus Abschnitt 3.5.5, ausschließlich diese *getter*- und *setter*-Methoden verwendet werden, ist es zwar nicht optimal aber akzeptable, dass die Views nicht automatisiert getestet werden.

Da jedoch beim Neuschreiben nicht auf eine leichte Testbarkeit geachtet wurde, sei erwähnt, dass der Source Code wahrscheinlich vor dem Test Schreiben dahingehend refaktorisiert werden muss, dass er leichter getestet werden kann. Des Weiteren wird noch, um das Hochladen von Daten zu testen, ein Konzept benötigt, wie die hochzuladenden Daten validiert werden können. Dies könnte zum Beispiel, wenn statt dem JSON-Format ein XML-Format verwendet wird, durch ein XML-Schema geschehen.

5.2 Alternative Möglichkeiten zur Persistenten Speicherung

Aus den pragmatischen Gründen, dass die alte Scout-Version bereits auf einer relationalen Datenbank aufbaut und, dass ich selber Erfahrungen mit relationalen Datenbanken habe, wurde die persistente Speicherung durch eine relationale Datenbank beibehalten. Da die Datenbank mit Abstand der größte Aufwand beim Neuschreiben war, werden hier Alternativen erörtert und gegenüber gestellt.

5.2.1 Binäre Speicherung

Bei einer binären Speicherung werden die zu speichernden Objekt-Zustände in einen Byte-Stream geschrieben, aus welchem diese später wieder herausgelesen werden können. Dies ist zum Beispiel durch Java Object Serialization[40] oder Hessian[41] möglich.

Der Vorteil davon ist, dass dies ohne großen Aufwand möglich ist und wird beispielsweise für entfernte Methodenaufrufe oder Sessions verwendet.

Diese Technik hat jedoch auch einige Nachteile. Es können stets nur alle Binärdaten gelesen oder geschrieben werden. Somit muss zum Beispiel, wenn von einem großen Objekt nur eine Variable geändert wurde, das gesamte Objekt zum Speichern zurückgeschrieben werden, was sich in der Performance niederschlägt. Als weiterer Nachteil besteht das Risiko, dass die gespeicherten binären Daten nicht mehr gelesen werden können, wenn sich die Objektdefinition, beispielsweise durchs Umbenennen einer Variablen, geändert hat. (Ullenboom 2010, Kap. 17.10)

5.2.2 Speicherung in einer XML-Datei

Objekte können auch in einer XML-Datei gespeichert und später wieder daraus gelesen werden. Dies hat im Gegensatz zu der binären Speicherung den Vorteil, dass die gespeicherten Daten in menschlich lesbarer Form sind und auch nur Teile gelesen und zurückgeschrieben werden können.

Durch beispielsweise die Java Architecture for XML Binding[42] (JAXB) können Objekte in einer XML-Datei gespeichert und später wieder daraus gelesen werden. Durch ein Eclipse Plugin[43] können aus einem JAXB-Schema sogar die entsprechenden JavaBeans, die zum Mapping benötigt werden, automatisch generiert werden.

Die Performance ist dabei jedoch kritisch zu hinterfragen. Eine relationale Datenbank ist sicherlich performanter als das Schreiben in und Lesen aus einer XML-Datei.

[40] („Serializable (Java Platform SE 6)“)
[41] („Hessian Binary Web Service Protocol“)
[42] („Java Architecture for XML Binding“)
[43] („JAXB Eclipse Plug-In“)

5.2.3 Objektrelationales Mapping

Dieser Abschnitt beruht vor allem auf Überlegungen von (Neward 2006) und (Fowler 2012).

In-Memory-Objekten und persistenten Daten in einer relationalen Datenbank liegen unterschiedliche Designs und somit auch unterschiedliche Eigenschaften zugrunde. Dies wird auch als *Object-Relational Impedance Mismatch* bezeichnet. Folgend werden beispielhaft drei Unterschiede zwischen dem Objekt- und dem Relational-orientierten Design genannt:

- **Assoziationen** werden in beiden Designs genau andersherum definiert. Im Objekt-orientierten Design werden Teil-Objekte vom gesamten Objekt heraus referenziert, so dass die Teil-Objekte nicht wissen, dass sie Teil eines anderen Objektes sind. Im Relational-orientierten Design hingegen geschieht dies durch einen Fremdschlüssel in den kleineren Teilen zum Gesamten.
- Das Objekt-orientierte Design beinhaltet Techniken zur **Vererbung** und zum **Polymorphismus**, welche zum Beispiel in Design Patterns sehr wichtig sind. Eine relationale Datenbank unterstützt diese Techniken jedoch nicht nativ.
- Das relationale Design unterstützt das Konzept von **Transaktionen**, das heißt, dass Änderungen an Daten erst nach einem *commit* übernommen werden und vorher vollständig durch ein *rollback* zurückgenommen werden können. Im Objekt-orientierten Design wird dies nicht nativ unterstützt, was insbesondere beim Multithreading zu einer großen Komplexität führen kann.

Das Objektrelationale Mapping hat nun das Ziel, die Unterschiede zwischen dem relationalen und dem Objekt-orientierten Design zu überbrücken, so dass ein Objekt-orientiertes Programm seine Daten persistent in einer relationalen Datenbank speichern kann. Das Objektrelationale Mapping versteckt dabei nach Möglichkeit die relationale Sicht, so dass der Programmier idealerweise nur mit Objekten arbeitet. Für Android ließe sich das Objektrelationale Mapping beispielsweise durch greenDao[44] realisieren.

Ein Objektrelationales Mapping ist jedoch oft komplex und eine weitere Abstraktionsebene, die verstanden werden muss. Die Performance dabei ist auch meist nicht optimal. Des Weiteren ist das Objektrelationale Mapping nicht vollständig, so dass für komplexere Ausdrücke, wie zum Beispiel *Inner Joins*, oft trotzdem SQL verwendet werden muss.

Ein weiterer Nachteil ist, dass das Datenmodell beim Objektrelationalem Mapping in der Datenbank sowie in den Objekten vorhanden sein muss, was dem DRY-Prinzip

[44] („greenDAO – Android ORM for SQLite“)

widerspricht und bei der Wartung und Weiterentwicklung Mehraufwand zur Folge haben kann.

(Neward 2006) geht soweit, dass er behauptet, dass die Unterschiede zwischen dem Objekt-orientierten und dem relationalen Design so groß sind, dass sie nicht einfach durch ein Objektrelationales Mapping vereint werden können. So bezeichnet er das Objektrelationale Mapping sogar als „*The Vietnam of Computer Science*“.

(Fowler 2012) relativiert dies. Er sagt, dass das Objektrelationale Mapping zu 80% das Mapping zwischen dem relationalen und Objekt-orientierten Design abnimmt. Der Programmierer muss sich eben bewusst sein, dass es nur 80% und nicht 100% sind und für die rechtlichen 20% werden Experten benötigt.

Beide stimmen auf jeden Fall überein, dass eine Lösung des Objektrelationalen Mapping-Problems ist, entweder nur Objekte oder nur eine relationale Datenbank zu verwenden. Wenn zum Beispiel nur Daten für ein Userinterface gelesen und zurück geschrieben werden und dabei einfache Logik verwendet wird, die durch SQL-Abfragen ausgedrückt werden kann, wird auch nur eine relationale Datenbank benötigt.

5.2.4 Bewertung der Alternativen

Aufgrund der in Abschnitt 5.2.1 aufgeführten Nachteile der binären Speicherung, wird diese in der Regel nicht für eine persistente Speicherung verwendet. Somit ist sie für den Scout ungeeignet.

Werden die Daten für den Scout in einer relationalen Datenbank gespeichert, so ist nach Abschnitt 5.2.3 ein Objektrelationales Mapping nicht nötig und wäre somit bloß eine unnötige Abstraktionsebene. Denn der Scout erledigt nichts anderes, als die Daten zu lesen, anzuzeigen und wieder zurückzuschreiben, was sich durch SQL unkompliziert erledigen lässt.

Das persistente Speichern in einer XML-Datei ist jedoch eine sehr interessante Alternative. Vor allem da der Datenaustausch mit dem Designer durch JSON realisiert ist und er somit sowieso mit JSON arbeiten muss, was sich leicht in XML überführen lässt.

Abbildung 44 zeigt das vorhandene Arbeiten mit der JSON-Datei für das Hoch- und Herunterladen mit dem Designer. Zurzeit exportiert der Designer eine JSON-Datei mit den gewünschten Daten. Der Scout speichert diese JSON-Datei vollständig in seine Datenbank. Beim Sammeln von Daten mit dem Scout werden dann die neuen Daten in die Datenbank geschrieben und bereits bestehende modifiziert. Abschließend schreibt der Scout die Daten aus der Datenbank wieder in eine JSON-Datei, welche vom Designer ausgelesen wird.

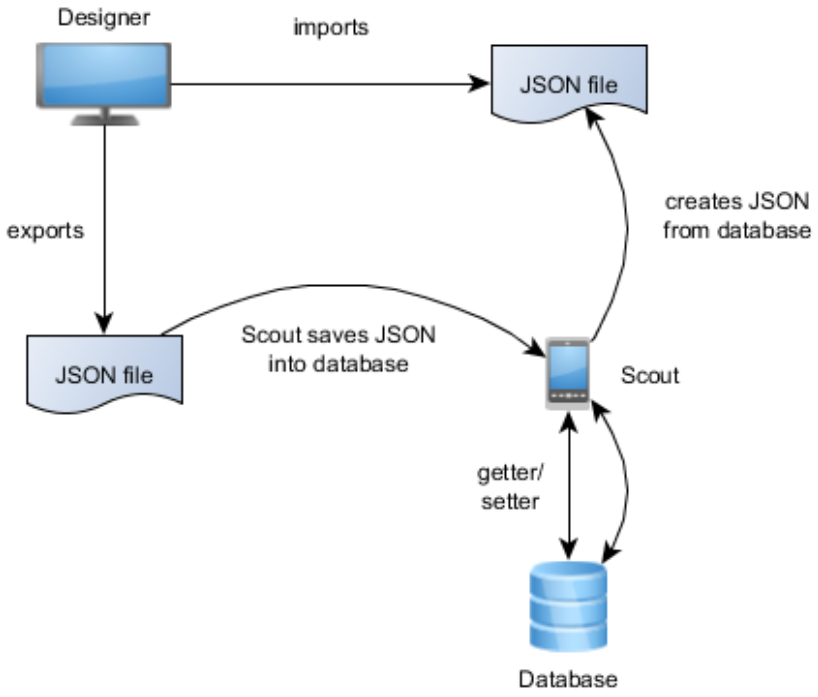

Abbildung 44. Scout mit JSON und Datenbank

Exportiert der Designer stattdessen eine XML-Datei, kann der Scout nun die persistente Speicherung durch diese XML-Datei realisieren. Die Datenbank-Schicht fällt so weg, wodurch eine Abstraktionsebene eingespart wird, wie in Abbildung 45 zu sehen ist.

Abbildung 45. Scout mit XML

Des Weiteren wären so Änderungen im Model des Designers einfacher in den Scout zu integrieren, da sich die benötigten JavaBeans automatisch aus einer vom Designer exportierten XML-Datei erzeugen lassen.

Die fehlende Thread-Sicherheit beim Zugriff auf die XML-Datei, die bei der Android-SQL-Datenbank von Android gegeben ist, könnte durch das Template Method Pattern, ähnlich

wie beim Datenbankzugriff aus Abschnitt 4.3.1, erreicht werden, in dem das Modifizieren in einer Queue eingereiht wird. Diese führt dann nacheinander alle Modifikationen aus.

Die Performance von dieser Lösung müsste jedoch getestet werden. Bei einem Test[45], mit einem Pentium Dual-Core CPU, 2.1 GHz und Windows 7, wurde eine XML-Datei bestehend aus 10.000 Personen geparst. Die Personen bestanden aus einem ID-Tag und einem Namen-Tag. Es wurde dazu weniger als 1/10 Sekunde benötigt. Dies ist vielversprechend, jedoch nicht auf ein mobiles Gerät übertragbar.

5.3 Dokumentation

Zum Einarbeiten in die TOTEM-Werkzeuge wurde mir nur ein Paper gegeben, welches die Idee der TOTEM-Werkzeuge beschreibt (Jurgelionis u. a. 2012), sowie der Source Code der TOTEM-Werkzeuge. Für ein besseres Verständnis der inneren Struktur, fing ich nach einer Zeit an, mir Klassendiagramme des Models zu erstellen.

Um den Nächsten das Einarbeiten in die nicht trivialen TOTEM-Werkzeuge zu erleichtern, soll hier übersichtlich der Kern des Datenmodells gezeigt werden. Es sei erwähnt, dass Dokumentation genauso wie Kommentare (s. Abschnitt 3.2.4 Smell „Kommentare“), gewartet werden muss, damit sie nicht veraltet und somit Fehlinformationen gibt. Ein gut lesbarer Source Code, welcher einer der Ziele des Neuschreibens war, ist des Weiteren auch als Dokumentation zu betrachten, welche beim Einarbeiten sehr hilfreich ist.

[45] (Staveley 2011)

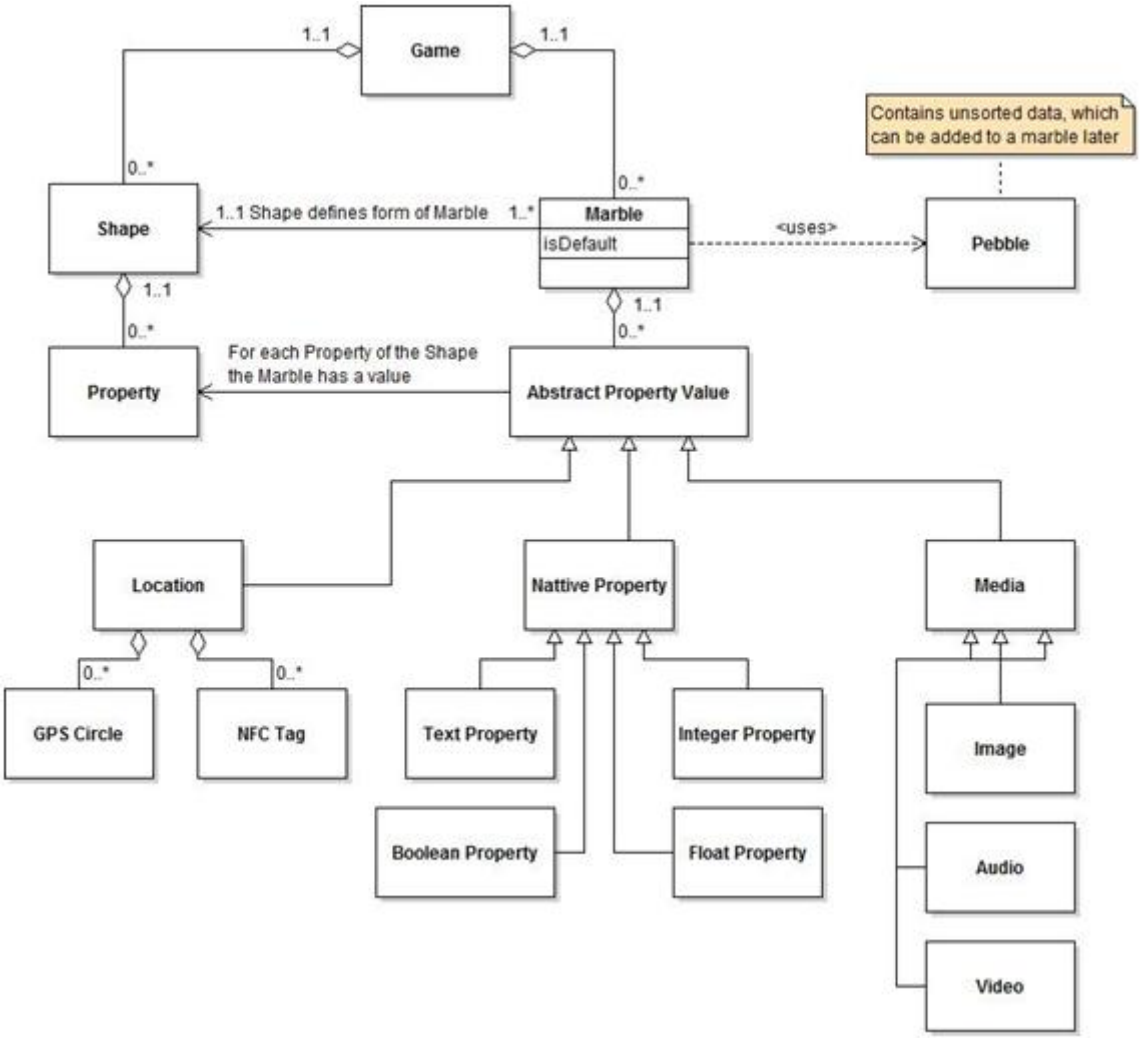

Abbildung 46. Datenmodell des Designers

Abbildung 46 zeigt das Datenmodell, welches direkt an die konzeptuellen Idee aus (Jurgelionis u. a. 2012) angelehnt ist. Der Designer setzt dieses Model eins zu eins um. Es sei hier schon einmal erwähnt, dass beim Neuschreiben des Scouts Verbesserungsideen für die Umsetzung des Models gekommen sind, welche in Abschnitt 5.4.2 erörtert werden. Dementsprechend wurde das Model des Scouts, welches ebenfalls in Abschnitt 5.4.2 dokumentiert wird, bereits modifiziert.

Ein Spiel hat beliebig viele Shapes und Marbles. Eine Shape beinhaltet beliebig vielen Datentypen, welche als Properties der Shape bezeichnet werden. Eine Marble kann logisch als Instanz der Shape bezeichnet werden, dass heißt, dass sie alle Properties seiner Shape hat. Zu jeder Shape gibt es genau eine Default Marble, für die das Feld *isDefault* auf *true* gesetzt ist. Für alle anderen hat das Feld den Wert *false*. Die Default Marble gibt für die Properties der Shape einen Default-Wert an. Wird eine neue Marble zu einer Shape erstellt, werden für die neue Marble alle Werte der entsprechenden Default Marble gesetzt.

Die Werte einer Property, sind wie folgt in die drei Kategorien Orte, einfache Werte und Media-Daten unterteilt:

- Ein Ort besteht aus beliebig vielen GPS-Kreisen und NFC-Tags.
- Die einfachen Werte sind Text, eine Ganzzahl, eine Gleitkommazahl, oder ein Boolescher Wert.
- Die Media-Daten sind Bilder, Audios oder Videos.

Die Pebbles bestehen aus einem Titel, einem Tag, einem Text, einem GPS-Kreis, sowie aus einer Audio-Datei, einem Video und einem Bild, was zu Gunsten der Übersichtlichkeit nicht eingezeichnet wurde. Nach dem Tag lassen sich die Pebbles in der View des Designers filtern.

5.4 Vorschläge zur Überarbeitung der TOTEM-Werkzeuge

Beim Neuschreiben des Scouts sind Verbesserungsideen für die TOTEM-Werkzeuge gekommen, welche folgend erläutert werden.

5.4.1 Simplizität und Refaktorisieren

Simplizität ist ein wichtiges Kriterium für guten Source Code (s. Abschnitt 3.2). Die Berücksichtigung von Features, die erst in Zukunft gebraucht werden, können den Source Code komplizierter machen, wie in den folgenden zwei Beispielen zu sehen ist:

1. Ein abstraktes Ortsmodell, welches einen abstrakten Trigger-Mechanismus für Orte beinhaltet. In (Putschli 2012) wurde ein entsprechendes Modell erarbeitet. In den Scout wurde dieses Modell in das Package *.locationModel* kopiert. Das einzige, was aus diesem Package jedoch im neuen Scout aktuell verwendet wird, sind zehn Zeilen, die einen gescannten NFC-Tag in einen String zu parsen.
2. Eine Ort-Property besteht im Modell aus beliebig vielen Orten (s. Abbildung 46). Benötigt wurde bisher jedoch nur, dass eine Ort-Property entweder genau eine GPS-Koordinate oder ein NFC-Tag ist. In den Views des Designers ist auch nur diese 1:1-Beziehung implementiert, wie in Abbildung 47 zu sehen ist.

Abbildung 47. Location View im Designer

Es kann also nur direkt der GPS-Kreis bzw. der NFC-Tag editiert werden und nicht die Ort-Property, zu welcher der GPS-Kreis bzw. der NFC-Tag gehört.

Im Scout werden jedoch auch die Ort-Properties angezeigt, wie in Abbildung 48 zu sehen ist.

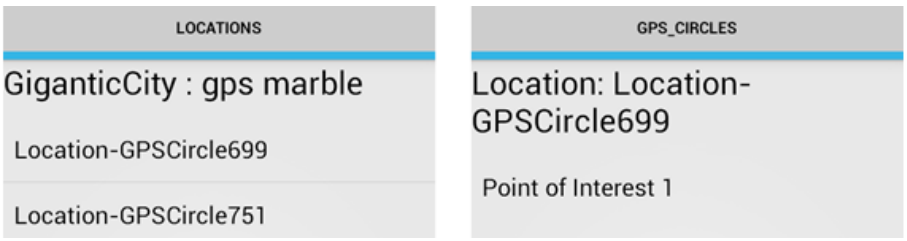

Abbildung 48. Links zwei Ort-Properties, rechts die zugehörigen Orte zur ersten Orts-Property

Sie haben dabei, da sie nicht im Designer editiert werden können, ihren Default-Namen, welcher sich aus der Typ-Bezeichnung und ihrer Property-ID zusammen setzt. Für den Anwender der im Designer *die Points of Interest 1* und *2* angelegt hat, ist es sicherlich verwirrend, im Scout stattdessen die Bezeichnungen *Location-GPSCircle699* und *751* zu finden, hinter denen sich die *Points of Interest* verbergen.

Anhand dieser zwei Beispiele möchte ich vorschlagen, stets nur das umzusetzen, was konkret benötigt wird. Denn jedes Feature vergrößert den Source Code, welcher verstanden und gewartet werden muss. Wie im zweiten Beispiel zu sehen ist, können unausgereifte Features des Weiteren zu Verwirrungen führen. Unterstützt wird dies von den folgenden Punkten:

- Im agilen Manifest[46] steht: „**Simplicity**--the art of maximizing the amount of work not done--is essential."
- In den Praktiken von Extreme Programming[47], welche auf dem agilen Manifest aufbauen, sind die Punkten **Simple Design**, **Consider the Simplest Thing That Could Possibly Work** und **YAGNI** enthalten.
- **YAGNI**[48] (You aren't gonna need it): Etwas Kompliziertes sollte nur umgesetzt werden, wenn es konkret gebraucht wird.
- **KISS**[49] (Keep it simple, stupid): Systeme funktionieren am besten, wenn sie simpel gehalten sind.

Wenn auf ein **Simples Design** aufgebaut wird, hat dies den Vorteil, dass der Source Code leicht zu warten und weiterzuentwickeln ist. Unter anderem verringert dies auch das Risiko, dass ein aufwendiges Neuschreiben nötig wird, wie in dieser Arbeit erforderlich war.

[46] (Beck u. a.)
[47] (Martin 2014, S. 11ff.)
[48] („Wikipedia - You Aren't Gonna Need It")
[49] („Wikipedia - KISS Principle")

Wird dann ein Feature konkret gebraucht, ist es einfach dieses in das simple Design einzuarbeiten (s. auch Fowler 2005, S. 57f.). Anschließend sollte **sofort refaktorisiert** werden, damit das Design simpel bleibt. So entsteht idealerweise ein regelmäßiger Zyklus, in dem sich die Integration neuen Source Codes und die Refaktorisierung des gesamten Source Codes abwechseln, wobei beide Schritte jeweils mit vertretbarem Aufwand durchgeführt werden können. Des Weiteren können sich dann neue Programmierer aufgrund des simplen Designs leicht einarbeiten.

5.4.2 Überarbeitung des Models

Beim Neuschreiben des Scouts und dem regelmäßigen Sich-selbst-Refaktorisieren, ist aufgefallen, dass sich das logische Datenmodell der TOTEM-Werkzeuge (s. Abbildung 46) im Source Code elegant zusammenfassen lässt und nicht eins zu eins umgesetzt werden muss. Dies wird folgend erläutert.

Die Shape ist zwar ein wichtiges Konzept für den Anwender, im Source Code jedoch redundant, da eine Shape bereits vollständig durch die entsprechende Default Marble beschrieben wird, die sowieso für jede Shape angelegt wird. Somit kann die Shape vollständig im Source Code durch die Default Marble ersetzt werden. In der Shape View kann dann, statt der Shape, die Default Marble bearbeitet werden, welche im Hintergrund immer mit der Shape modifiziert werden musste. Das Konzept der Shapes sowie die Views können dabei unverändert bleiben, so dass der Anwender keine Änderung feststellen kann. Im Source Code hingegen fallen die Shapes und die Properties weg, wodurch dieser leichtgewichtiger wird. Dies kommt dem Prinzip nach, dass alles klein sein sollte sowie dem DRY-Prinzip.

Die Default Marble sollte dazu in einer eigenen Tabelle gespeichert werden. So kann im Source Code jedes Mal die Abfrage *isDefaultMarble==true* eingespart werden. Des Weiteren kann so einfach sichergestellt werden, dass es zu jeder Shape genau eine Default Marble gibt. Bei der Unterscheidung durch ein boolesches Feld könnten theoretisch mehrere Marbles Default Marble sein. Durch die eigene Tabelle für Default Marbles wird auch das Konzept hervorgehoben, dass die Default Marble nun an die Stelle der Shape tritt. Nebenbei sind die Datenbank Operationen dadurch performanter, da nun nur die Default Marbles bzw. nur die Marbles durchsucht werden müssen.

Das so verschlankte Datenmodell ist in Abbildung 49 zu sehen.

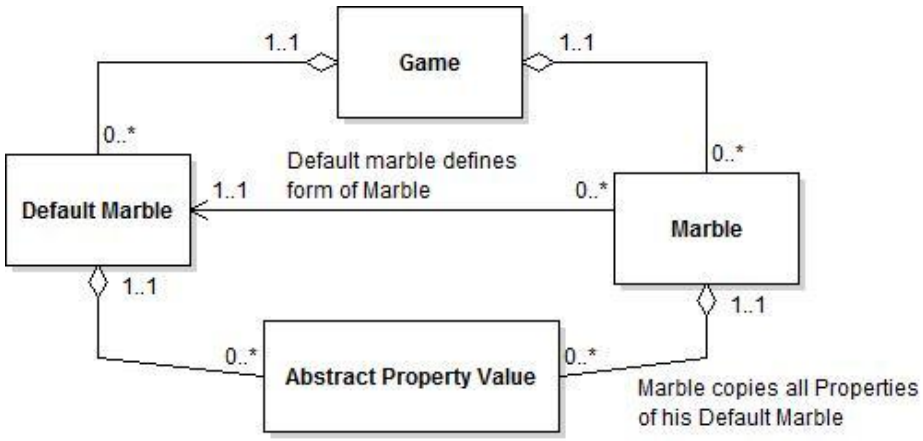

Abbildung 49. Shape durch Default Marble ersetzt

Des Weiteren ist aufgefallen, dass die Unterscheidung der Property-Werte in Orte, native Properties und Media im Source Code nicht optimal ist. Properties, deren Werte sich in einer Spalte einer Datenbank speichern lassen, sind im Source Code nahezu identisch. Die Media Properties werden durch einen Pfad zu ihrer Datei im Dateisystem gespeichert und gehören somit dazu. Die Gemeinsamkeiten dieser Properties können daher in einer Oberklasse zusammengefasst werden.

Somit wurde das Datenmodell aus Abbildung 50 im neuen Source Code des Scouts umgesetzt.

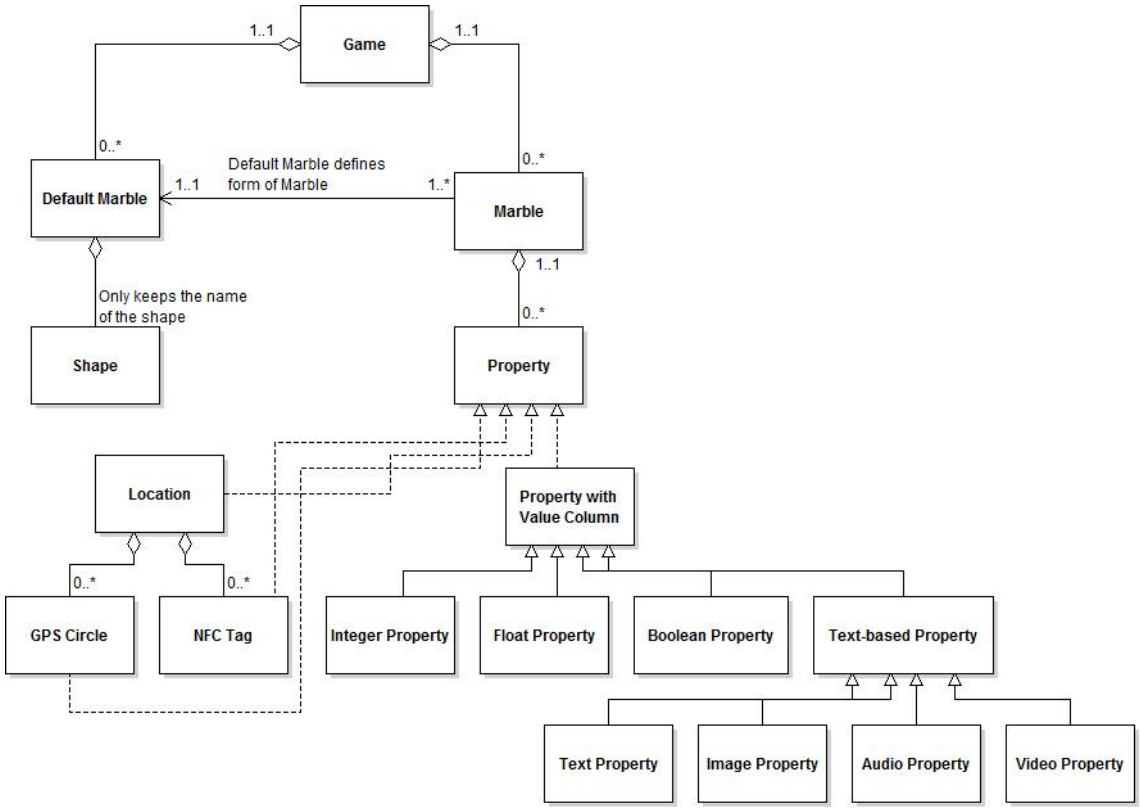

Abbildung 50. Datenmodell vom Scout

Die Shape-Tabelle wurde dabei noch nicht ganz entfernt. Dies hat praktische Gründe, da der Shape-Name, damit der Anwender keinen Unterschied merkt, beibehalten werden muss. Dieser Name hätte auch im Namen der Default Marbles gespeichert werden können, beim Parsen der JSON-Dateien wäre dies jedoch ein kleiner Mehraufwand gewesen.

Es wird vorgeschlagen, diese Vereinfachung des Datenmodells auch im Designer umzusetzen. Wenn der Designer dann den Shape-Namen in der Default Marble speichert und die Shape komplett aus dem Source Code entfernt hat, kann die Shape auch beim Scout wegfallen.

5.4.3 Zukünftige Verbesserungen für die TOTEM-Werkzeuge

Hier werden nun weitere Verbesserungen und mögliche Erweiterungen für die TOTEM-Werkzeuge erörtert und vorgeschlagen.

Wenn als Alternative zur relationalen Datenbank im Scout eine **persistente Speicherung in einer XML-Datei** umgesetzt wird, würde dies den Source Code des Scouts vereinfachen (s. Abschnitt 5.2.4).

Idealerweise ließen sich der **Designer und Scout in der gleichen Sprache implementieren**. Zurzeit werden für den Designer Kenntnisse in Django und somit auch Python benötigt und für den Scout Kenntnisse in Android. Dadurch waren in der Regel für den Designer und den Scout unterschiedlichen Programmierer zuständig. So ist zum Beispiel zu erklären, dass es im Designer die Datentypen Boolean und Float gibt, welche im Scout nicht vorhanden waren. Durch eine einheitliche Programmiersprache wird der Aufwand die Werkzeuge zu pflegen und weiterzuentwickeln erheblich reduziert. In (Oppermann 2008) wird der Vorteil einer plattformübergreifenden Programmiersprache ebenfalls hervorgehoben.

Idealerweise müsste sogar nur ein Programm implementiert werden, welches auf einem mobilen Gerät als Scout funktioniert und in einer Desktop-Umgebung als Designer. So müssten nur die Views in Abhängigkeit der Umgebung angepasst werden, und das Model und der Controller könnten identisch sein.

Ein sehr interessanter Ansatz dafür wäre, nur die Django-basierte Web-Lösung zu verwenden. Die APIs von Android, zum Beispiel zur Ortsbestimmung, könnten als Objekt gekapselt in JavaScript zur Verfügung gestellt werden. Ein simples Beispiel wie durch JavaScript Android-Code aufgerufen werden kann, ist in („Android Developers - Building Web Apps in WebView“) zu finden. Um den Scout dann allerdings weiterhin ohne aktive Internetverbindung verwenden zu können, müsste zusätzlich eine offline Cache-Strategie

umgesetzt werden. Genauere Forschungen zu diesem Thema halte ich für vielversprechend.

Globale Shapes wären ein neues Feature, um das Teilen des gesammelten Inhalts zu erweitern. Zurzeit sind die Shapes an ein Spiel gebunden und können somit nicht über ein Spiel hinaus geteilt werden. Statt wie bisher über einen Fremdschlüssel an ein Spiel gebunden zu sein, könnten sie über ein Fremdschlüssel an einer User Gruppe gebunden sein. Somit könnte eine Shape innerhalb einer User Gruppe über ein Spiel hinaus geteilt werden. Dazu müsste noch ein Berechtigungskonzept für die einzelnen User und User-Gruppen erarbeitet werden.

Das **Indoor Model** wurde in der neuen Version als eigener Property-Typ umgesetzt. Dabei setzt sich ein Indoor-Raum aus den bereits bestehenden Datentypen von Text und optional ein Bild als Flur Plan, sowie einer Koordinate als Marker auf diesem Bild, zusammen. Durch die globalen Shapes könnte eine Indoor-Property dementsprechend auch als eine Shape „Indoor Room“ umgesetzt werden, welche aus den entsprechenden Properties besteht. Somit würden wieder Komponenten eingespart, welche ansonsten stets mit gewartet werden müssten.

Der **Synchronisationsalgorithmus** zwischen dem Designer und dem Scout ist zurzeit simpel gehalten. Zum Beispiel wird folgendes Szenario nicht behandelt:

> Ein Benutzer *A* lädt eine Marble vom Designer auf den Scout herunter. Dort setzt er einen Wert *a*. Ein Benutzer *B* lädt dieselbe Marble herunter und setzt dort den Wert *b*. Lädt nun zuerst der Benutzer A diese Marble wieder hoch und anschließend der Benutzer B, so ist der Wert *a* verloren gegangen.

Durch die Einführung einer Versionsnummer, die bei jedem Hochladen inkrementiert wird, ließe sich dieses Szenario entdecken und eine Fehlermeldung ausgeben.

Eine weitere Verbesserung wäre es, wenn beim Hochladen der Marbles ein Fehler auftritt, die entsprechenden **Fehlermeldungen für den Anwender aussagekräftig zu formulieren**, da diese nun im Scout ausgegeben werden (s. Abbildung 32).

Eine Option, welche der Anwender setzen kann, um **beim Download die Media-Dateien wegzulassen**, wäre eine weitere sinnvolle Ergänzung. Diese sollen wahrscheinlich sowieso nicht im Scout editiert werden. Stattdessen werden hauptsächlich Media-Daten vor Ort gesammelt. Die meist großen Media-Daten nicht mit herunterzuladen, entlastet den beschränkten Datenverkehr auf dem mobilen Gerät. Statt die großen Media-Daten könnten auch kleinere Thumbnails heruntergeladen werden.

Ein weiteres mögliches Einsatzszenario für die TOTEM-Werkzeuge wäre eine **Validierung der Ortsbestimmung**. Wie bereits in Abschnitt 2.2.1 nebenbei erwähnt, ist

zu berücksichtigen, dass die Ortsbestimmung durch GPS ungenau ist. (Oppermann 2009) beschäftigt sich genauer mit dieser Problematik. So könnte zum Beispiel durch den Scout, während mit ihm vor Ort herumgelaufen wird, die Stärke des GPS-Empfangs geloggt und später auf einer Karte visualisiert werden.

Abschließend sei darauf verwiesen, dass die TOTEM-Werkzeuge zukünftig garantiert erweitert werden. So bereiten bereits zwei Kollegen ihre Bachelorarbeit im Kontext der TOTEM-Werkzeuge vor. Einer wird die Ortsbestimmung um iBeacons[50] erweitern und ein anderer eine Integration an den Junaio Browser[51] erarbeiten.

5.5 Selbstreflexion der Arbeit

In diesem Abschnitt wird über die eigene Arbeit reflektiert.

5.5.1 Die Fragestellung

Die Motivation dieser Arbeit war, dass festgestellt wurde, dass der Source Code des TOTEM.Scout einem historisch gewachsenen System entspricht. Somit war er schwer zu warten und zu erweitern. Deswegen sollte der Source Code in dieser Arbeit refaktorisiert werden.

Dazu wurde sich in Literatur über guten Source Code und das Refaktorisieren eingearbeitet. Das Resultat der anschließenden Ist-Zustands-Analyse des Source Codes war jedoch, dass ein Neuschreiben einfacher ist, als ein refaktorisieren.

Wie in Abschnitt 3.4.2 erwähnt, sollten keine Extraphasen für das Refaktorisieren eingeplant werden. Das wurde damit begründet, dass das Refaktorisieren keinen Selbstzweck hat und dementsprechend nur im Zusammenhang durchgeführt werden sollte. Nach der Ist-Zustands-Analyse sehe ich dafür noch einen wichtigen zweiten Grund. Bis es zu der Extraphase für das Refaktorisieren kommt, kann es bereits schlichtweg zu spät dafür sein. Refaktorisierungen sind kleine Schritte, die stets sofort durchgeführt werden sollten.

Dementsprechend hätte das Refaktorisieren bereits am Anfang kritischer hinterfragt werden sollen. Statt dem Stichwort „*Refaktorisierung*“ im Titel dieser Arbeit wäre das Stichwort „*Verbesserung des Source Codes*“ somit im Nachhinein zweifelsfrei zutreffender gewesen.

5.5.2 Das Vorgehen

Im Nachhinein lässt sich der Punkt, dass zuerst ausführlich das Theoriekapitel zu guten Source Code und über das Refaktorisieren geschrieben wurde, auch mit einem *Big Design Upfront* der Bachelorarbeit vergleichen. Es wurde davon ausgegangen, dass die

[50] („Wikipedia - iBeacon“)
[51] („Junaio – Augmented Reality Browser“)

Theorie fast vollständig im Vorhinein erarbeitet werden kann und die Praxis anschließend nur noch auf diese aufbauen muss. Stattdessen sehe ich es nun so, dass alles ein Lernprozess ist. Im Rahmen dieses Lernprozess wurde *gelernt*, dass der Source Code des Scouts nicht refaktorisiert, sondern neu geschrieben werden sollte. Dass die Theorie des Refaktorisierens angeschaut wurde und anschließend neu geschrieben wurde, zeigt, wie falsch so ein *Big Design Upfront* sein kann.

Es sei erwähnt, dass die Theorie des Refaktorisierens jedoch keinesfalls nutzlos war. Da alles ein Lernprozess ist, kamen beim Neuschreiben ständig neue Erkenntnisse, welche durch Refaktorisieren umgesetzt werden konnten. Ein Beispiel dafür ist die modifizierte Aufteilung der Properties aus Abschnitt 5.4.2.

5.5.3 Der Zeitrahmen und das Testen

Aufgrund der Tatsache, dass neu geschrieben werden musste und nicht refaktorisiert, war die Zeit für diese Arbeit zu kurz.

Deswegen wurden kurz vorm Ende die letzten Features unter Zeitdruck eingebaut. Ein Beispiel dafür ist die Methode *getShapeAdapter* der Klasse *Shape.java*, welche nicht als guter Source Code bezeichnet werden kann. Direkt vor einer Deadline ist dies nach Abschnitt 3.4.3 akzeptabel. Bevor jedoch neue Features eingebaut werden, sollten diese wenigen Stellen sofort nach Abgabe dieser Arbeit refaktorisiert werden.

Auch wurden einige wenige sekundäre Features noch nicht implementiert.

Das des Weiteren auf Unit-Tests verzichtet wurde, ist nicht so leicht nachzuholen. Ursprünglich wurde gedacht, dass diese einfach später hinzugefügt werden können. Im Rahmen meines Lernprozesses während dieser Arbeit sehe ich hingegen nun ein, dass für Testbarkeit designed werden muss (s. auch Abschnitt 4.4 und 5.1).

Um die Wichtigkeit von Unit-Tests zu unterstreichen sei erwähnt, dass diese bereits gegen Ende der Arbeit vermisst wurden, da nach jeder kleinen Refaktorisierung der gesamte Ablauf zeitaufwendig manuell getestet werden musste.

Deswegen würde ich, wenn ich noch einmal mit dem Neuschreiben anfangen würde, sofort Unit-Tests schreiben und ggf. weitere Features erst einmal weglassen. Ich bin der Überzeugung, dass dies langfristig viel Zeit spart und kurzfristig nicht wesentlich langsamer ist. (s. auch Vigenschow 2010, S. 187ff.)

6 Zusammenfassung

In dieser Arbeit wurde die Notwendigkeit von Werkzeugen zur Unterstützung des Erstellungsprozesses von ortsbasierten Spielen erläutert. Insbesondere die Wichtigkeit von Werkzeugen die auch vor Ort verwendet werden können, wurde hervorgehoben.

In diesem Kontext sind in einem mehrjährigen Projekt die TOTEM-Werkzeuge entstanden, welche in dieser Arbeit vorgestellt wurden. Sie sind Werkzeuge, um das Sammeln und Strukturieren von ortsbasierten Inhalt zu unterstützen, die vielseitig eingesetzt werden können. Sie sind in diesem Kontext die ersten Werkzeuge, die vollständig ineinander integrierte Unterstützung für die Desktop-Umgebung und für vor Ort bieten.

Das Warten und Erweitern dieser TOTEM-Werkzeuge war mit dem Laufe der Zeit jedoch immer schwieriger geworden. Dies gilt besonders für den Source Code der komplizierteren Android-Anwendung TOTEM.Scout. Deswegen sollte sich diese Arbeit mit der Refaktorisierung des Scouts befassen.

Dazu wurde erläutert, was guter Source Code ist, und das Refaktorisieren vorgestellt. Auf der Basis von dieser Theorie wurde der Source Code des Scouts analysiert. Es wurde dabei festgestellt, dass es einfacher ist, den Scout neuzuschreiben, statt ihn zu refaktorisieren.

Bei dem neugeschriebenen Scout wurde gezeigt, dass die Schwachstellen der vorherigen Analyse nicht wiederholt wurden. Besonders wichtig dabei war die Umsetzung des Open-Closed-Prinzips für die Datentypen, so dass einfach neue Datentypen hinzugefügt werden können. Eine anschließende Erweiterung des Scouts um ein Indoor-Raummodell, die einfach durchzuführen war, bekräftigt die Qualität des neugeschriebenen Source Codes.

Somit wurde in dieser Arbeit das Warten und Weiterentwickeln des Scouts erheblich vereinfacht. Konkret ist demnächst eine Anbindung an den Augmented Realitiy Browser Junaio geplant sowie die Erweiterung der Ortsbestimmungsmöglichkeiten um iBeacons. Durch diese Arbeit wurde der dafür benötigte Aufwand definitiv reduziert.

Literaturverzeichnis

Online Quellen:

„Android Developers - AsyncTask“. Zugegriffen am 16. April 2014. http://developer.android.com/reference/android/os/AsyncTask.html.

„Android Developers - Building Web Apps in WebView“. Zugegriffen am 19. April 2014. http://developer.android.com/guide/webapps/webview.html.

„Android Developers - Keeping Your App Responsive“. Zugegriffen am 16. April 2014. http://developer.android.com/training/articles/perf-anr.html.

„Android Developers - NumberPicker“. Zugegriffen am 16. April 2014. http://developer.android.com/reference/android/widget/NumberPicker.html.

„Android Developers - Progress & Activity“. Zugegriffen am 16. April 2014. http://developer.android.com/design/building-blocks/progress.html.

„Android Developers - Testing Fundamentals“. Zugegriffen am 16. April 2014. http://developer.android.com/tools/testing/testing_android.html.

„android-maps-extensions“. Zugegriffen am 20. April 2014. https://code.google.com/p/android-maps-extensions/.

„Awaitility“. Zugegriffen am 16. April 2014. https://code.google.com/p/awaitility/.

„Code-Inside Blog“. Zugegriffen am 16. April 2014. http://blog.codeinside.eu/2011/11/27/pragmatische-softwareentwicklung/.

„Django“. Zugegriffen am 16. April 2014. https://www.djangoproject.com/.

„Eclipse documentation“. Zugegriffen am 16. April 2014. http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Fcodestyle%2Fref-preferences-formatter.htm.

„Google Maps Android API v2“. Zugegriffen am 16. April 2014. https://developers.google.com/maps/documentation/android/.

„greenDAO – Android ORM for SQLite“. Zugegriffen am 16. April 2014. http://greendao-orm.com/.

„Hessian Binary Web Service Protocol“. 2014. Zugegriffen am 16. April 2014. http://hessian.caucho.com/index.xtp.

„IPCity“. Zugegriffen am 14. April 2014. http://ipcity.fit.fraunhofer.de/.

„IPCity » Time Warp“. Zugegriffen am 9. April 2014. http://ipcity.fit.fraunhofer.de/?page_id=10.

„Java Architecture for XML Binding“. Zugegriffen am 20. April 2014. http://www.oracle.com/technetwork/articles/javase/index-140168.html.

„JAXB Eclipse Plug-In“. *SourceForge*. Zugegriffen am 16. April 2014. http://sourceforge.net/projects/jaxb-builder/.

„Jester“. Zugegriffen am 16. April 2014. http://jester.sourceforge.net/.

„Junaio – Augmented Reality Browser“. Zugegriffen am 16. April 2014. http://www.junaio.com.

„Junit 3 vs Junit 4 Comparison“. Zugegriffen am 5. Januar 2014.http://www.asjava.com/junit/junit-3-vs-junit-4-comparison/.

„The size of the World Wide Web“. Zugegriffen am 16. April 2014. http://www.worldwidewebsize.com/.

„Metrics 1.3.6“. Zugegriffen am 16. April 2014. http://metrics.sourceforge.net/.

„Natural Europe“. Zugegriffen am 16. April 2014. http://www.natural-europe.eu/.

„NaturalEurope.Designer“. Zugegriffen am 16. April 2014. http://naturaleurope.fit.fraunhofer.de/nedesigner/.

„Portal Hunt“. Zugegriffen am 17. April 2014. http://www.totem-games.org/?q=portalhunt%20.

„Refactoring-Tools in Eclipse“. Zugegriffen am 20. April 2014. http://wiki.fernuni-hagen.de/eclipse/index.php/Refactoring-Tools_in_Eclipse.

„Serializable (Java Platform SE 6)“. Zugegriffen am 16. April 2014. http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html.

„The Java™ Tutorials - The Reflection API“. Zugegriffen am 16. April 2014. http://docs.oracle.com/javase/tutorial/reflect/.

„Tidy City“. Zugegriffen am 16. April 2014. http://totem.fit.fraunhofer.de/tidycity/.

„TOTEM Summer School“. Zugegriffen am 17. April 2014. http://www.totem-games.org/?q=node/145.

„Wikipedia - Content-Management-System“. Zugegriffen am 16. April 2014. http://de.wikipedia.org/w/index.php?title=Content-Management-System&oldid=126739285.

„Wikipedia - Globale Überwachungs- und Spionageaffäre“. Zugegriffen am 16. April 2014. http://de.wikipedia.org/wiki/Globale_%C3%9Cberwachungs-_und_Spionageaff%C3%A4re.

„Wikipedia - Heinzelmännchen“. Zugegriffen am 16. April 2014. http://de.wikipedia.org/w/index.php?title=Heinzelm%C3%A4nnchen&oldid=128608858.

„Wikipedia - iBeacon“. Zugegriffen am 16. April 2014. http://de.wikipedia.org/w/index.php?title=IBeacon&oldid=129104486.

„Wikipedia - KISS Principle“. Zugegriffen am 16. April 2014. http://en.wikipedia.org/w/index.php?title=KISS_principle&oldid=600054720.

„Wikipedia - You Aren’t Gonna Need It“. Zugegriffen am 16. April 2014. http://en.wikipedia.org/w/index.php?title=You_aren%27t_gonna_need_it&oldid=602240408.

„WordPress“. Zugegriffen am 16. April 2014. http://de.wordpress.org/.

„Zwergenwelten“. Zugegriffen am 16. April 2014. http://zwergenwelten.net/.

Referenzen:

Beck, Beedle, van Bennekum, Cockburn, Cunningham, Fowler, Grenning, u. a. 2014. „Manifesto for Agile Software Development“. Zugegriffen am 16. April 2014. http://agilemanifesto.org/iso/en/principles.html.

Buxton und Sniderman. 1980. „Iteration in the Design of the Human- Computer Interface“. Toronto, Kanada.

Feng. 2011. „Mobile Autorenwerkzeuge zur Erstellung ortsbasierter Spiele“. Bachelorarbeit, Dortmund: Fachhochschule Dortmund.

Fischer, Lindt und Stenros. 2006. „Final Crossmedia report (part II) – Epidemic Menace II Evaluation Report“. Integrated Project on Pervasive Gaming.

Fowler. 2005. „Refactoring“. München: Addison-Wesley Verlag.

Fowler. 2012. „OrmHate“. Zugegriffen am 16. April 2014. http://martinfowler.com/bliki/OrmHate.html.

Gamma, Helm, Johnson und Vlissides. 2007. „Design Patterns“. 5. Aufl. Boston: Addison-Wesley.

Groundspeak, Inc. 2014. „Geocaching - The Official Global GPS Cache Hunt Site“. Zugegriffen am 14. April 2014. http://www.geocaching.com/.

Hamill. 2005. „Unit Test Frameworks“. Sebastopol: O’Reilly.

Hull, Clayton und Melamed. 2004. „Rapid Authoring of Mediascapes“. Bristol, UK.

Jurgelionis, Oppermann, Blum und Wetzel. 2012. „SHAPES, MARBLES AND PEBBLES: TEMPLATE-BASED CONTENT FOR LOCATION-BASED GAMES“. Sankt Augustin.

Kaner und Bond. 2004. „Software Engineering Metrics: What Do They Measure and How Do We Know?“. Zugegriffen am 16. April 2014. http://testingeducation.org/a/metrics2004.pdf

Kerievsky. 2004. „Refactoring to Pattern“. Amsterdam: Addison Wesley.

Lippert und Roock. 2004. „ Refactorings in großen Softwareprojekten“. Heidelberg: dpunkt.verlag GmbH.

Martin. 1996. „The Open-Closed Principle“. Zugegriffen am 16. April 2014. http://objectmentor.com/resources/articles/ocp.pdf.

Martin. 2009. „Clean Code“. Heidelberg: mipt-Verlag.

Martin. 2014. „Agile software development, principles, patterns, and practices“. Harlow: Pearson.

Meszaros. 2007. „xUNIT TEST PATTERNS“. Boston: Addison Wesley.

Milgram und Kishino. 1994. „A TAXONOMY OF MIXED REALITY VISUAL DISPLAYS“. In: IEICE Transactions on Information Systems*, Vol E77-D*.

Montola, Stenros und Waern. 2009. „Pervasive Games“*: Theory and Design*. United States of America: Elsevier Inc.

Neward. 2006. „The Vietnam of Computer Science“. Zugegriffen am 16. April 2014. http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx.

Oppermann. 2008. „On the choice of programming languages for developing location-based mobile games“. München.

Oppermann. 2009. „Facilitating the Development of Location-Based Experiences“. University of Nottingham.

Putschli. 2012. „Möglichkeiten der Indoorlokalisierung mit NFC und WLAN“. Bachelorarbeit, Sankt Augustin: Hochschule Bonn-Rhein-Sieg.

Simon, Seng und Mohaupt. 2006. „Code-Qualitiy-Management“. dpunkt.verlag.

Staveley, Alex. 2011. „JAXB, SAX, DOM Performance“. Zugegriffen am 16. April 2014. http://dublintech.blogspot.de/2011/12/jaxb-sax-dom-performance.html.

Ullenboom. 2010. „Java ist auch eine Insel“. 9. Auflage. Galileo Computing. Zugegriffen am 28. Februar 2014. http://openbook.galileocomputing.de/javainsel9/index.htm#_top.

Vigenschow. 2010. „Testen von Software und Emmbedded Systems“. Heidelberg: dpunkt.verlag GmbH.

Weal, Hornecker, Cruickshank, Michaelides, Millard, Halloran, De Roure und Fitzpatrick. 2006. „Requirements for In-Situ Authoring of Location Based Experiences“. Helsinki, Finland.

Anhang

Anhang 1 - Pattern Happiness Beispiel

```
public interface MessageStrategy {
    public void sendMessage();
}

public abstract class AbstractStrategyFactory {
    public abstract MessageStrategy createStrategy(MessageBody mb);
}

public class MessageBody {
    Object payload;

    public Object getPayload() {
        return payload;
    }

    public void configure(Object obj) {
        payload = obj;
    }

    public void send(MessageStrategy ms) {
        ms.sendMessage();
    }
}

public class DefaultFactory extends AbstractStrategyFactory {
    private DefaultFactory() {
        ;
    }

    static DefaultFactory instance;

    public static AbstractStrategyFactory getInstance() {
        if (instance == null)
            instance = new DefaultFactory();
        return instance;
    }

    public MessageStrategy createStrategy(final MessageBody mb) {
        return new MessageStrategy() {
            MessageBody body = mb;

            public void sendMessage() {
                Object obj = body.getPayload();
                System.out.println((String) obj);
            }
        };
    }
}

public class HelloWorld {
    public static void main(String[] args) {
        MessageBody mb = new MessageBody();
        mb.configure("Hello World!");
        AbstractStrategyFactory asf = DefaultFactory.getInstance();
        MessageStrategy strategy = asf.createStrategy(mb);
        mb.send(strategy);
    }
}
```

Abbildung 51. Pattern Happiness (Nach Kerievsky 2004, S. 24f.)

Anhang 2 - Ausschnitt einer heruntergeladenen JSON-Datei


```
{
    "games": [
        {
            "author_id": 30,
            "author_name": "knauf",
            "description": "test description",
            "id": 36,
            "marbles": [
                {
                    "id": 816,
                    "name": "DefaultMarble",
                    "properties": {
                        "audios": [],
                        "booleans": [
                            {
                                "id": 6886,
                                "property_id": 707,
                                "property_name": "Boolean",
                                "value": false
                            }
                        ],
                        "floats": [
                            {
                                "id": 6885,
                                "property_id": 706,
                                "property_name": "Float",
                                "value": 0.0
                            }
                        ],
                        "images": [],
                        "integers": [
                            {
                                "id": 6884,
                                "property_id": 705,
                                "property_name": "Integer",
                                "value": 0
                            }
                        ],
                        "locations": [],
                        "texts": [
                            {
                                "id": 6883,
                                "property_id": 704,
                                "property_name": "Text",
                                "value": ""
                            }
                        ],
                        "videos": []
                    },
                    "shape_id": 128
                },
```


Abbildung 52. games.json