ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Implementierung einer mobilen Anwendung zur Unterstützung von Tinnitus-Patienten

Masterarbeit an der Universität Ulm

Vorgelegt von:
Michael Lindinger
michael.lindinger@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Dr. Winfried Schlee

Betreuer:
Marc Schickler

2014

Fassung 22. Mai 2014

Kurzfassung

Tinnitus ist ein Symptom, bei dem betroffene Personen dauerhaft Töne oder Geräusche wahrnehmen, ohne dass es dafür eine physikalische Ursache gibt. Ein großes Problem bei Tinnitus ist die fehlende Standardtherapie. Tinnitus-Patienten müssen daher in vielen Fällen zahlreiche Therapien durchlaufen, bis eine Besserung eintritt. Die lange Suche nach einer geeigneten Therapie frustriert nicht nur viele Patienten, sondern wirkt sich auch negativ auf deren psychische Verfassung aus. Um diese Situation zu verbessern, wurde die App Tinnitus Navigator entwickelt, die Tinnitus-Patienten, basierend auf den in die App eingegebenen Daten, geeignete Therapievorschläge erstellt.
Diese Arbeit besteht aus zwei großen Teilen. Zunächst wird die Webanwendung vorgestellt, über die Regeln und Therapievorschläge zentral verwaltet werden können. Dazu wird detailliert auf die zugrunde liegenden Strukturen eingegangen und gezeigt, wie das Recommender-System in eine bereits existierende Server-Umgebung integriert wurde. Anschließend werden die wichtigsten Funktionen von Tinnitus Navigator aus technischer und grafischer Sicht vorgestellt. Zu diesen Funktionen zählen unter anderem die persönliche Tinnitusakte, über die Fragebögen, Hörtests und Ereignisse eingegeben werden können und der Kalender zur Verwaltung aller eingegebenen Daten. Abschließend wird der Algorithmus, der mit Hilfe von ECA-Regeln Therapievorschläge erstellt, im Detail beschrieben.

Inhaltsverzeichnis

1

Einleitung

Tinnitus (lat. Klingeln) ist ein Symptom, bei dem betroffene Personen dauerhaft Töne oder Geräusche wahrnehmen. Studien zufolge ist ungefähr jeder Zehnte von solchen Ohrengeräuschen dauerhaft betroffen. Bei rund 1% der Gesamtbevölkerung ist der Tinnitus überdies so stark, dass die Lebensqualität dieser Personen spürbar darunter leidet [KVL13]. Dieses Leiden wirkt sich außerdem in vielen Fällen negativ auf die Psyche betroffener Personen aus, sodass Begleiterscheinungen wie Depressionen, Schlafstörungen oder Angststörungen bei Tinnitus keine Seltenheit sind. Die Ursachen für Tinnitus sind vielfältig und meistens in einer Fehlfunktion des zentralen Nervensystems begründet. In seltenen Fällen können sie aber auch auf organische Krankheiten zurückgeführt werden. Da es für Tinnitus zudem keine Standardtherapie gibt, sind viele Ärzte ratlos, wie sie die Ohrengeräusche ihrer Patienten therapieren sollen [KVL13].

1.1 Motivation

Die fehlende Möglichkeit, Tinnitus gezielt zu behandeln, ist ein großes, medizinisches und psychologisches Problem. So gibt es in der klinischen Praxis viele verschiedene Therapieansätze, auf die Patienten jeweils unterschiedlich gut ansprechen. Dementsprechend bedarf es oftmals mehreren Therapien, bevor einem Patienten geholfen werden kann. Diese lange Suche frustriert nicht nur viele Tinnitus-Patienten, sie verstärkt auch die psychischen Leiden, die betroffene Personen in dieser Zeit oftmals zusätzlich belasten. Um diese Situation zu verbessern, bedarf es somit einer Lösung, die jedem Patienten individuelle Therapievorschläge erstellen kann, die mit hoher Wahrscheinlichkeit erfolgreich sind.
Ein Mittel, mit dem dieses Ziel erreicht werden kann, ist das Smartphone. Denn mit einem solchen mobilen Gerät, das immer mehr Menschen besitzen, können nahezu an jedem Ort und zu jeder Zeit Daten eingegeben und abgefragt werden. Ereignisse, die mit dem Tinnitus in Zusammenhang stehen, können daher immer sofort in das Smartphone eingetragen werden. Gerade bei der Erstellung von Therapievorschlägen, wo jede zusätzliche Information von großer Bedeutung sein kann, spielt diese direkte Eingabemöglichkeit eine wichtige Rolle. Des Weiteren ist auch die Aktualität der Daten gewährleistet, sodass Therapievorschläge immer an das aktuelle Befinden des Patienten angepasst sind. Da außerdem alle tinnitusrelevanten Daten an einem zentralen Ort gespeichert werden, erhält sowohl der Patient als auch der behandelnde Arzt bzw. Therapeut eine optimale Übersicht über den Verlauf des Tinnitus.
Ziel dieser Arbeit ist es somit eine App zu entwickeln, die, basierend auf den eingegebenen Daten eines Tinnitus-Patienten, individuelle Therapievorschläge erstellt. In einer ersten Version der App, auf der diese Arbeit beruht, werden dem Patienten ausschließlich Therapien auf Basis wissenschaftlicher Erkenntnisse vorgeschlagen. Diese Therapievorschläge werden dabei durch eine Webanwendung verwaltet. Zu einem späteren Zeitpunkt ist zudem geplant, auch Bewertungen von Therapien durch App-Nutzer mit in die Berechnung einfließen zu lassen. Um Therapievorschläge zu erhalten, müssen Nutzer möglichst viele Informationen zu ihrem Tinnitus in die App eingeben.

Die App stellt daher Eingabemöglichkeiten für Ereignisse, Hörtests und Fragebögen zur Verfügung. Über eine Art mobile Tinnitusakte können Patienten diese Informationen dann wieder einsehen und so den Verlauf ihrer Erkrankung analysieren oder diesen mit einem behandelnden Arzt oder Therapeuten besprechen. Schließlich soll die App auch über das Symptom Tinnitus informieren. Dazu zeigt sie beispielsweise die wichtigsten Therapiearten an und bietet Zugriff zu einem Forum, in dem sich Patienten gegenseitig austauschen können.

1.2 Struktur der Arbeit

Diese Arbeit ist in sieben Kapitel eingeteilt. Nach diesem einleitenden Kapitel werden zunächst die medizinischen und psychologischen Grundlagen des Symptoms „Tinnitus“ behandelt. Im Anschluss daran folgt eine Anforderungsanalyse, in der die funktionalen und nichtfunktionalen Anforderungen an die App vorgestellt werden. Daran lehnt sich ein Abschnitt über die Benutzeroberfläche an. Anhand von Mockups wird gezeigt, wie die Interaktion des Nutzers mit der App zu Entwicklungsbeginn geplant war. Kapitel fünf beschreibt dann die Realisierung der App. Mit Hilfe eines Architekturbildes wird die Interaktion zwischen den einzelnen Komponenten Server, App und Datenbank veranschaulicht. Hierbei wird jede Komponente einzeln vorgestellt und analysiert. Unmittelbar darauf folgt ein bewertendes Kapitel. Darin wird überprüft, ob und inwieweit die zuvor definierten Anforderungen an die App erfüllt sind. Um eine einheitliche Bewertung zu gewährleisten, erhält jede Anforderung dazu eine Schulnote. Abgeschlossen wird diese Arbeit schließlich mit einer Zusammenfassung und einem Ausblick, wie die App weiter verbessert werden kann und wie diese Ziele in Zusammenhang mit der Partner-App „Track Your Tinnitus“ stehen.

2

Grundlagen

Im Folgenden werden nun die psychologischen und medizinischen Grundlagen von Tinnitus beschrieben. Diese basieren auf den wissenschaftlichen Arbeiten von Peter M. Kreuzer et al. und Wilhelm Frank et al. [KVL13] [FKS06]. Zu beachten gilt allerdings, dass hierbei nur ein Grundverständnis geschaffen werden soll, um den Inhalt der weiteren Kapitel besser einordnen zu können. Detailliertere Informationen zur Symptomatik Tinnitus finden sich zum Beispiel in [FKS06].

2.1 Formen von Tinnitus

Bei einem Tinnitus (lat. Klingeln) nehmen betroffene Personen ein ungeformtes Geräusch oder Pfeifen wahr, das in Wirklichkeit gar nicht existiert. Je nach Ursache wird zwischen subjektivem und objektivem Tinnitus unterschieden.

Auslöser für einen subjektiven Tinnitus ist beispielsweise ein Hörsturz, der in Folge eines Unfalls oder einer chronischen Überlastung ausgelöst werden kann. Die Stärke des subjektiven Tinnitus reicht von kaum wahrnehmbar bis unerträglich laut und kann innerhalb eines Tages variieren. Wo genau sich der Tinnitus befindet, ist von Patient zu Patient verschieden. Grob können drei verschiedene Bereiche angegeben werden:

- nur an einem Ohr
- an beiden Ohren
- im Inneren des Kopfes

Während der subjektive Tinnitus also meist psychischer Natur ist, liegt dem objektiven Tinnitus eine organische Krankheit zu Grunde. Der objektive Tinnitus ist zwar selten, lässt sich aber im Vergleich zum subjektiven Tinnitus deutlich besser behandeln, da das Geräusch von einem untersuchenden Arzt messbar ist. Dadurch kann die verursachende Krankheit bestimmt und gezielt dagegen therapiert werden. Nach erfolgreicher Behandlung verschwindet der Tinnitus in der Regel auch wieder.

2.2 Fragebögen

Abhängig davon, wie gut eine betroffene Person mit ihrem Tinnitus umgehen kann, lässt sich dieser in verschiedene Schweregrade unterteilen. Nimmt eine Person das Ohrgeräusch zwar wahr, kann jedoch gut damit leben, wird der Tinnitus als kompensiert bezeichnet. Hat das Ohrgeräusch hingegen starke, negative Auswirkungen auf das Privat- und Berufsleben einer Person, wird der Tinnitus als dekompensiert bezeichnet. Die klinische Einteilung der Schweregrade bei Tinnitus nach Biesinger et al. kennt vier solche Grade, die in Tabelle 2.1 aufgelistet sind [ECV+98].

Da jede Person ihren Tinnitus unterschiedlich stark wahrnimmt, bedarf es eines Messinstruments, mit dem der Tinnitus in einen dieser vier Schweregrade eingeteilt werden kann. Aufgrund ihrer einfachen Struktur eignen sich hierfür Fragebögen in besonderem Maße.

Schweregrad	Beschreibung
Grad 1	Der Tinnitus ist gut kompensiert. Es gibt keinen Leidensdruck.
Grad 2	Der Tinnitus tritt vorwiegend in der Stille auf und ist unter Stress oder unter körperlicher bzw. seelischer Belastung störend.
Grad 3	Der Tinnitus beeinträchtigt das private und berufliche Leben der betroffenen Person dauerhaft. Es treten geistige, seelische und körperliche Störungen auf.
Grad 4	Völlige Dekompensation des Privatlebens durch den Tinnitus. Betroffene Personen können ihren Beruf nicht mehr ausüben.

Tabelle 2.1: Schweregrade von Tinnitus nach Biesinger et al.

Die Fragen dieser Fragebögen beziehen sich dabei nicht nur exklusiv auf die subjektive Wahrnehmung des Tinnitus, sondern auch auf die Lebensumstände der betroffenen Personen. Tabelle 2.2 listet fünf solcher Tinnitusfragebögen auf.

Fragebogen	Ziel des Fragebogens
Tinnitus Schweregrad	Einordnung des Schweregrads
Mini Tinnitus Fragebogen (Mini TF)	Einfluss des Tinnitus auf Gefühle, Verhaltensweise und persönliche Einstellung
Tinnitus Sample Case History Questionnaire (TSCHQ)	Ermittlung der Lebensumstände des Patienten und dessen Umgang mit dem Tinnitus.
Tinnitus Handicap Inventory (THI)	Erkennung von Schwierigkeiten, die betroffene Personen aufgrund ihres Tinnitus haben.
Tinnitus Impairment Questionnaire (TBF)	Erkennung von Schwierigkeiten, die betroffene Personen aufgrund ihres Tinnitus haben

Tabelle 2.2: Tinnitusfragebögen und ihre jeweiligen Ziele

Die Bearbeitung von Fragebögen spielt in zwei Bereichen eine entscheidende Rolle. Zum einen dienen die Ergebnisse der Datengewinnung in Tinnitusstudien. Zum anderen sind Fragebögen eine wichtige Informationsquelle bei der Anamnese eines Patienten, um, wie gerade beschrieben, dessen Gesundheitszustand einordnen zu können.

2.3 Anamnese

Zu einer fundierten Tinnitus-Diagnose gehört die Anamnese des Patienten. Dazu zählt neben dem Ausfüllen von Fragebögen die Durchführung eines Hörtests, mit dem das Hörvermögen untersucht wird. Weiterhin ist wichtig festzustellen, wie lange der Patient den Tinnitus schon wahrnimmt. Dabei werden drei Phasen unterschieden:

- akut: < 3 Monate
- subakut: 3 - 12 Monate
- chronisch: > 12 Monate

Je nach Dauer des Tinnitus kommen unterschiedliche Behandlungsmaßnahmen in Frage, die im Abschnitt Therapieformen erläutert werden. Von Bedeutung ist außerdem, ob der Tinnitus synchron zum Puls wahrgenommen wird, da dies auf eine Gefäßerkrankung hindeuten könnte. Schließlich kann in einigen Fällen ein Tinnitus bereits durch eine Physiotherapie oder eine kieferorthopädische Maßnahme behoben werden. Aus diesem Grund ist es empfehlenswert herauszufinden, ob der Tinnitus durch bestimmte Bewegungen von Kiefer oder Halswirbelsäule moduliert werden kann.

Dieser Abschnitt hat sich aus Gründen der Übersicht auf die vier wichtigsten Punkte beschränkt. In der klinischen Praxis werden selbstverständlich noch viele weitere Aspekte untersucht, bevor die Diagnose Tinnitus gestellt und ein entsprechender Therapieplan erstellt werden kann.

2.4 Therapieformen

Dieser Abschnitt erläutert die wichtigsten Therapieformen für Tinnitus. Da es für Tinnitus keine Standardtherapie gibt, ist das oberste Ziel dieser Maßnahmen nicht den Tinnitus zu heilen, sondern Wege aufzuzeigen, die das Leben mit diesem Symptom vereinfachen.

1. **Counseling**

   Die Grundlage aller Therapieverfahren ist das so genannte Counseling, bei dem es sich um ein Beratungs- und Aufklärungsgespräch handelt.

   Da viele Patienten nur wenig über die Hintergründe von Tinnitus wissen, gehen sie von einer unheilbaren Krankheit aus. In vielen Fällen wird dieses Gefühl zusätzlich durch eine Vielzahl von erfolglos verlaufenden Therapiemaßnamen verstärkt. Infolgedessen entwickeln sich bei diesen Personen häufig psychische Erkrankungen wie Depressionen oder Panikattacken. Ziel des Counseling ist daher, betroffene Personen in einfühlsamer und verständlicher Weise über die Ursachen und Mechanismen von Tinnitus zu informieren.

2. **Hörtest**

   Um abzuklären, inwieweit der Tinnitus die Hörfähigkeit einschränkt, wird ein Hörtest durchgeführt. Ein Hörtest ist eine medizinische Untersuchung, bei der dem Patienten Töne in unterschiedlichen Frequenzen und Lautstärken vorgespielt werden. Dabei wird für jede Frequenz sukzessive die Lautstärke so lange erhöht, bis der Patient den Ton wahrgenommen hat. Dieser Vorgang wird für beide Ohren getrennt durchgeführt. Das Ergebnis wird schließlich in ein Liniendiagramm zur Visualisierung eingetragen. *Abbildung 2.1* zeigt ein solches Diagramm.

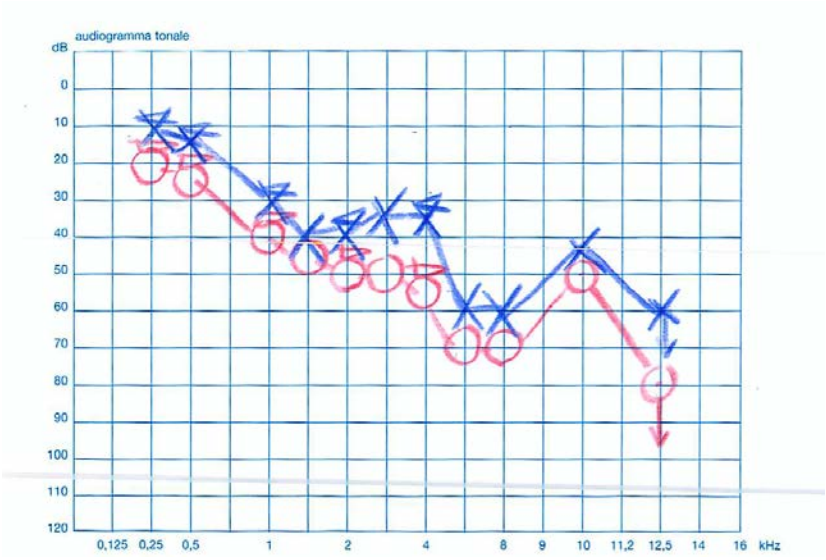


Abbildung 2.1: Diagramm zur Darstellung der Ergebnisse eines Hörtests

3. **Apparativ-Akustische Therapien**
   Als apparativ-akustisch werden all diejenigen Therapien bezeichnet, die auf Apparate wie zum Beispiel Hörgeräte, Tinnitusmasker oder Tinnitusnoiser setzen. Während Hörgeräte zu einer dauerhaften Verbesserung des Tinnitus beitragen können [SKH+10], sollen Tinnitusmasker und Tinnitusnoiser vor allem helfen, von ihm abzulenken oder ihn zu verdecken.
   Leiden betroffene Personen an einer Schwerhörigkeit, wird diesen zu Beginn einer Therapie ein Hörgerät eingesetzt. Durch das Hörgerät nehmen diese Personen wieder Umgebungsgeräusche wahr, die den Tinnitus in den Hintergrund drängen sollen. Patienten, die nur an einem geringen oder gar keinem Hörverlust leiden, kann möglicherweise mit einem sogenannten Tinnitusmasker geholfen werden. Dieser erzeugt mit Hilfe eines Rauschgenerators spezielle Umgebungsgeräusche, die den Tinnitus überdecken sollen. Ein ähnliches Instrument ist der Tinnitusnoiser, dessen Ziel die Ablenkung vom eigentlichen Ohrgeräusch ist. Dazu wird dem Patienten ein Rauschen vorgespielt, das leiser als der Tinnitus selbst ist. Mit der Zeit soll sich das Gehirn an dieses Rauschen gewöhnen und in der Konsequenz sowohl die Geräuschquelle als auch den Tinnitus als unwichtig einstufen. Tinnitusnoiser müssen keineswegs teuer sein. Bereits die Anschaffung eines Zimmerspringbrunnens oder einer CD mit Meeresrauschen kann hierfür ausreichend sein.

4. **Psychologie Therapieverfahren**
   Dieser Abschnitt stellt einige psychologische Therapieverfahren vor. In der Regel werden diese nicht einzeln angewandt, sondern sind jeweils Bestandteil bestimmter Verhaltenstherapien.
   Eine der Hauptursachen für einen subjektiven Tinnitus ist Stress. Dieser erhöht zum einen die Sensibilität gegenüber Geräuschen und bewirkt zum anderen Muskelverspannungen. Mit Hilfe von Entspannungsverfahren wird daher versucht, diesen Verspannungen entgegenzuwirken. Die Idee, die dahinter steckt, ist, dass sich eine entspannte Muskulatur positiv auf das Stresslevel auswirkt und als Konsequenz daraus die Empfindlichkeit gegenüber Geräuschen deutlich gesenkt wird.

Entspannungstechniken werden hauptsächlich in der subakuten Phase des Tinnitus eingesetzt. Zu den bekanntesten Techniken gehört neben Yoga und Autogenes Training die progressive Muskelrelaxation nach Jacobson.
In eine andere Richtung zielen kognitive Verfahren. Diese basieren auf der Annahme, dass die Art zu denken sich unmittelbar auf das Gefühlsleben und das Verhalten und somit auch auf die körperliche Wahrnehmung eines Menschen auswirkt. Patienten sollen daher mit Hilfe von kognitiven Therapien lernen, den Tinnitus neu zu bewerten und so ihre Angst bzw. Verunsicherung gegenüber dem Ohrgeräusch verlieren.
Eine weitere Möglichkeit, den Tinnitus einzudämmen, ist die Aufmerksamkeitsverlagerung. Der dahinterliegende, psychologische Gedanke dabei ist, dass es zu jedem Ohrgeräusch ein äquivalentes, externes Geräusch gibt. Indem die Aufmerksamkeit betroffener Personen gezielt auf dieses externe Geräusch gelenkt wird, sollen sich die betroffenen Personen daran gewöhnen, und so den Tinnitus verdrängen. Da eine Gewöhnung bei einem Tinnitus allerdings nicht möglich ist, zielt diese Therapie darauf ab, dass Patienten das Geräusch möglichst als Stille interpretieren und im Erfolgsfall das Geräusch nicht mehr oder nur noch schwach wahrnehmen.

5. **Tinnitus-Retraining-Therapie**
   Die Tinnitus-Retraining-Therapie, kurz TRT, ist eine ambulante Therapie, die sich aus Counseling und auditorischer Stimulation zusammensetzt (siehe Therapieformen 1 & 3). Die Tinnitus-Retraining-Therapie verfolgt zwei Ziele. Das erste Ziel besteht darin, den Tinnitus zu maskieren. Dazu wird meistens ein Tinnitusnoiser eingesetzt. Das zweite und wichtigere Ziel der TRT ist, dass sich Patienten an den Tinnitus gewöhnen und diesen nicht mehr als störend wahrnehmen. Das Besondere an einer Tinnitus-Retraining-Therapie ist die aktive Zusammenarbeit zwischen HNO-Arzt, Psychotherapeuten und Akustikern. Auf Basis der Diagnose des HNO-Arztes wird dabei zunächst eine Therapie erstellt. Diese wird im weiteren Verlauf von Psychotherapeuten und Akustikern betreut und setzt eine aktive Mitarbeit des Patienten voraus. Allerdings kann eine TRT nicht bei jedem Tinnitus-Patienten angewandt werden.

Voraussetzung zur Teilnahme an einer Tinnitus-Retraining-Therapie ist eine stabile, psychische Verfassung, die durch Counseling oder eine Psychotherapie erreicht werden kann.

6. **Pharmakologische Therapien**
   Mit Hilfe von Medikamenten wird versucht, die dem Tinnitus zugrunde liegende Erkrankung zu behandeln. Eine pharmakologische Therapie, deren oberstes Ziel es ist, die Entstehung eines chronischen Tinnitus zu verhindern, wird hauptsächlich in der akuten Phase angewandt. Dabei werden häufig Mittel zur Gefäßerweiterung verabreicht, sodass die Gefäße im Gehirn besser durchblutet werden und ein möglicher Sauerstoffmangel behoben wird. Auch Kortison wird in vielen Fällen verabreicht, um Schwellungen in den Gefäßen zu behandeln.

   Zusätzlich werden oftmals Medikamente eingesetzt, die die Fließeigenschaften des Bluts verbessern oder die Erregungsprozesse der afferenten Hörnervenfasern[1] beeinflussen sollen. Da in den meisten Fällen die zugrunde liegende Erkrankung allerdings nicht eindeutig bestimmt werden kann, ist eine rein medikamentöse Therapie nur in den seltensten Fällen erfolgreich.

7. **Chirurgische Verfahren**
   In seltenen Fällen gibt es außerdem noch die Möglichkeit, den Tinnitus durch einen chirurgischen Eingriff zu behandeln. Zu diesen Fällen gehören beispielsweise eine gequetschte Nervenbahn (neurovaskuläre Kompressionssyndrom), die Menière-Krankheit (Erkrankung des Innenohrs) oder die Existenz eines Tumors (Akustikusneurinom).

[1] von lat. affere = hintragen. Nervenfasern, die in das zentrale Nervensystem führen

2.5 Subtypen von Tinnitus

Ein großes Problem bei der Behandlung von Tinnitus ist, dass Patienten unterschiedlich gut auf einzelne Therapieformen ansprechen [Cos]. Dies deutet auf eine größere Zahl von Tinnitus-Subtypen hin, denen jeweils eine bestimmte Fehlfunktion eines physiologischen Prozesses zugrunde liegt. Gelingt es, diese Prozesse zu identifizieren, führt dies nicht nur zu einer Verbesserung einzelner Therapieformen, sondern ermöglicht auch eine genauere Vorhersage, ob eine bestimmte Therapie Erfolg hat. Zukünftig wäre es somit möglich, Tinnitus-Patienten mit einem spezifischen Subtyp mit einer bestimmten Therapieform gezielt zu behandeln.

2.6 Zuordnung der Daten zu Therapieformen

Das Ziel dieser Arbeit ist die Implementierung einer App, die auf Basis eingegebener Daten Therapievorschläge erstellt. Dies geschieht auf zwei unterschiedliche Arten.

Beim ersten Start der App werden die wichtigsten Informationen zur aktuellen Situation des Nutzers durch einen kleinen Fragebogen ermittelt. Mit diesen Fragen soll festgestellt werden, wie stark der Nutzer unter seinem Ohrgeräusch leidet und welche Maßnahmen er bereits getroffen hat. Die Fragen sind zudem so gestellt, dass der Nutzer nur mit „Ja“ oder „Nein“ oder der Eingabe einer Zahl antworten kann. Da dieser Fragebogen zum Zeitpunkt dieser Arbeit allerdings noch nicht existiert, zeigt *Abbildung 2.2* lediglich eine mögliche Auswahl der Fragen des Fragebogens. Nach Beantwortung der Fragen wird dem Nutzer empfohlen, alle noch nicht durchgeführten Maßnahmen in Angriff zu nehmen. Zusätzlich dazu werden dem Nutzer erste einfache Therapievorschläge unterbreitet. Dazu gehören beispielsweise Entspannungstechniken, wenn dieser angibt, gestresst zu sein oder das Tragen eines Hörgerätes bei Hörproblemen.

Sobald der Fragebogen einmal ausgefüllt ist, basieren gemachte Therapievorschläge auf komplexen Regeln, die von Experten erstellt werden. Jeder Regel ist dabei ein Therapievorschlag zugeordnet.

Abbildung 2.2: Fragebogen zur Bestimmung grundlegender Nutzerinformationen

Möchte ein Nutzer einen Therapievorschlag erhalten, werden die Daten nach Stichworten oder Schwellenwerten, die diese Regel enthalten, durchsucht. Anschließend werden dem Nutzer diejenigen Therapievorschläge angezeigt, deren Regeln erfolgreich auf die Daten angewandt werden konnten. Das folgende Beispiel, das Therapievorschläge mit Regeln aus Tabelle 2.3 erstellt, soll dieses Vorgehen nochmals verdeutlichen.

Name der Regel	**Suchbegriff/Schwellenwert**	**Behandlungsvorschlag**
R1 (Ereignis)	Stress/stressig	Erlernen von Entspannungstechniken
R2 (Hörverlust links)	40% - 60%	Tragen eines Hörgerätes am linken Ohr
R3 (Hörverlust rechts)	40% - 60%	Tragen eines Hörgerätes am rechten Ohr
R4 (Dauer des Tinnitus)	< 3 Monate	Medikamentöse Therapie
R5(TSCHQ)	Frage 8: Ja, im Rhythmus meines Herzschlages	Termin in Radiologie zum Ausschließen von Gefäßerkrankungen

Tabelle 2.3: Regeln, mit denen beispielhaft ein Therapievorschlag erstellt wird

Ein Nutzer mit chronischen Tinnitus trägt in die App die Ergebnisse eines Hörtests ein. Der Hörtest zeigt auf einer der unteren Frequenzen auf dem rechten Ohr eine Hörminderung von 48% gegenüber dem normalen Hörvermögen an. Zuvor hat der Nutzer bereits den Fragebogen TSCHQ bearbeitet und Frage 8 mit „Nein“ beantwortet. Die Auswertung der Regeln nach Eingabe dieser Daten führt zu folgendem Ergebnis, welches in Tabelle 2.4 zu sehen ist:

Name der Regel	**Trifft zu (Ja / Nein)**	**Grund**
R1	Nein	Suchbegriff wurde nicht gefunden
R2	Nein	Hörverlust auf dem rechten Ohr
R3	Ja	Hörverlust von 48% auf rechtem Ohr
R4	Nein	Nutzer leidet an chronischem Tinnitus
R5	Nein	Nutzer beantwortet Frage 8 des TSCHQ mit „Nein“

Tabelle 2.4: Auswertung der Regeln

Basierend auf diesem Ergebnis wird dem Nutzer somit nur das Tragen eines Hörgerätes am rechten Ohr empfohlen (Regel R3).

3

Anforderungsanalyse

In diesem Kapitel werden die funktionalen und nichtfunktionalen Anforderungen an die mobile Anwendung Tinnitus Navigator vorgestellt, die das Track Your Tinnitus Projekt von Jochen Hermann erweitert [Her]. Tinnitus Navigator besteht aus den beiden Komponenten App und Server. Aufgabe der App ist es, dem Nutzer durch Auswertung aller in die App eingetragenen Daten Therapievorschläge zu unterbreiten. Die Verwaltung des Empfehlungssystems sowie die Synchronisation der Daten erfolgt jeweils über den Server.

3.1 Funktionale Anforderungen

Der folgende Abschnitt stellt nun die wichtigsten Funktionen vor, die die App Tinnitus Navigator enthalten soll. Dabei wird zunächst immer die Anforderung genannt. Anschließend wird jeweils auf das zu Grunde liegende Problem eingegangen.

FA 1: Eingabe beliebiger, tagesaktueller Daten in wenigen Schritten

Wie aus Kapitel 1.1 hervorgeht, ist das oberste Ziel der App Tinnitus Navigator, Tinnitus-Patienten bei der Suche nach einer geeigneten Therapie zu unterstützen. Hierzu ist die Eingabe von persönlichen Daten eine notwendige Voraussetzung. Daher sollen Ereignisse, Dokumente oder Hörtestergebnisse des aktuellen Tages in wenigen Schritten in die App eingegeben werden können.

FA 2: Kalenderfunktion zur Verwaltung von Daten jeglicher Art mit beliebigem Datum

Zusätzlich zur Eingabemöglichkeit tagesaktueller Daten muss es dem Nutzer auch möglich sein, Daten beliebigen Datums in die App eingeben zu können. Wichtig ist hierbei ein intuitives Bedienkonzept, sodass der Nutzer schnell mit der App vertraut wird. Auch soll es dem Nutzer jederzeit möglich sein, seine Eingaben zu editieren oder zu löschen. Dies betrifft auch diejenigen Daten, die über FA1 eingetragen werden.

FA 3: Eingabe von zeitunabhängigen Informationen unterhalb des Kalenders

Bestimmte Daten wie Vorerkrankungen oder Medikamente ändern sich über einen längeren Zeitraum nur selten. Daher sollen diese Informationen separat vom Kalender aufgeführt sein.

FA 4: Eingabemöglichkeit mit anschließender Visualisierung für manuelle Hörtests

Über den Kalender soll es möglich sein, die Ergebnisse von Hörtests einerseits direkt hochzuladen, andererseits manuell einzugeben. In letzterem Fall soll der Nutzer die bei ihm gemessenen Werte nicht nur bei den standardmäßig getesteten Frequenzen, sondern auch bei patientenspezifischen Frequenzen eintragen können. Wie bei einem normalen Hörtest soll das Ergebnis nach Eingabe aller gemessenen Werte in Form eines Audiogramms darstellbar sein.

FA 5: Möglichkeit zum Ausfüllen von Fragebögen

Nur Daten in den Kalender einzutragen, reicht für einen fundierten Therapievorschlag nicht aus. Hierzu bedarf es zusätzlich des Ausfüllens von Fragebögen. Ist ein Fragebogen fertig beantwortet, soll dieser nicht mehr bearbeitet werden können. Es soll dem Nutzer aber möglich sein, den gleichen Fragebogen mehrfach hintereinander auszufüllen. In die Bewertung soll aber immer nur derjenige Fragebogen eingehen, der als letztes abgeschlossen wurde.

FA 6: Ständige Synchronisierung der Antworten von Fragebögen mit dem Server

Da Nutzer nicht immer einen kompletten Fragebogen auf einmal beantworten können, soll die Beantwortung unterbrochen und zu einem späteren Zeitpunkt fortgeführt werden können. Dies soll nicht nur in der App, sondern wie bei Track Your Tinnitus auch auf der Website umgesetzt sein.

FA 7: Synchronisierung aller eingegebenen Daten in der App mit dem Server

Neben den Antworten von Fragebögen sollen auch alle weiteren eingegebenen Daten mit dem Server synchronisiert werden. Dies ist beispielsweise dann wichtig, wenn ein Nutzer seinen Account auf einem zusätzlichen mobilen Gerät nutzen möchte oder sein altes mobiles Gerät defekt gegangen ist.

FA 8: Nutzen der App ohne Internetverbindung

Aufgrund von Funklöchern kann es immer wieder vorkommen, dass kein Internet zur Verfügung steht. In diesem Fall soll es trotzdem möglich sein, die App weiter zu verwenden. Entsprechende Synchronisierungen müssen nachgeholt werden, sobald wieder eine Internetverbindung aufgebaut werden kann.

FA 9: Statusanzeige über den aktuellen Stand der Nutzung

Um einen Überblick über die eingegebene Datenmenge zu erhalten, soll die App eine tabellarische Übersicht anbieten. Aus dieser soll hervorgehen, wie viele Ereignisse und Hörtests schon eingegeben, wie viele Dokumente hochgeladen und wie viele Fragebögen bereits beantwortet wurden.

Von weiterem Interesse könnte vor allem zu Nutzungsbeginn sein, welche Daten mindestens noch benötigt werden, um einen ersten Therapievorschlag zu erhalten. Auch die Möglichkeit, den Nutzer über Neuerungen zu informieren, könnte über diese Funktion realisiert werden.

FA 10: System zum Verwalten von Regeln und Therapievorschlägen auf der Website

Das wichtigste Ziel der App Tinnitus Navigator ist die Generierung von mindestens einem Therapievorschlag. Hierzu bedarf es Regeln, mit denen die eingegebenen Daten ausgewertet werden. Damit die Eingabe solcher Regeln nicht immer im Mobilgerät des Patienten vorgenommen werden muss, soll es eine Eingabemöglichkeit auf der Website geben, die es berechtigten Personen ermöglicht, Regeln und Therapievorschläge für Nutzer zu erstellen.

FA 11: Update-Funktion in der App für Therapievorschläge und Regeln

Da sich Regeln und Therapievorschläge immer wieder ändern können, soll außerdem stets die aktuellste Version der Regeln und Therapievorschläge in der App vorhanden sein.

FA 12: Übersicht über Therapievorschläge mit Zusatzinformationen

Schließlich soll die App eine Übersicht über die für den Nutzer jeweils passenden Therapievorschläge geben. Die Übersicht soll auf jeden Fall einen kurzen Titel und eine Beschreibung der Therapie enthalten. Auch eine kleine Begründung, warum diese Therapie vorgeschlagen wird, soll nicht fehlen. Zusätzlich dazu könnten Informationen zu einem Facharzt in der Nähe des jeweiligen Nutzers oder Erfahrungen von Personen mit ähnlicher Datenlage dem Nutzer angezeigt werden.

FA 13: Weitere Informationsquellen für Nutzer in der App und auf der Website

Um den Informationswert für den Nutzer zu steigern, soll die App eine Übersicht über die wichtigsten Therapiearten geben können. Auch ein Forum, wo sich Projektteilnehmer austauschen können, würde sich hierfür anbieten. Diese Informationsquellen sollen zudem auch auf der Website verfügbar sein.

3.2 Nichtfunktionale Anforderungen

Als Nächstes werden Anforderungen vorgestellt, die hauptsächlich für die Optik und die Bedienung der App sowie für den Datenschutz von Bedeutung sind. Wie bei den funktionalen Anforderungen wird jede nichtfunktionale Anforderung zunächst erwähnt, bevor dann die Problemstellung beschrieben wird.

NFA 1: Aufklärung des Nutzers über die Datenerhebung zu Projektbeginn

Gerade bei Projekten, in denen personenbezogene Daten gespeichert werden, spielt der Datenschutz eine wichtige Rolle. Deshalb soll der Nutzer zu Beginn des Projektes darauf aufmerksam gemacht werden, wo und wie seine Daten verwendet werden. Für den Fall, dass der Nutzer mit der Verarbeitung nicht einverstanden ist, kann er die App nicht nutzen. Stimmt er zu, soll ihm zu jeder Zeit das Recht eingeräumt werden, das Projekt abbrechen und alle Daten löschen zu können.

NFA 2: Intuitive Bedienung der App

Eine weitere wichtige Eigenschaft stellt die intuitive Bedienung der App dar. Bedienelemente in der App sollen daher an Apps angelehnt sein, die dem Nutzer bereits bekannt sind. Hierzu zählen beispielsweise Apps, die im System bereits vorinstalliert sind.

NFA 3: Eingabemöglichkeit auf verschiedene Art und Weisen.

NFA 2 impliziert zudem, dass wichtige Funktionen über mehrere Wege erreichbar sein sollen. So soll zum Beispiel die Eingabe von aktuellen Ereignissen nicht nur über den dafür vorgesehenen Bereich in der App, sondern auch über das Menü und den Kalender erfolgen können.

NFA 4: Anlehnung von Tinnitus Navigator an Track Your Tinnitus.

Tinnitus Navigator ist nach Track Your Tinnitus eine weitere App des Tinnitus Projekts. Da beide Apps die gleiche Zielgruppe ansprechen, sollen die grundlegenden Bedienelemente an ähnlichen Stellen zu finden sein. Da zu einem späteren Zeitpunkt beide Apps zu einer verschmolzen werden sollen, soll dies bei der Entwicklung von Tinnitus Navigator bereits bedacht werden. Bis dahin könnte eine Funktion, die den Wechsel zwischen den beiden Apps ermöglicht, hilfreich sein.

4

Benutzerschnittstellenentwurf

Nachdem im vorangehenden Kapitel die Anforderungen an die wichtigsten Funktionen der Tinnitus Navigator App erarbeitet wurden, zeigt dieses Kapitel anhand von Mockups einen ersten Benutzerschnittstellenentwurf ausgewählter Funktionen.

Bei Mockups handelt es sich um Prototypen der Benutzerschnittstelle, die außer einer Anordnung der Bedienelemente keine weitere Funktionalität aufweisen. Das spätere Design der App kann daher von diesem ersten Entwurf abweichen. Ziel dieses Entwicklungsabschnittes ist es, in einer frühen Phase der Entwicklung die Benutzerschnittstelle mit den Anforderungen zu verknüpfen. Dadurch können eventuelle Schwachstellen in der Anforderungsanalyse noch vor der Implementierung aufgedeckt und behoben werden. Da Mockups nicht programmiert, sondern mit einer Grafik-Software erstellt werden, sind Mockups für diese Aufgabe geradezu prädestiniert. Die Mockups in dieser Arbeit wurden mit der Software Balsamiq Mockups erstellt. [Biq]. Weitere Informationen zum Design komplexer mobiler Anwendungen finden sich in [GSP+14].

4.1 Startseite

Die Startseite in Abbildung 4.1 soll so aufgebaut sein, dass alle wichtigen Funktionen durch eine einzige Bildschirmberührung erreicht werden können. Über „Meine Tinnitusakte“ kann die Dateneingabe und Datenbearbeitung erreicht werden. Unter Status bekommt der Nutzer Informationen zu seinem aktuellen Fortschritt in der App. Und mit einer Touch-Geste auf „Therapie Vorschlag“ erhält der Nutzer, vorausgesetzt er hat eine ausreichende Zahl an Daten in die App eingegeben, einen oder mehrere Therapievorschläge.

Abbildung 4.1: Prototyp der Startseite

Zusätzlich zu diesen drei Funktionen sind auf der Startseite noch zwei weitere Elemente integriert, die dem Nutzer weiterführende Informationen bieten sollen. Dazu gehört zum einen die Kachel „Therapie Formen“, die eine Übersicht über aktuell angewandte Tinnitustherapien gibt und zum anderen ein „Forum“, in dem sich der Nutzer mit anderen Projektteilnehmern austauschen kann.

FA13 kann somit als erfüllt angesehen werden. Wie Abbildung 4.1 zeigt, ist die Startseite in Kacheloptik gehalten. Der Vorteil der Kacheloptik besteht unter anderem darin, dass Kacheln auch noch auf kleinen Bildschirmen gut bedient werden können. Die vergrößerte Bedienfläche erlaubt es außerdem, neben einer Beschriftung auch ein mnemonisches Icon anzuzeigen, welches die Wiedererkennungsrate der zugrunde liegenden Funktion erhöht. Damit auch Nutzer mit Sehproblemen die Kacheln gut unterscheiden können, sind diese jeweils mit einem pastellfarbigen Hintergrund versehen.

4.2 Meine Tinnitusakte

Tippt der Nutzer auf die Kachel „Meine Tinnitusakte“, gelangt er von der Startseite in seine persönliche Tinnitusakte. Wie auf Abbildung 4.2 zu sehen ist, ist auch diese in einer Kacheloptik gehalten.

Abbildung 4.2: Prototyp der Seite *Meine Tinnitusakte*

In der persönlichen Tinnitusakte finden sich alle Funktionen, die aktiv genutzt werden müssen, um mindestens einen Therapievorschlag zu erhalten. So kann ein Kalender aufgerufen werden, in dem Ereignisse eingegeben, editiert oder gelöscht werden können. Des Weiteren können Untersuchungsergebnisse in Form von Dokumenten oder Bildern über „Dokumente hochladen“ auf dem Server gespeichert werden. Zuletzt hat der Nutzer die Möglichkeit, „Fragebögen auszufüllen“. Die persönliche Tinnitusakte deckt somit die funktionalen Anforderungen FA1, FA3 und FA5 ab.

4.3 Kalender

Der oben erwähnte Kalender soll dem Nutzer unabhängig vom aktuellen Datum die Möglichkeit bieten, sowohl Daten neu eingeben als auch bestehende Daten editieren und löschen zu können. Wie Abbildung 4.3 links zeigt, sind Tage, an denen bereits Daten eingetragen sind, zur Kenntlichmachung farblich markiert. Durch längeres Drücken auf einen solchen Tag öffnet sich eine Übersicht, die alle Ereignisse und Dokumente anzeigt, die für diesen Tag hinterlegt sind.

Abbildung 4.3: Prototypen des Kalenders, der Ereignisübersicht und der Eingabemaske für Ereignisse

Im Gegensatz dazu kann mit einer kurzen Touch-Geste auf einen Tag ein neues Ereignis erstellt werden. Die Ereignisübersicht zeigt Abbildung 4.3 Mitte, die Eingabemöglichkeit eines neuen Ereignisses ist in Abbildung 4.3 rechts zu sehen. Mit einer Touch-Geste auf „OK“ werden die Änderungen in einer lokalen Datenbank und, Internetverbindung vorausgesetzt, auf dem Server gespeichert. Somit sind die funktionalen Anforderungen FA2, FA7 und FA8 eingehalten

4.4 Hörtest

Neben Arztterminen und Therapiebeginne könnte ein neues Ereignis zum Beispiel auch ein Hörtest sein. Damit die Ergebnisse von Hörtests mit in den Therapievorschlag einfließen können, sollen Hörtestdokumente nicht nur einfach auf den Server hochgeladen, sondern auch in die App eingetragen werden können. Zur Eingabe steht wie in Abbildung 4.4 links zu sehen ist für jedes Ohr ein eigener Button zur Verfügung, über den das Eingabefenster, welches Abbildung 4.4 rechts zeigt, erreicht wird.

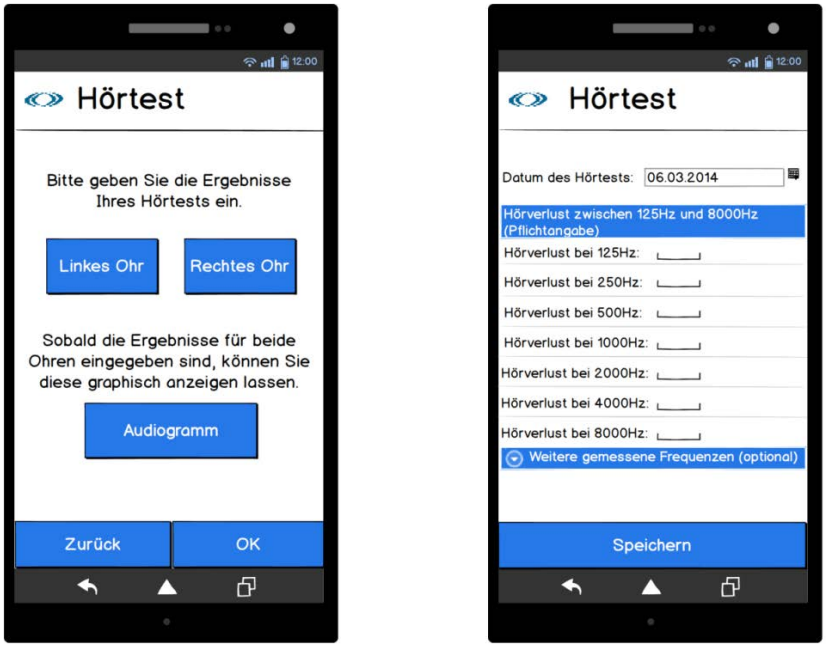


Abbildung 4.4: Prototyp der Eingabemaske für Hörtestergebnisse

Neben der Dateneingabe soll hierin außerdem auch das Datum, an dem der Hörtest durchgeführt wurde, gewählt werden können. Nachdem die Messwerte für beide Ohren eingetragen sind, soll es möglich sein, diese in Form eines Liniendiagramms zu betrachten. Abbildung 4.5 zeigt ein solches Liniendiagramm, bei dem durch zwei Auswahlboxen ausgewählt werden kann, ob nur die Werte des linken Ohrs oder nur die Werte des rechten Ohrs oder die Werte beider Ohren angezeigt werden sollen.

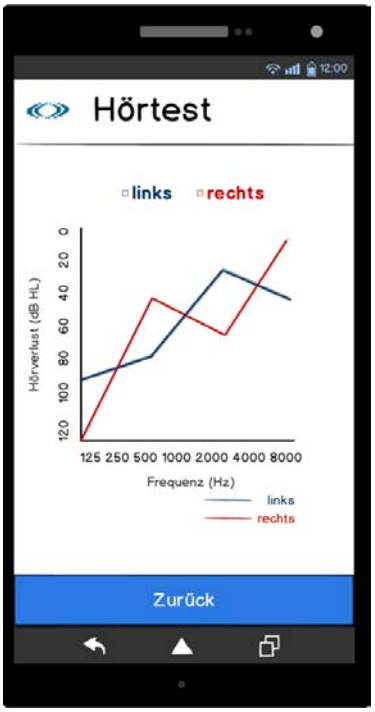


Abbildung 4.5: Prototyp des Liniendiagramms zur Visualisierung von Hörtestergebnissen

4.5 Menü

Um die wichtigsten Funktionen von jeder Seite der App aufrufen zu können, empfiehlt sich die Implementierung eines Menüs. Mit der Android Support Library v4 wurde dazu der Navigation Drawer eingeführt, mit dem ein Menü erstellt werden kann, das durch Ziehen von der linken Seite in die Mitte des Bildschirms aktiviert wird.

Dieses auch als Sliding-Menü bezeichnete Menü ist sehr platzsparend, da es sich bei Bedarf einfach über den aktuellen Bildschirm legt, ohne diesen zu verändern. Außerdem ist das Sliding-Menü inzwischen in vielen bekannten Apps integriert, sodass damit nur unerfahrene Nutzer anfangs Probleme haben sollten. Wie das Mockup in Abbildung 4.6 zeigt, ist das Sliding-Menü in vier Abschnitte unterteilt. Abschnitt 1 enthält alle Funktionen, die über den Startbildschirm erreichbar sind, Abschnitt 2 alle Funktionen der persönlichen Tinnitusakte. Die letzten beiden Abschnitte beinhalten alle weiteren Funktionen, die die App enthalten soll. Da diese Funktionen vergleichsweise selten aufgerufen werden, sollen diese platzsparend nur über das Sliding-Menü erreichbar sein. Um die einzelnen Einträge des Sliding-Menüs noch besser zu kennzeichnen, könnten die bereits im Kachelmenü verwendeten Icons vor dem Text platziert werden

Abbildung 4.6: Prototyp des Sliding-Menüs

5

Architektur

Im folgenden Kapitel wird die Architektur der Tinnitus Navigator Anwendung vorgestellt. Die Anwendung besteht aus den beiden Komponenten Server und App und teilt dieses Kapitel daher in zwei Abschnitte auf. Abschnitt 5.1 beschreibt zunächst die Komponente Server, die nach dem Model-View-Controller-Pattern aufgebaut ist. In Abschnitt 5.2 wird dann die Komponente App vorgestellt. Darin werden die wichtigsten Strukturen und Funktionen präsentiert, die für die Erstellung eines Therapievorschlags benötigt werden. Jeder Abschnitt ist in zwei Untereinheiten gegliedert. Nach einer kurzen Vorstellung der Komponente wird ihre Architektur jeweils im Detail beschrieben.

5.1 Architektur des Servers

Dieser Abschnitt befasst sich mit der Architektur des Servers. Ein Teil der Webanwendung wurde bereits im Rahmen des Track Your Tinnitus Projektes erstellt [Her]. Die Beschreibung der Architektur bezieht sich daher nur auf diejenigen Teile, die für Tinnitus Navigator zusätzlich erstellt wurden. Eine Ausnahme bildet die Beschreibung der Fragebögentabellen, da das Beantworten von Fragebögen für die Bestimmung von Therapievorschlägen wichtig ist.

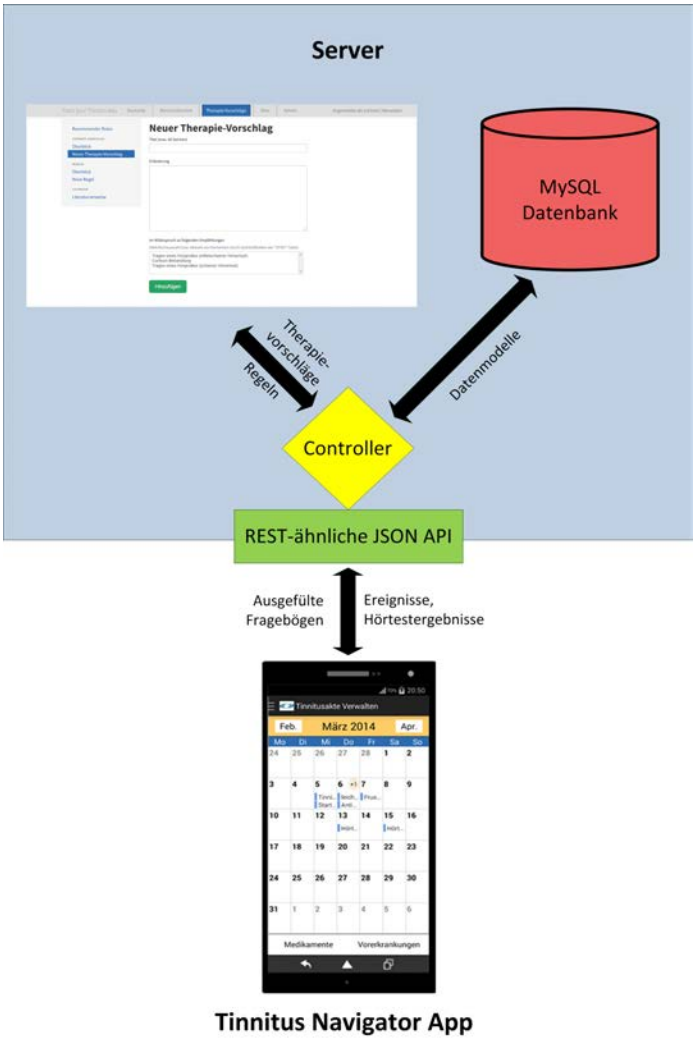


Abbildung 5.1: Überblick über die Gesamtarchitektur

5.1.1 Übersicht

Die Eingabe von Regeln und Therapievorschlägen erfolgt über eine Webanwendung, die auf einem Apache Web-Server läuft. Das Schaubild in Abbildung 5.1 zeigt im oberen Bereich diesen Server. Für die Entwicklung der Webanwendung und der API wurde das Open-Source-PHP-Web-Application-Framework Laravel in Version 3 eingesetzt [Larc]. Dieses unterstützt das Model-View-Controller-Pattern. Das bedeutet, dass die Präsentation, das Datenmodell und die Steuerung der Webanwendung jeweils als eigenständige Komponenten agieren. Im Folgenden werden nun das Datenmodell und der Controller der Webanwendung anhand von Codebeispielen erläutert. Abgeschlossen wird dieses Kapitel durch die Präsentation der Webseiten, die Formulare zur Erstellung von Therapievorschlägen und Regeln bereitstellen.

5.1.2 Datenbanktabellen

Sämtliche Daten, die durch Eingaben in der App oder auf dem Server anfallen, werden in einer MySQL-Datenbank zentral auf dem Server gespeichert. Diese ist in Abbildung 5.1 oben rechts eingezeichnet. Anhand von Entity-Relationship-Diagrammen werden nun diejenigen Tabellen vorgestellt, deren zugehörige Datenmodelle für die Erstellung eines Therapievorschlags von Bedeutung sind.

Eine bedeutende Informationsquelle in Tinnitus Navigator stellt der Kalender dar, in dem der Nutzer verschiedene Ereignisse eintragen kann. Die Ereignisse werden nach dem Upload in der Tabelle *calendarevents* gespeichert, die in Abbildung 5.2 dargestellt ist. Jedes Ereignis besteht aus einem Titel und einer Beschreibung. Während der Titel für jedes Ereignis angegeben werden muss, ist die Beschreibung optional. Da jedes Ereignis zu einer bestimmten Zeit stattfindet, enthält die Tabelle außerdem eine Spalte *_time*, in der das Datum in Millisekunden gespeichert wird. Zusätzlich dazu werden über die Standardzeitstempel *created_at* und *updated_at* die Zeitpunkte gespeichert, wann der Datensatz erzeugt und wann der Datensatz zuletzt aktualisiert wurde. Weiterhin kann der Nutzer im Kalender der App verschiedene Arten von Ereignissen eintragen.

Dazu gehören zum Beispiel eine Tinnitus-Attacke, der Beginn einer Therapie oder ein Arztbesuch. Zusätzlich gibt es noch Ereignistypen, die durch die Verwendung anderer Funktionen in der App entstehen. Als Beispiel sei hier das Ausfüllen von Fragebögen oder das Eingeben eines Hörtests aufgeführt. Um diese Ereignistypen unterscheiden zu können, besitzt jeder Ereignistyp eine bestimmte ID, die in der Spalte *type* gespeichert wird. Derzeit gibt es zehn verschiedene Ereignistypen, deren Zahl in späteren Versionen ohne großen Aufwand erweitert werden kann. Schließlich muss jedes Ereignis einem bestimmten Nutzer zugeordnet werden können. Hierzu enthält die Tabelle *calendarevents* eine Referenz auf die *user_id* des Nutzers, die gleichzeitig der Primärschlüssel der Tabelle *users* ist (Abbildung 5.2). Eine zusätzliche Referenz auf die ID des Nutzers dient dabei dem Erhalt der Information, wenn sich der Nutzer löschen sollte.

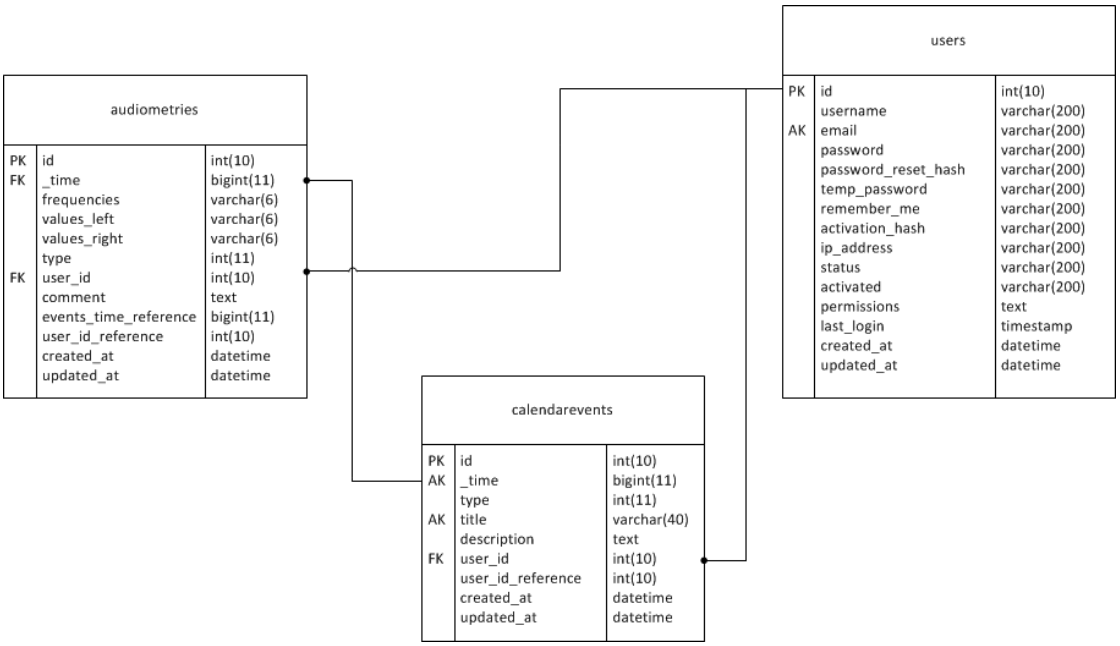


Abbildung 5.2: ER-Diagramm der Tabellen *audiometries*, *calendarevents* und *users*

Abbildung 5.2 zeigt außerdem die Tabelle *audiometries*, welche die Ergebnisse von Hörtests speichert. Jedes Tupel der Tabelle beschreibt das Messergebnis eines bestimmten Hörtests für beide Ohren bezüglich einer bestimmten Frequenz. Über die Spalte *_time*, in der das Datum des jeweiligen Hörtests als Fremdschlüssel gespeichert ist, können alle Datensätze, die zu einem Hörtest gehören, bestimmt werden.

Jede Frequenz lässt sich zudem in eine von drei Kategorien einordnen. Neben den Standardfrequenzen, die in jedem Hörtest geprüft werden, gibt es Frequenzen, die bei einem Hörverlust in hohen Frequenzbereichen getestet werden. Schließlich gibt es noch Frequenzen, die je nach Patient individuell geprüft werden. In welche dieser drei Kategorien die gemessene Frequenz fällt, wird in der Spalte *type* gespeichert. Weiterhin müssen die Hörtestergebnisse eindeutig einem Nutzer zugeordnet werden können. Dies geschieht wiederum mit *user_id* bzw. *user_id_reference*.

Neben Ereignissen und Hörtestergebnissen stellen die Ergebnisse von Fragebögen eine weitere, wichtige Datenquelle dar. Die Tabellen, in denen die Fragebögen, Fragen und Antworten gespeichert sind, wurden bereits für die App Track Your Tinnitus entwickelt. Aufgrund der Bedeutung der Fragebögen für die Bestimmung eines Therapievorschlags, werden diese Tabellen, die in Abbildung 5.3 zu sehen sind, in dieser Arbeit nochmals vorgestellt.

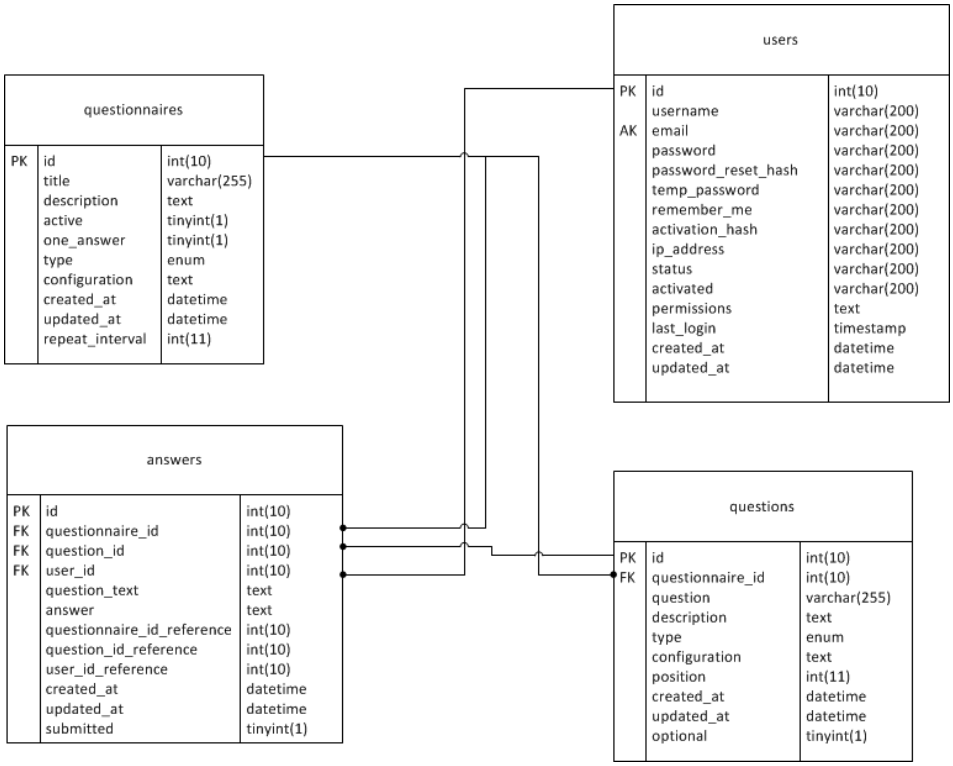


Abbildung 5.3: ER-Diagramm der Tabellen *questionnaires*, *questions*, *answers* und *users*

Die Tabelle *questionnaires* besteht aus einer eindeutigen ID, einem Titel und einer Beschreibung des Fragebogens. Da einige Fragebögen ausschließlich Fragen eines bestimmten Antworttyps enthalten, gibt es ein *one_answer*-Flag. Ist dieses gesetzt, können in den Spalten *type* und *configuration* Antworttyp und Antwortkonfiguration direkt ausgelesen werden. Andernfalls erfolgt die Spezifizierung der Antwortmöglichkeiten über die namensgleichen Spalten in der Tabelle *questions*, in der alle Fragen gespeichert werden. Jede Frage besteht aus einer ID, der eigentlichen Frage und einer Beschreibung der Frage. Außerdem hält jede Frage eine Referenz auf den zugehörigen Fragebogen. Die Position der Frage im Fragebogen nimmt die Spalte *position* auf. Antworten zu den Fragen werden in der Tabelle *answers* gespeichert. Jedes Antworttupel besteht aus der Frage *question_text* und der gegebenen Antwort *answer*. Um die Antwort eindeutig einem Fragebogen, einer Frage und einem Nutzer zuordnen zu können, werden zusätzlich deren IDs als Referenz gespeichert.

Damit schließlich aus den in die App eingegebenen Daten ein Therapievorschlag erstellt werden kann, bedarf es gewisser Regeln. Diese Regeln sind in der Tabelle *ruledefs* gespeichert, die Abbildung 5.4 zeigt. Neben einem Namen besitzt jede Regel eine ID, die sie eindeutig identifiziert. Mit Hilfe der Regeln wird letztendlich ein SQL Statement aufgebaut, das die in der Regel angegebene Spalte einer Tabelle abfragt. Der abzufragende Wert oder Begriff ist in der Spalte *value* gespeichert und wird durch die Bedingung *filter* eingegrenzt. Regeln können außerdem mit anderen Regeln verknüpft werden, wenn diese den gleichen Therapievorschlag enthalten. Dadurch ist es möglich, den Detailgrad der Abfrage steuern. Über *connection_type* wird angegeben, wie die Verknüpfung erfolgen muss. Aktuell stehen dazu die vier boolesche Operatoren „AND“, „OR“, „NAND“ und „NOR“ zur Verfügung. Die Spalte *connected_with* bezeichnet die ID der zu verknüpfenden Regel. Schließlich enthält jede Regel eine Referenz auf die ID des ihr zugeordneten Therapievorschlags. Die Therapievorschläge in der Tabelle *recommendations* bestehen jeweils aus einem Titel und einer optionalen Beschreibung. Um einen Vorschlag eindeutig bestimmen zu können, enthält dieser außerdem eine ID. Da sich Therapievorschläge gegenseitig widersprechen können, sind in der Spalte *contradicted_to* jeweils IDs von Vorschlägen enthalten, die sich nicht mit dem Vorschlag *recommendation_id* vertragen.

Zuletzt zeigt Abbildung 5.4 unten die Tabelle *mappings*, die die internen Namen von Tabellen und Spalten in benutzerfreundlichere Begriffe übersetzt. Dadurch kann der Nutzer im Formular zur Erstellung von Regeln auf einfache Weise die Tabellen und Spalten erkennen, auf die er seine neue Regel anwenden möchte.

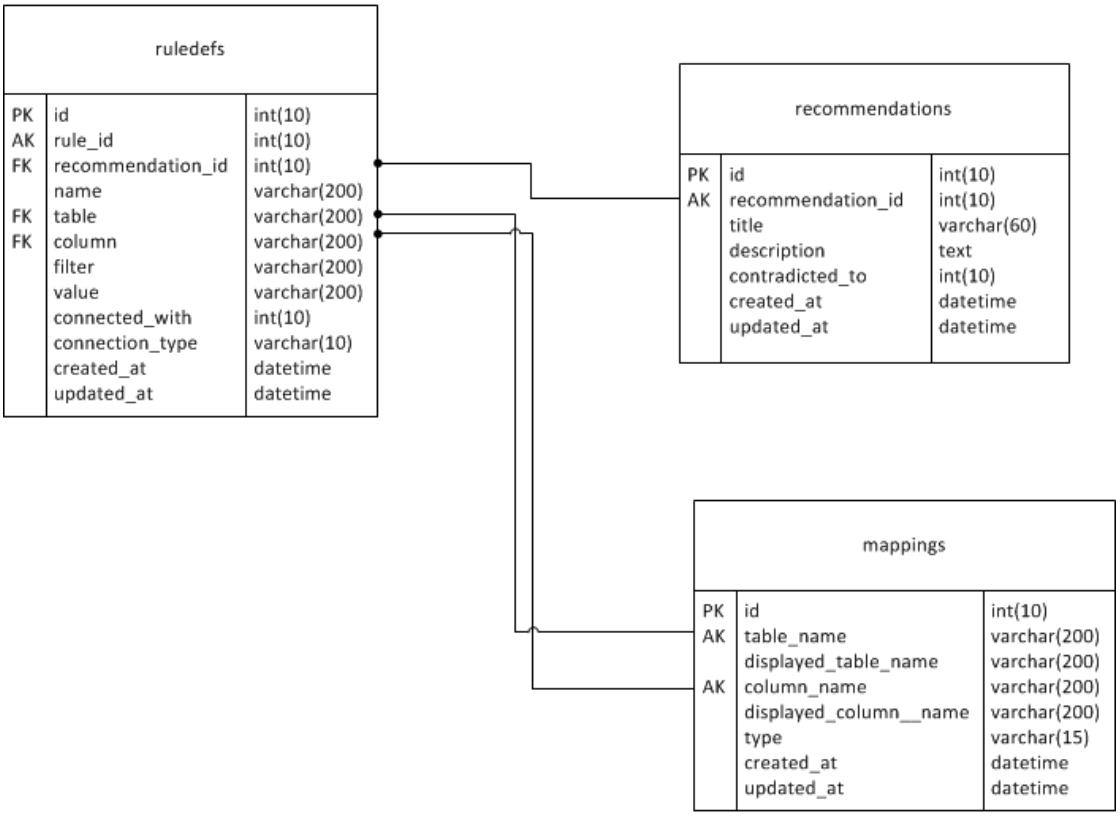


Abbildung 5.4: ER-Diagramm der Tabellen *ruledefs*, *recommendations* und *mappings*

Alle Tabellen, die in diesem Abschnitt vorgestellt wurden, finden sich noch einmal in Abbildung 5.5 als ER-Diagramm. Dieses Diagramm ist so strukturiert, dass Tabellen, die miteinander in Beziehung stehen, nah beieinander liegen. Der besseren Übersicht wegen sind jedoch nur Primärschlüssel, Alternativschlüssel und Fremdschlüssel eingetragen.

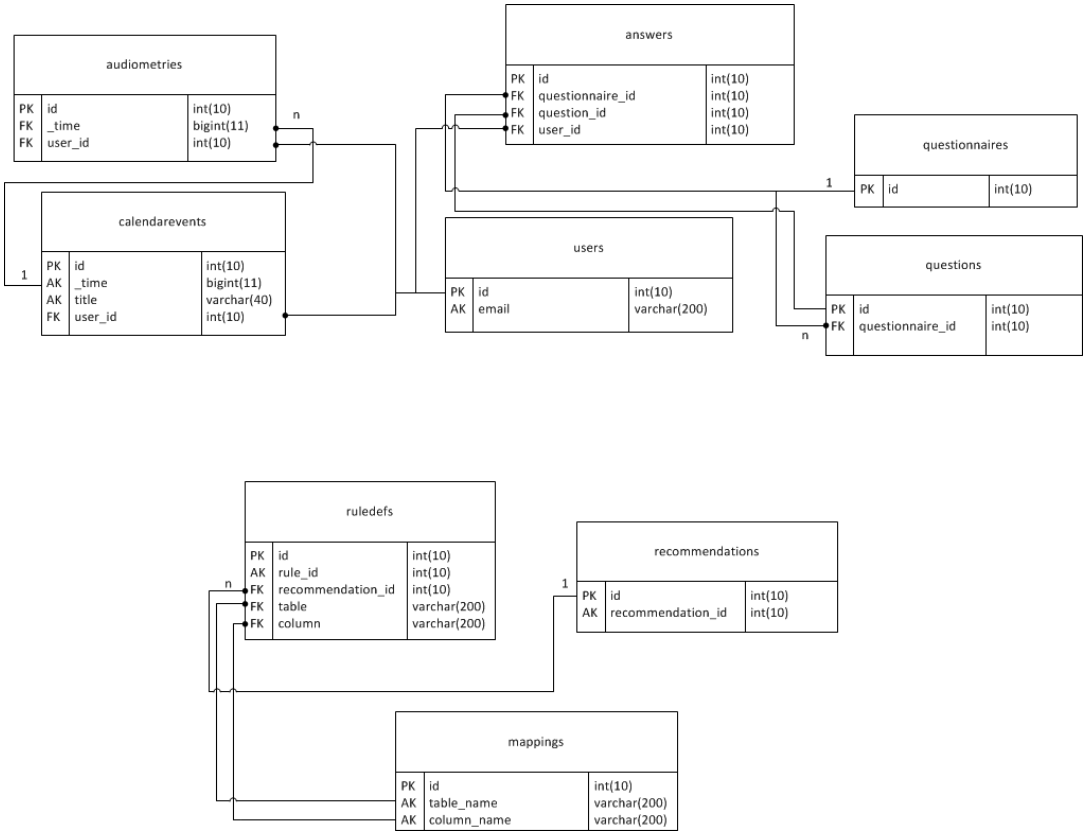


Abbildung 5.5: Gesamtübersicht über alle Tabellen, die für Tinnitus Navigator von Bedeutung sind

5.1.3 Beziehungen zwischen Tabellen

Wie im ER-Diagramm in Abbildung 5.5 im letzten Abschnitt angedeutet ist, befinden sich alle Datenbanktabellen, die für Tinnitus Navigator zusätzlich erstellt wurden, in einer One-To-Many-Beziehung. So können zum Beispiel einem Ereignis aus der Tabelle *calendarevents* mehrere Hörtestwerte aus der Tabelle *audiometries* zugeordnet werden. Allerdings gehört ein Hörtestwert immer nur zu einem bestimmten Ereignis. Wie diese Beziehung in Laravel definiert wird, zeigt Listing 5.1 in den Zeilen 8 - 10. Darin wird der Modellklasse *Calendarevent* die Methode *audiometries* hinzugefügt, welche wiederum die Methode *has_many* aufruft. Andersherum wird in Modellklasse *Audiometry* die Methode *event* implementiert, die die Methode *belongs_to* aufruft (Zeile 21 – 23).

Neben der One-To-Many-Beziehung unterstützt Laravel noch zwei weitere Beziehungstypen:

1. One-To-One
2. Many-To-Many

Bei einer One-To-One-Beziehung ist jedem Datensatz einer Tabelle A ein anderer Datensatz einer Tabelle B zugeordnet. Beispielsweise besitzt ein Patient genau eine Patienten-ID. Diese ID kann wiederum genau einem Patienten zugeordnet werden. Eine Many-To-Many-Beziehung kann zum Beispiel zwischen einer Tabelle Patient und einer Tabelle Medikamente bestehen. Denn ein Patient muss ein oder mehrere Medikamente einnehmen. Gleichzeitig kann jedes Medikament aber auch einem oder mehreren Patienten verschrieben werden. Bei einer Many-To-Many-Beziehung ist immer eine dritte Tabelle notwendig, in der die Fremdschlüssel der miteinander verknüpften Tabellen gespeichert sind.

Eine Anleitung, wie die Beziehungen One-To-One und Many-To-Many in Laravel implementiert werden, findet sich zum Beispiel unter [Larb].

Listing 5.1: Beziehung zwischen den Modellen *CalendarEvent* und *Audiometry*

```
1  <?php
2     class CalendarEvent extends Eloquent {
3
4        public function user() {
5           return $this->belongs_to('User');
6        }
7
8        public function audiometries(){
9           return $this->has_many('Audiometry');
10       }
11    }
12 ?>
13
```

```
14 <?php
15 class Audiometry extends Eloquent {
16
17     public function user() {
18         return $this->belongs_to('User');
19     }
20
21     public function event() {
22         return $this->belongs_to('CalendarEvent');
23     }
24 }
25 ?>
```

5.1.4 Datenmodelle

Laravel bietet für die Implementierung des Modells das sogenannte Eloquent-ORM an. Die Abkürzung ORM steht für Object-Relational-Mapper und beschreibt ein Konzept zum Abbilden von Objekten auf relationale Datenbanken. Jede Modellklasse erbt dazu von der Klasse *Eloquent* und ist mit einer Datenbanktabelle verknüpft, in der die Modelldaten gespeichert werden. Damit in Laravel das Mapping erfolgreich durchgeführt werden kann, müssen zwei Voraussetzungen erfüllt sein:

1. Jede Tabelle muss einen Primärschlüssel namens id enthalten
2. Der Tabellenname ist die Pluralform des Namens des korrespondierenden Datenmodells

Alternativ können allerdings auch der Tabellenname und der Name des Primärschlüssels explizit in der Modellklasse angegeben werden. Listing 5.2 zeigt das Datenmodell *RuleDef*, welches zur Speicherung von Regeln verwendet wird. Die zugehörige Tabelle, die beide Voraussetzungen erfüllt, befindet sich im vorangehenden Abschnitt in Abbildung 5.4.

Listing 5.2: Das Datenmodell *RuleDef*

```
1  <?php
2  class RuleDef extends Eloquent {
3
4      public function user() {
5          return $this->belongs_to('User');
6      }
7
8      public function recommendation(){
9          return $this->belongs_to('Recommendation');
10     }
11 }
12 ?>
```

5.1.5 Erstellen und Bearbeiten von Datenmodellen

Zum Erstellen und Bearbeiten von Datenmodellen zeigt Listing 5.3 einen Ausschnitt der Controller-Klasse *Api_Apicalendarevents_Controller*, die die aus der App hochgeladenen Kalendereereignisse verarbeitet und in der Datenbank calendarevents speichert.

Eine neue Instanz des Ereignismodells *$event* wird durch den Befehl *new* in Zeile 7 erstellt. Anschließend können die Felder von *$event* direkt bearbeitet werden (Zeile 8 – 14). Es sei darauf hingewiesen, dass die Bezeichnung jedes Feldes identisch zu der Bezeichnung ihrer entsprechenden Spalte in der Datenbank ist.
Soll ein bereits bestehender Datensatz bearbeitet werden, wird das *CalendarEvent*-Objekt in den Zeilen 2 bis 5 zunächst aus der Datenbank geladen. Jedes *CalendarEvent*-Objekt lässt sich dabei eindeutig über einen Nutzer, eine Uhrzeit und einen Typ bestimmen. Danach kann das Objekt ganz normal bearbeitet werden. Sowohl nach dem Erstellen als auch nach dem Bearbeiten eines Datenmodells müssen die Änderungen in der Datenbank gespeichert werden. Dies geschieht durch Aufruf der Methode *save* in Zeile 14.

Listing 5.3: Ausschnitt der Controller-Klasse *Api_Apicalendarevents_Controller*

```
<?php
    $event = CalendarEvent::where('user_id', '=', $user['id'])
                -> where('_time', '=', $data -> _time)
                -> where('type', '=', $data -> type)
                -> first();

    if ($event == null) $event = new CalendarEvent();
    $event -> user_id = $user['id'];
    $event -> _time = $data -> _time;
    $event -> type= $data -> type;
    $event -> title = $data -> title;
    $event -> description = $data -> description;
    $event -> user_id_reference = $user['id'];
    $event -> save();
?>
```

5.1.6 Abrufen von Datenmodellen aus der Datenbank

Modelle können auf drei Arten aus der Datenbank geladen werden. Eine Möglichkeit besteht in der Verwendung der Methode *find* (Listing 5.4, Zeile 4). Diese nutzt den ihr als Parameter übergebenen Primärschlüssel, um den gesuchten Datensatz in der Datenbank zu finden.
Eine andere Möglichkeit ist das Zusammenstellen eines SQL-Statements mit dem QueryBuilder. Dabei wird durch den Aufruf vordefinierter Datenbankzugriffsmethoden wie *insert* oder *join* eine SQL-Query definiert, der Prepared Statements zu Grunde liegen. Der WHERE-Teil einer solchen Query wird durch den Aufruf der *where*-Methode spezifiziert. Durch Verknüpfen mehrerer *where*-Methoden können, wie in den Zeilen 19 - 21 zu sehen ist, auch mehrere Attribute auf einmal gefiltert werden. Statements, die mit dem QueryBuilder erstellt werden, müssen immer durch einen Aufruf von *first* oder *get* abgeschlossen werden.

Während mit *first* in Zeile 28 immer nur das erste gefundene Ergebnistupel zurückgeliefert wird, gibt *get* prinzipiell alle Datensätze zu einer Query aus (Zeile 21). Schließlich können mit der Methode *all* alle Datensätze einer Tabelle geladen werden. Wie Zeile 31 zeigt, ist hierfür kein QueryBuilder notwendig.

Listing 5.4: Abrufen von Datenmodellen

```
1  <?php
2     // findet das Regel-Objekt mit id=3 und
3     // gibt den Namen der Regel aus
4     $rule = RuleDef::find(3);
5     echo $rule->name;
6
7     // baut eine Query mit dem Laravel
8     // QueryBuilder auf, die alle Regeln
9     // zurückliefert, die auf die Tabelle
10    // audiometries angewandt werden
11    $audiometry_rules =
12       RuleDef::where('table', '=', 'audiometries')
13                ->get();
14    // baut eine Query auf, die alle Regeln
15    // zurückliefert, die auf die Tabelle
16    // 'audiometries' und die Spalte 'frequencies'
17    // angewandt werden
18    $audiometry_rules1 =
19       RuleDef::where('table', '=', 'audiometries')
20               -> where('column', '=', 'frequencies')
21               -> get();
22
23    // gibt nur den ersten gefundenen Datensatz
24    // zu obiger Query zuröck
25    $audiometry_rules2 =
26    RuleDef::where('table', '=', 'audiometries')
```

```
27              -> where('column', '=', 'frequencies')
28              -> first();
29
30      // gib alle Regeln der Tabelle 'ruledefs' aus
31      $rules = RuleDef::all();
32  ?>
```

5.1.7 Controller

Im Model-View-Controller-Pattern dient der Controller der Steuerung der Anwendung. In Tinnitus Navigator empfängt der Controller daher sämtliche Daten, die in der Webanwendung oder der App eingegeben wurden und reicht diese an die Modell-Komponente weiter. Des Weiteren synchronisiert der Controller auch den Datenbestand mit der App, wenn diese eine entsprechende Anfrage schickt. Abbildung 5.6 veranschaulicht dies noch einmal.

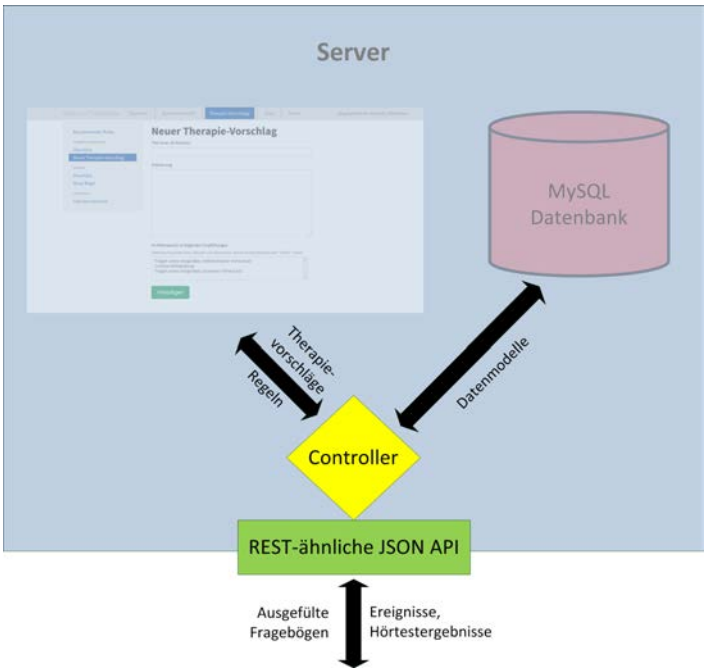


Abbildung 5.6: Der Controller im Model-View-Controller-Pattern

In Laravel werden Controller-Klassen unter *application/controllers* gespeichert und erben von der Klasse *Base_Controller*. Diese Klasse ist wiederum eine Subklasse von *Controller* und stellt die wichtigsten Methoden den Controllerklassen zur Verfügung [Lara].

Listing 5.5 zeigt den Anfang der Controllerklasse *recommender*. In Laravel können Methoden in Controllerklassen verschiedenen HTTP-Requests zugeordnet werden. Diese Eigenschaft wird als RESTful bezeichnet und muss zu Anfang einer Controllerklasse durch Setzen der Variable *$restful* auf true aktiviert werden (Zeile 4). Anschließend muss dem Namen einer Methode eines der vier Schlüsselwörter „post_“, „get_“, „delete_“ oder „put_“ vorangestellt werden, sodass die Methode nur beim entsprechenden HTTP-Request ausgeführt wird. Die Seite *recommender.index* in Zeile 7 wird somit nur geladen, wenn diese durch ein entsprechendes GET-Request angefragt wird.

Listing 5.5: Beginn der Controllerklasse *recommender* deren Methoden RESTful sind

```
1 <?php
2 class Recommender_Controller extends Base_Controller {
3
4    public $restful = true;
5
6    public function get_index() {
7        return View::make('recommender.index');
8    }
```

In Tinnitus Navigator sind alle Controllerklassen der Webanwendung RESTful. Tabelle 5.1 zeigt dies am Beispiel der Klasse *recommender*.

Methode	**Beschreibung**
get_index	gibt eine View zurück, die die Startseite der Therapievorschlagsverwaltung darstellt
get_recommendationall	gibt eine View zurück, die alle eingegebenen Therapievorschläge anzeigt
post_recommendationall	löscht den vom Nutzer selektierten Therapievorschlag

get_recommendationnew	gibt eine View zurück, in der ein neuer Therapievorschlag erstellt werden kann
post_recommendationnew	speichert den neu erstellten Therapievorschlag in der Tabelle *recommendations*
get_recommendationedit	gibt eine View zurück, in der ein bestehender Therapievorschlag bearbeitet werden kann
post_recommendationedit	speichert den bearbeiteten Therapievorschlag in der Tabelle *recommendations*
get_rulesall	gibt eine View zurück, die alle Regeln anzeigt
post_rulesall	löscht die vom Nutzer selektierte Regel
get_rulesnew	gibt eine View zurück, in der eine neue Regel erstellt werden kann
post_rulesnew	speichert die neu erstellte Regel in der Tabelle *ruledefs*
get_rulesedit	gibt eine View zurück, in der eine Regel bearbeitet werden kann
post_ruleseidt	speichert die bearbeitete Regel in der Tabelle *ruledefs*
get_literature	gibt eine View zurück, in der weiterführende Literatur aufgelistet ist
post_selectcolumns	liefert benutzerfreundliche Spaltennamen zu einer ausgewählten Tabelle zurück
post_selectfilters	liefert benutzerfreundliche Namen für Operatoren in einem WHERE-Statement
post_selectrules	liefert alle Regeln mit gleichem Therapievorschlag zurück

Tabelle 5.1: Übersicht über alle Methoden der Controller-Klasse *recommender*

Wie aus Tabelle 5.1 ersichtlich wird, werden die Seiten der Webanwendung durch Aufruf von GET-Methoden geladen. POST-Methoden erfüllen hingegen zwei Funktionen in der Webanwendung.

Zum einen verarbeiten sie die Daten, die auf den Seiten der Webanwendung eingegeben werden. Zum anderen liefern sie bei einer AJAX HTTP POST-Anfrage Daten aus der Datenbank an die Webseite zurück. Aufgrund der ähnlichen Abläufe innerhalb der GET- und POST-Methoden wird im Folgenden anhand der Methoden *get_recommendationedit* und *post_rulesnew* jeweils eine GET-Methode und eine POST-Methode näher erläutert. Auf die unteren drei Methoden in Tabelle 5.1, die AJAX-Anfragen verarbeiten, wird im nächsten Abschnitt View eingegangen

Die Methode *get_recommendationedit* in Listing 5.6 wird aufgerufen, wenn ein Nutzer einen bereits existierenden Therapievorschlag bearbeiten möchte. Die ID dieses Therapievorschlags wird *get_recommendationedit* durch den Parameter *$rid* übergeben (Zeile 1). Zunächst wird der Therapievorschlag in den Zeilen 2 - 4 anhand der ID in der Tabelle *recommendations* gesucht. Anschließend wird in Zeile 8 die View zurückgegeben, in der der Nutzer den Therapievorschlag bearbeiten kann. Damit die Daten des Therapievorschlags direkt in die entsprechenden Komponenten der View eingetragen werden können, wird das Therapievorschlag-Objekt der View über die Methode *with(Key, Value)* zur Verfügung gestellt (Zeile 9). Zusätzlich dazu wird der Parameter *$rid* übertragen. Ein PHP-Abschnitt in der View ermittelt damit alle Therapievorschläge, die dem Therapievorschlag in *$edit_recommendation* widersprechen.

Listing 5.6: Die Methode *get_recommendationedit*

```
1  public function get_recommendationedit($rid) {
2      $edit_recommendation =
3          Recommendation::where('recommendation_id', '='$rid)
4                          ->first();
5      if( count($edit_recommendation)==0 )
6      return View::make('recommender.recommendation_all');
7      else
8      return View::make('recommender.recommendation_edit')
9              -> with('edit_recommendation', $edit_recommendation)
10             -> with('rid', $rid);
11 }
```

Um eine neu angelegte Regel zu speichern, wird die Methode *post_rulesnew* aufgerufen. Diese überprüft zunächst, ob alle notwendigen Werte eingegeben sind und gibt im Fehlerfall mit der *with*-Methode eine Fehlermeldung zurück, die dem Nutzer mitteilt, warum die Regel nicht gespeichert werden konnte. Aus Gründen der Übersicht zeigt Listing 5.7 in den Zeilen 2 bis 4 nur die Überprüfung, ob ein Titel angegeben wurde. Wenn alle Pflichtwerte vorhanden sind und die Regel noch nicht existiert, wird diese neu angelegt. Da Regeln in Tinnitus Navigator mit anderen Regeln verknüpft werden können, wird dazu in Zeile 23 für jede Verknüpfung eine neue Instanz eines *RuleDef*-Datenmodells erstellt und in Zeile 34 in der Tabelle *ruledefs* gespeichert.

Listing 5.7: Die Methode *post_rulesnew* - Teil 1

```
1  public function post_rulesnew() {
2      if( !(Input::has('rule_title_new')) )
3      return View::make('recommender.rules_new')
4                  -> with('error', 'Kein Titel');
5      // An dieser Stelle folgen weitere Abfragen, die der
6      // besseren übersicht wegen entfernt wurden
7
8      $rule = RuleDef::where('name', '=',
9                        trim(Input::get('rule_title_new')))
10                       ->first();
11     if($rule==null){
12        $rule_id = DB::table('ruledefs')->max('id') +1;
13        if(Input::get('connection')=='connection_yes'){
14
15            $connection_types = Input::get('connection_types');
16            $connection_rule_ids =
17                              Input::get('connection_rule_ids');
18            $types = json_decode($connection_types,true);
19            $rules = json_decode($connection_rule_ids,true);
20
21            if( count($types)==count($rules) ){
```

```
22                for($i = 0; $i < count($types); $i++){
23                  $ruledef = new RuleDef();
24                  $ruledef->name = Input::get('rule_title_new');
25                  $ruledef->rule_id = $rule_id;
26                  $ruledef->recommendation_id =
27                            Input::get('recommendation_select');
28                  $ruledef->table = Input::get('rule_scope');
29                  $ruledef->column = Input::get('column_table');
30                  $ruledef->filter = Input::get('column_filter');
31                  $ruledef->value = Input::get('rule_item');
32                  $ruledef->connected_with =  $rules[$i];
33                  $ruledef->connection_type = $types[$i];
34                  $ruledef-> save();
35                }
36              }
37          }
```

Handelt es sich jedoch um eine eigenständige Regel, so wird das entsprechende Datenmodell in den Zeilen 1 – 12 von Listing 5.8 erstellt. Das Datenobjekt einer eigenständigen Regel unterscheidet sich dabei vom Datenobjekt einer verknüpften Regel nur durch die nicht benötigten Felder *connected_with* und *connection_type*. Da diese Felder optional sind, müssen sie nicht angegeben werden und erhalten automatisch den Wert NULL.

Listing 5.8: Die Methode *post_rulesnew* - Teil 2

```
1               $ruledef = new RuleDef();
2               $ruledef->name = Input::get('rule_title_new');
3               $ruledef->rule_id = $rule_id;
4               $ruledef->recommendation_id =
5                         Input::get('recommendation_select');
6               $ruledef->table = Input::get('rule_scope');
7               $ruledef->column = Input::get('column_table');
8               $ruledef->filter = Input::get('column_filter');
```

```
9             $ruledef->value = Input::get('rule_item');
10            $ruledef-> save();
11        }
12    }
13    else return View::make('recommender.rules_new')
14                -> with('error', 'Regel existiert schon');
15  return View::make('recommender.rules_new')
16              -> with('success', __('recommender.success')); }
```

Schließlich sollen noch zwei besondere Eigenschaften in der Implementierung erwähnt sein. Alle Datensätze, die zu einer Regel gehören, besitzen die gleiche *rule_id*. Diese ID wird bei jedem Aufruf von *post_rulesnew* nur einmal erstellt und ist daher für alle *RuleDef*-Datenobjekte gleich. Damit sichergestellt ist, dass die *rule_id* für jede Regel verschieden ist, entspricht diese stets dem um eins erhöhten Primärschlüssel (Listing 5.7, Zeile 12).

Um außerdem alle verknüpften Regeln und alle Verknüpfungsoperatoren auf einmal an den Controller übertragen zu können, werden diese beim Aufruf der POST-Methode in der View als JSON-Zeichenkette kodiert. Auf Controllerseite werden diese Daten zur weiteren Verarbeitung anschließend wieder mit der Methode *json_decode* in ein assoziatives Array dekodiert (Listing 5.7, Zeile 18 & 19).

5.1.8 View

Im Model-View-Controller-Pattern ist die Hauptaufgabe der View, Daten zu präsentieren, die ihr über einen Controller zur Verfügung gestellt werden. Eine weitere Aufgabe der View ist außerdem, Benutzeraktionen, wie zum Beispiel die Eingabe von Daten auf einer Webseite, an den Controller weiterzuleiten. Abbildung 5.7 visualisiert diese beiden Vorgänge. In der Webapplikation von Tinnitus Navigator zeigt die View-Komponente Webseiten an, über die Therapievorschläge und Regeln eingegeben und bearbeitet werden können. Dieser Abschnitt stellt diese Webseiten nun vor.

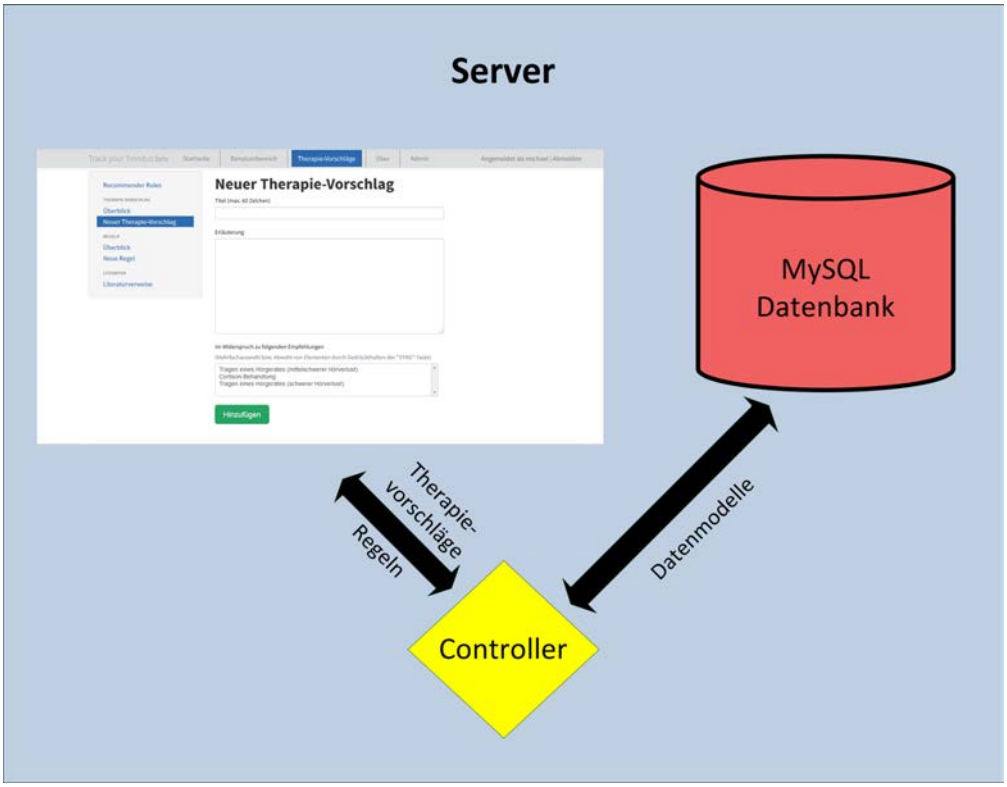


Abbildung 5.7: Zusammenhang zwischen View und Controller

Damit Nutzern in Tinnitus Navigator Therapievorschläge angezeigt werden können, müssen diese zuvor an einem zentralen Punkt eingegeben werden. Dazu wurde auf der Webseite des Tinnitus-Projekts eine zusätzliche Rubrik *Therapie-Vorschläge* eingerichtet, die in Abbildung 5.8 zu sehen ist. Damit es zu keinen falschen Therapieempfehlungen kommen kann, können Therapievorschläge und Regeln allerdings nur von Personen wie Ärzte oder Therapeuten verwaltet werden, die Mitglieder der Benutzergruppe *Recommender* sind. Entsprechend sind die Optionen zur Bearbeitung von Therapievorschlägen oder Regeln bei Standardnutzern ausgeblendet. Für diese Nutzer aufrufbar ist hingegen die Webseite Literaturhinweise, die weitere Informationen und Literatur zum Thema Tinnitus zur Verfügung stellt. Im Folgenden werden nun die Eingabemasken für Therapievorschläge und Regeln vorgestellt, die in der rot umrahmten Übersicht in der linken Spalte aufgerufen werden können.

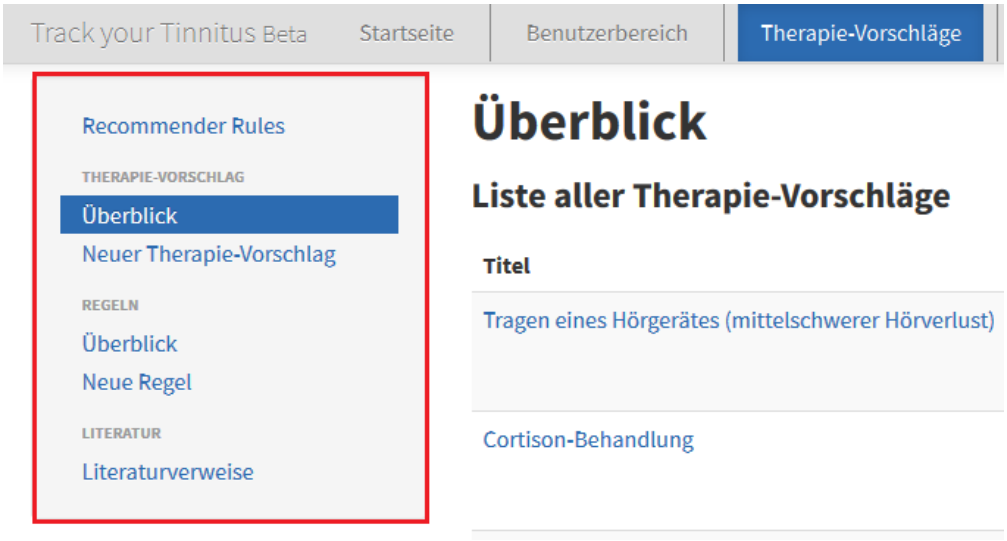


Abbildung 5.8: Übersicht über die Rubrik Therapie-Vorschläge

Da für die Erstellung von Regeln mindestens ein Therapievorschlag vorhanden sein muss, werden zunächst die einzelnen Komponenten der Seite *Neuer Therapie-Vorschlag* beschrieben. Diese Seite ist in Abbildung 5.9 dargestellt. Der Titel des Therapievorschlags wird über das Textfeld eingetragen (1). Dieser soll so gewählt sein, dass eine normale Person den Inhalt der vorgeschlagenen Therapie erahnen kann. Über das Eingabefeld soll dann der Therapievorschlag beschrieben werden (2). Dazu gehört zunächst eine Begründung, warum diese Therapie empfehlenswert ist. Des Weiteren ist es empfehlenswert, die Therapie ausführlich zu erklären und über mögliche Risiken zu informieren. Da der Titel und die Erläuterung später in der App angezeigt werden, wenn ein Nutzer einen Therapievorschlag erhalten möchte, handelt es sich bei diesen beiden Eingabefeldern um Pflichtfelder. Weiterhin können über ein Auswahlmenü Therapievorschläge selektiert werden, die zu diesem Vorschlag im Widerspruch stehen (3). So kann zum Beispiel verhindert werden, dass einem Nutzer in der akuten Phase sofort eine umfangreiche Therapiemaßnahme vorgeschlagen wird, obwohl zu diesem Zeitpunkt möglicherweise eine medikamentöse Behandlung schon ausreichend sein könnte. Über den Button *Hinzufügen* wird der neue Therapievorschlag abschließend in der Datenbank gespeichert.

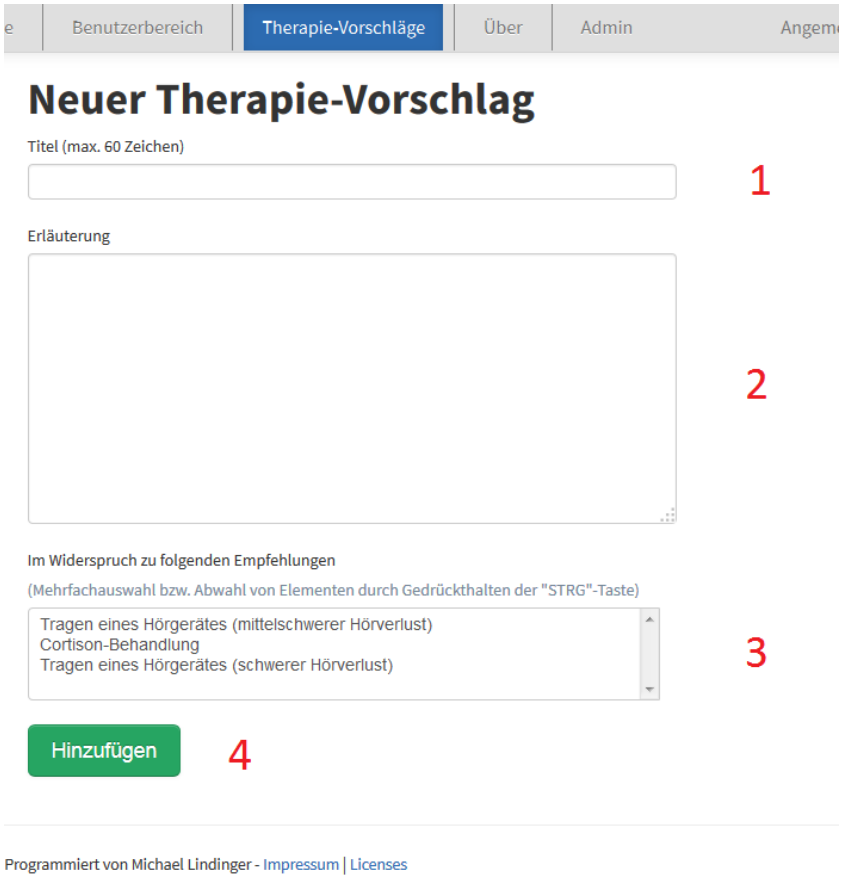


Abbildung 5.9: Eingabeformular zur Erstellung neuer Therapievorschläge

Sobald mindestens ein Therapievorschlag erstellt ist, können Regeln erzeugt werden. Dies geschieht über das Formular in Abbildung 5.10. Mit Regeln wird in Tinnitus Navigator definiert, unter welchen Bedingungen eine Therapie vorgeschlagen werden kann. Wie ein Therapievorschlag benötigt eine Regel einen Namen (1). Da Regeln miteinander verknüpft werden können, ist es empfehlenswert, den Namen der Regel so zu wählen, dass eindeutig daraus hervorgeht, was durch die Regel festgelegt wird. Über den *Bereich* wird anschließend ausgewählt, auf welche Tabelle die Regel angewandt werden soll (2). Der Tabellenname ist dabei in benutzerfreundlicher Sprache dargestellt. Abhängig von der Wahl des Bereichs zeigt *Kategorie* dann alle Spalten der selektierten Tabelle in benutzerfreundlicher Sprache an, die durchsucht werden können (3).

Aktuell können im Formular folgende Bereiche und Kategorien ausgewählt werden:

- Antworten
  - Antwort
- Ereignisse
  - Titel
  - Beschreibung
- Hörtests
  - Frequenz
  - Wert linkes Ohr
  - Wert rechtes Ohr

Um die Suche nach dem eingegebenen Begriff weiter einzuschränken zu können, zeigt das Auswahlmenü rechts neben *Kategorie* außerdem unterschiedliche Suchkriterien an. Diese richten sich nach dem Datentyp der selektierten Spalte. So kann zum Beispiel bei textuellen Daten angegeben werden, ob der Suchbegriff exakt in der Ergebnismenge enthalten oder nur ein Teil dieser sein muss. In ähnlicher Weise lassen sich numerische Daten anhand von bestimmten Schwellenwerten aussortieren.

Anschließend muss der Therapievorschlag ausgewählt werden, der dem Nutzer angezeigt werden soll, wenn diese Regel zutrifft (4). Der Therapievorschlag spielt außerdem bei der Verknüpfung von Regeln eine Rolle. Denn um Widersprüche bei der Kombination mehrerer Regeln zu vermeiden, können aktuell nur solche Regeln miteinander verknüpft werden, die die gleiche Therapie empfehlen. Soll eine Regel mit einer anderen Regel verknüpft werden, muss diese Funktion zunächst über einen Radiobutton aktiviert werden (5). Anschließend zeigt sich eine Tabelle, über die der Typ der Verknüpfung und die zu verknüpfende Regel ausgewählt werden können (6). Wie auf Abbildung 5.10 zu sehen ist, können außerdem weitere Regeln hinzugefügt werden. Um die Regel schließlich zu speichern muss der Button *Hinzufügen* gedrückt werden (7).

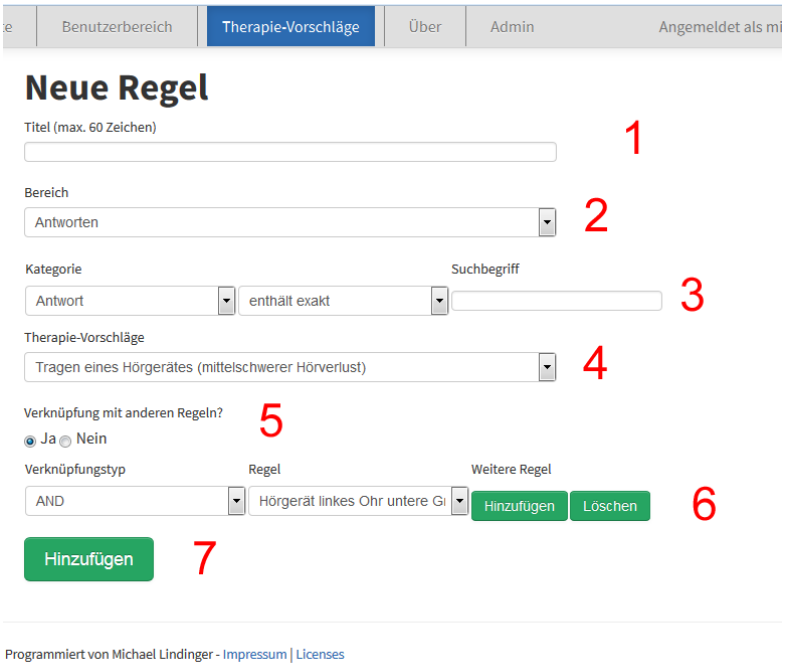


Abbildung 5.10: Eingabeformular zur Erstellung neuer Regeln

Am Ende des vorangehenden Kapitels Controller wurden in einer Tabelle drei Methoden gezeigt, die durch einen AJAX HTTP POST-Request in der View aufgerufen werden. Tabelle 5.2 zeigt diese Methoden nochmals:

Methode	**Beschreibung**
post_selectcolumns	liefert benutzerfreundliche Spaltennamen zu einer ausgewählten Tabelle zurück
post_selectfilters	liefert benutzerfreundliche Namen für Operatoren in einem WHERE-Statement
post_selectrules	liefert alle Regeln mit gleichem Therapievorschlag zurück

Tabelle 5.2: Methoden, die über einen AJAX HTTP POST-Request ausgeführt werden

Die Verwendung von AJAX ist notwendig, da wie eben erwähnt bei der Erstellung von Regeln gewisse Entscheidungen getroffen werden müssen, die sich auf den Inhalt der anderen Auswahllisten auswirken.

Im Folgenden wird nun mit Abbildung 5.11 erklärt, wie diese Methoden nacheinander ausgeführt werden. Nachdem alle durchsuchbaren Tabellen geladen sind, wird über eine JQuery-Funktion, die in die Webseite integriert ist, ein AJAX HTTP POST-Request an den Controller gesendet, um die Methode *post_selectcolumns* aufzurufen. Diese Methode gibt ein Array von durchsuchbaren Spalten an die View zurück, die in der selektierten Tabelle enthalten sind. Der Abschluss dieses Vorgangs triggert anschließend eine weitere JQuery-Funktion, die über einen AJAX HTTP POST-Request die Methode *post_selectfilters* aufruft. Diese liefert abhängig vom Datentyp der angezeigten Spalte ein Array von Suchkriterien zurück. So können Suchergebnisse beispielsweise mit mathematischen Operatoren eingegrenzt werden, wenn die zu durchsuchende Spalte auf einem numerischen Datentyp basiert. Die Methode *post_selectrules* wird schließlich nur ausgeführt, wenn die Regel mit anderen Regeln verknüpft werden soll. Diese Methode liefert ein Array von Regeln zurück, die den gleichen Therapievorschlag enthalten wie die neu angelegte Regel.

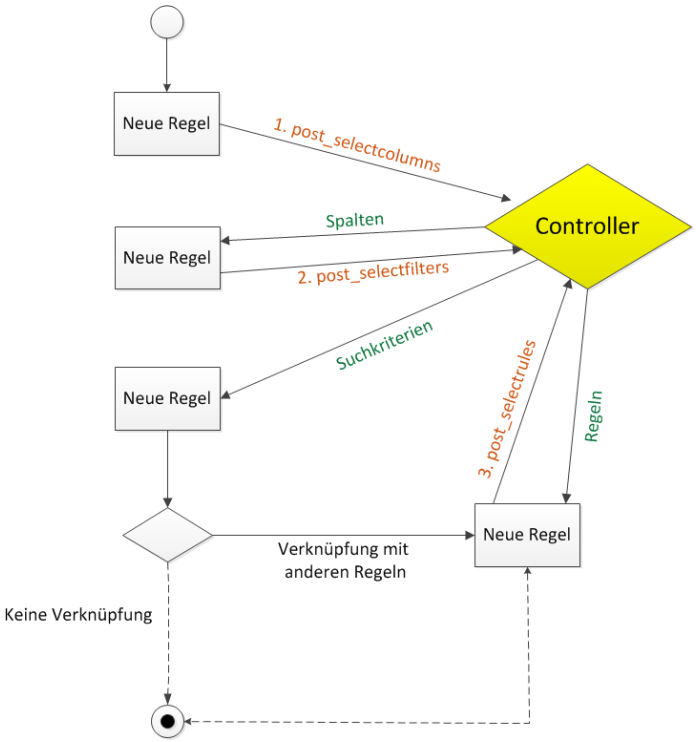


Abbildung 5.11: Ablauf beim Laden der Seite *Neue Regel*

Da sich Therapien immer wieder ändern können, müssen eingegebene Therapievorschläge und Regeln bearbeitet oder gelöscht werden können. Diese Funktion kann über die Übersichtsseiten der Therapievorschläge bzw. Regeln aufgerufen werden. Abbildung 5.12 zeigt beispielhaft die Übersichtsseite der Therapievorschläge. Diese listet alle bereits eingegebenen Therapievorschläge auf. Für den Fall, dass sich eine Therapie als wirkungslos ergibt, können Therapievorschläge über den Button *Löschen* aus entfernt werden. Andernfalls kann durch einen Klick auf den Titel ein Therapievorschlag auch bearbeitet werden. Da die Webseite zur Bearbeitung eines Therapievorschlags der Webseite entspricht, über die ein neuer Therapievorschlag eingerichtet werden kann, wird an dieser Stelle auf eine Abbildung verzichtet. Der einzige Unterschied besteht darin, dass die Komponenten der Webseite nicht leer, sondern mit den Daten des Therapievorschlags bereits vorausgefüllt sind.

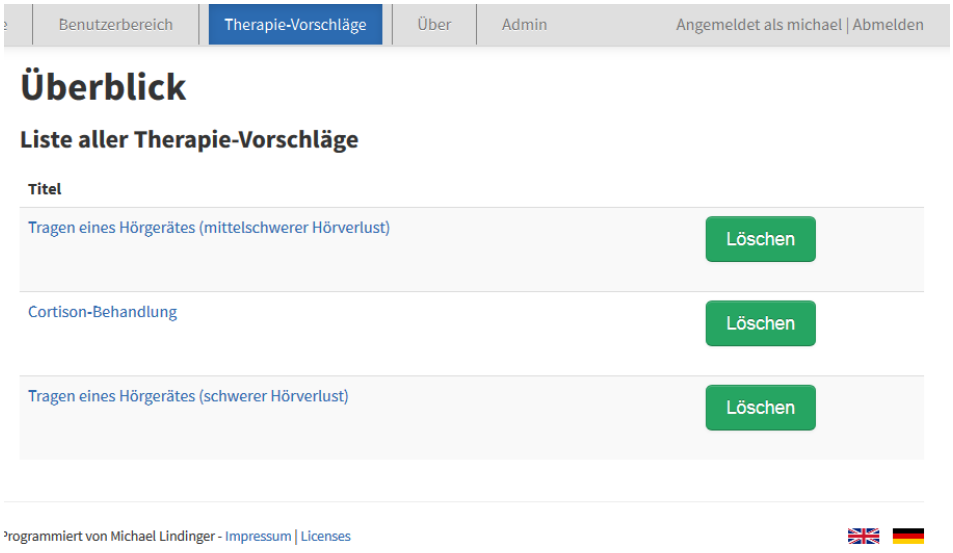


Abbildung 5.12: Übersichtsseite der Therapievorschläge

In der rechten, unteren Ecke von Abbildung 5.12 sind außerdem zwei Icons in Form der britischen bzw. deutschen Flagge dargestellt. Durch einen Klick auf ein solches Icon kann die Sprache der Webseite festgelegt werden. Aktuell werden zwar nur deutsch und englisch unterstützt, da Sprachen in Laravel jedoch sehr einfach hinzugefügt werden können, sind für die Zukunft weitere Sprachversionen geplant.

5.2 Architektur der App

Dieser Abschnitt beschreibt die Architektur der App anhand von ausgewählten Implementierungsaspekten. Die App ist nativ in Java geschrieben und, wie der untere Bereich von Abbildung 5.13 zeigt, nach dem Model-View-Controller Prinzip aufgebaut. Einige Gründe, wieso sich eine native Implementierung von komplexen, mobilen Anwendungen lohnt, finden sich in [RPR11][SSP+13][GPSR13]. Bis auf die Darstellung der Hörtestergebnisse wird die App ausschließlich im Hochformat ausgeführt. Grund dafür ist, dass vielen wichtigen Funktionen eine vertikal scrollbare View zugrunde liegt. Der Platz des Bildschirms kann im Hochformat somit deutlich besser ausgenutzt werden.

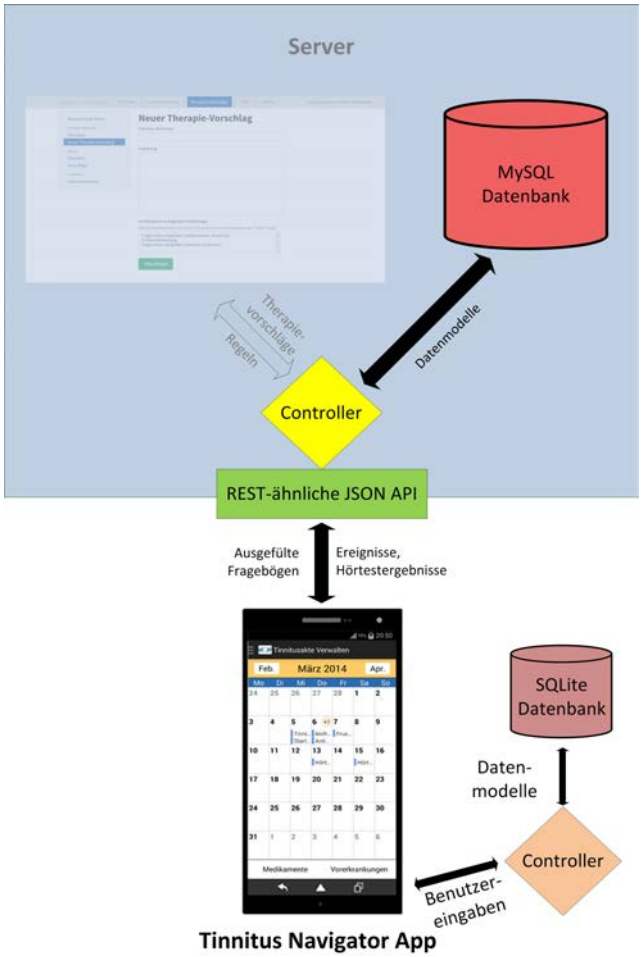


Abbildung 5.13: Model-View-Controller-Pattern der App

Zunächst werden nun grundlegende Strukturen der App vorgestellt. Dazu gehört neben der Klasse *Login* die Hauptklasse *MainActivity*. Anschließend werden alle Bereiche der App näher erläutert, die zur Gewinnung eines Therapievorschlags beitragen. Dies schließt insbesondere die persönliche Tinnitusakte und alle mit ihr verknüpften Funktionen zur Dateneingabe ein. Abschließend wird dann erläutert, wie aus den eingegebenen Daten Therapievorschläge ermittelt werden. Außerdem wird eine alternative Auswertungsstrategie vorgestellt.

5.2.1 Der Login-Vorgang

Voraussetzung für die Nutzung von Tinnitus Navigator ist ein Account im Track Your Tinnitus Projekt [Her]. Dieser kann entweder über die Website oder direkt in der App eingerichtet werden. Nach erfolgreicher Registrierung kann sich der Nutzer in die App einloggen. Den gesamten Login-Vorgang mit allen beteiligten Klassen zeigt das Ablaufdiagramm in Abbildung 5.14. Der eigentliche Login findet dabei in der Klasse *Login* statt.

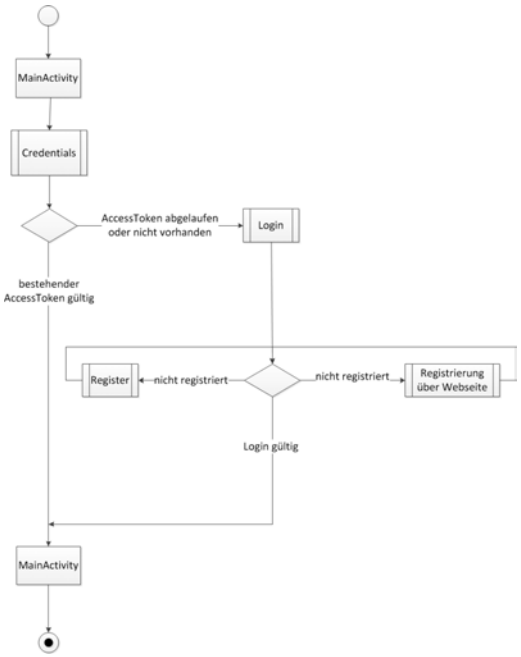


Abbildung 5.14: Flussdiagramm, das alle am Login beteiligten Klassen enthält

In obigem Ablaufdiagramm ist zu erkennen, dass zunächst die Klasse *MainActivity* geladen wird. *MainActivity* ruft direkt nach dem Start die Methode *checkAccessToken-ForValidity* in der Klasse *Credentials* auf, um zu prüfen, ob sich der Nutzer bereits erfolgreich auf dem Server authentifiziert hat. Da dies nach einer Neuregistrierung nicht der Fall ist, wird der Nutzer zur Login-Seite weitergeleitet. Nach erfolgreichem Login ist in der App ein Access-Token hinterlegt. Anders als der Name *SharedPreferences* in Zeile 3 von Listing 5.9 vermuten lässt, kann auf den Access-Token nur die App selbst zugreifen. Öffnet der Nutzer zu einem späteren Zeitpunkt erneut die Tinnitus Navigator App und ist der Access-Token weiterhin gültig, wird der Nutzer in Zeile 12 sofort zur Startseite der App weitergeleitet. Durch diese Implementierungsart wird verhindert, dass sich der Nutzer bei jedem Aufruf der App erneut einloggen muss, selbst wenn die Sitzung noch gültig ist. Ist der Access-Token hingegen abgelaufen, wird der Nutzer über die Methode *getUserData* zu einem erneuten Login aufgefordert (Zeile 36 & 40).

Listing 5.9: Die Methode *checkAccessTokenForValidity*

```
1  public void checkAccessTokenForValidity(boolean fastCheck) {
2
3     SharedPreferences prefs = activity.getSharedPreferences(
4                               Container.tinnitusnavigator, 0);
5     final String accessToken = prefs.getString(
6                                Container.accessToken, null);
7
8     if (accessToken == null) {
9        Intent i = new Intent(activity, Login.class);
10       activity.startActivityForResult(i, 0);
11    }
12    else if (accessToken != null && fastCheck) {
13
14   /*
15    * accelerates start of the app since accessToken is already
16    * existing and most certainly valid even if accessToken is
17    * not valid anymore, user will be prompted to enter
```

```
18      * credentials not before a method requests a valid
19      * accessToken
20      */
21
22      } else {
23        RequestParams params = new RequestParams();
24        params.put(Container.accessToken, accessToken);
25        RestClient.get("api/user", params,
26                       new CustomJsonHandler(activity) {
27          @Override
28          public void onSuccess(JSONObject response) {
29              Log.d("Credentials",
30                    "success - accessToken in DB is still valid");
31           }
32
33           public void onFailure(Throwable e,
34                                 JSONObject errorResponse) {
35              Log.d("Credentials", "Error in checking User");
36              getUserData();
37           }
38
39           public void onFailure(Throwable e) {
40              getUserData();
41              Log.d("Credentials", "Error in checking User");
42              System.out.println(e.getLocalizedMessage());
43           }
44        });
45      }
46  }
```

Nachdem sich der Nutzer zum ersten Mal in die App eingeloggt hat, werden Fragebögen und Fragen sowie Therapievorschläge und Regeln vom Server geladen werden. Dies geschieht durch mehrere, innere Klassen in der Klasse *Login*, die im Klassendiagramm in Abbildung 5.15 dargestellt sind. Fragebögen mit den zugehörigen Fragen werden durch *DownloadQuestionnairesAndQuestions*, Therapievorschläge durch *ReommendationsDownload* und Regeln durch *RulesDownload* in die App geladen. Bei Nutzern, die bereits einen Account haben und die App beispielsweise aufgrund eines Gerätewechsels erneut einrichten müssen, werden außerdem bereits eingegebene Antworten, Ereignisse und Hörtestergebnisse durch *AnswersDownload*, *EventsDownload* und *AudiometryDownload* vom Server geladen. Das Prinzip, nach dem der Download der Daten abläuft, ist für alle diese inneren Klassen gleich und erfolgt wie in Abbildung 5.13 dargestellt über eine REST-ähnliche JSON-API [RES]. Ein direkter Zugriff auf die auf dem Server gespeicherten Daten erfolgt somit nicht.

Abbildung 5.15: Klassendiagramm der Klassen *Login*, *MainActivity* und *Credentials*

Anhand von Abbildung 5.16 wird der Download-Vorgang nun erläutert. Zunächst wird eine Verbindung zur API hergestellt. Damit der Server den Nutzer eindeutig identifizieren kann, muss dabei der Access-Token mit übertragen werden (1). Konnte durch die API in (2) der Nutzer erfolgreich bestimmt werden, werden die Datenmodelle aus der Datenbank geladen und als Antwort an die App geschickt (3). Die App empfängt dann in (4) die Datenmodelle und speichert sie schließlich in (5) in der lokalen SQLite-Datenbank.

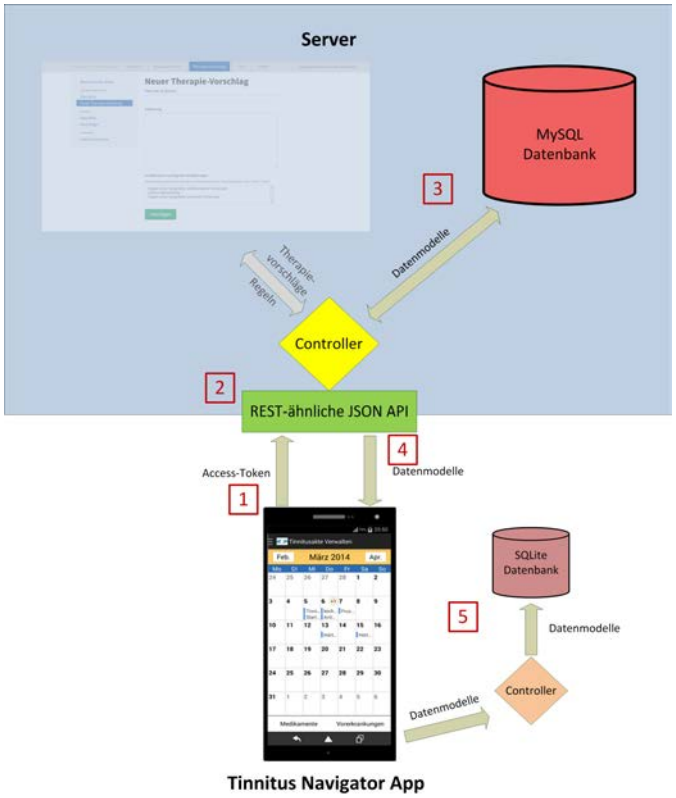


Abbildung 5.16: Herunterladen der Daten vom Server in die App

Da die Datenübertragung über Funk oftmals fehleranfällig ist, muss ein inkonsistenter Zustand der Datenbank durch fehlerhafte Datenpakete verhindert werden. An einem Ausschnitt der inneren Klasse *RulesDownload* zeigt Listing 5.10 in den Zeilen 31 - 36, dass in diesem Fall alle bis dahin durchgeführten Datenbankaktionen rückgängig gemacht. Zuvor ist in den Zeilen 2 – 18 dargestellt, wie ein Datenobjekt aus der Antwort extrahiert wird. In Zeile 19 wird dieses Datenobjekt anschließend in der lokalen Datenbank gespeichert.

Listing 5.10: Ausschnitt aus der inneren Klasse *RulesDownload*

```
try {
   JSONObject jsonEventObject = response.getJSONObject(i);

   int rule_id = jsonEventObject.getInt("rule_id");
   int recommendation_id = jsonEventObject.
                           getInt("recommendation_id");
   String table = jsonEventObject.getString("table");
   String column = jsonEventObject.getString("column");
   String filter = jsonEventObject.getString("filter");
   String value = jsonEventObject.getString("value");
   int connected_rule =
      jsonEventObject.isNull("connected_with") ?
         -1 : jsonEventObject.getInt("connected_with");
   String connection_type = jsonEventObject.
                            getString("connection_type");
   Rule rule =
      new Rule(rule_id,recommendation_id, table, column,
               filter, value, connected_rule, connection_type);
   if(rule.insertRulesObject(rulesDatabase)==0 ) {
       Log.d(getClass().getName(),
             "New Rule-row was added: " + rule_id +
             ", " +  table +", " + column); }
   else {
      onFailure(null);
      rulesDatabase.endTransaction();
      rulesDatabase.close();
      rulesHelper.close();
      return;
   }
} catch (JSONException e) {
```

```
31      onFailure(e);
32      e.printStackTrace();
33      rulesDatabase.endTransaction();
34      rulesDatabase.close();
35      rulesHelper.close();
36      return;
37      }
38  }
```

5.2.2 Die Klasse MainActivity

Nachdem durch die Klasse *Login* alle Daten in die App geladen wurden, wird die Klasse *MainActivity* wieder aufgerufen. Bei *MainActivity* handelt es sich um die zentrale Klasse der App, die neben dem bereits vorgestellten Login-Prozess alle Kommunikationsabläufe innerhalb der App steuert. Außerdem implementiert *MainActivity* ein sogenanntes Sliding-Menü. Bei einem Sliding-Menü handelt es sich um ein spezielles Menü, welches durch Ziehen vom linken Bildschirmrand zur Bildschirmmitte aktiviert werden kann. Wie das Menü genau implementiert ist, wird nun im folgenden Abschnitt erläutert.

Sliding-Menü

Das Sliding-Menü ermöglicht den Aufruf der wichtigsten Funktionen von jeder Seite der App. Außerdem ist es so implementiert, dass es sich in der App auf mehrere Arten öffnen lässt

1. Ziehen vom Bildschirmrand zur Bildschirmmitte
2. Drücken der Menütaste des Smartphones
3. Klick auf das App-Icon in der Actionbar

Auf diese Weise kann auch auf das Menü zugegriffen werden, wenn das Smartphone nur mit einer Hand bedient wird.

Das Sliding-Menü basiert auf dem Konzept des *NavigationDrawers* [Goog]. Dabei handelt es sich um ein Designelement, das in der App jederzeit durch Ziehen von der linken Seite zur Mitte des Bildschirms aktiviert werden kann. Wird das Menü in umgekehrter Richtung zurück an den linken Bildschirmrand gezogen, verschwindet es wieder. Das Besondere am *NavigationDrawer* ist, dass der darunterliegende Inhalt nur überdeckt und nicht wie bei einem neuen Fragment von diesem ersetzt wird. Das Layout, das dem Sliding-Menü zugrunde liegt, ist eine ListView. Bei einer ListView handelt es sich um einen Layout-Typ, der eine Liste darstellt, die mit einer Wischbewegung nach oben bzw. unten mit dem Finger durchblättert werden kann [Goof].

Wie Abbildung 5.17 zeigt, besteht jeder Eintrag einer ListView wiederum aus einer View, in welche die anzuzeigenden Daten integriert sind. Die Daten selbst werden dabei in Datenstrukturen wie Arrays oder Listen gespeichert. Dies trifft auch auf die Implementierung des Sliding-Menüs zu, dem eine ArrayList zugrunde liegt. Diese ArrayList speichert zu jedem Sliding-Menü-Eintrag ein Objekt, welches aus zwei Attributen besteht

- title: Name der Funktion, die der Nutzer aufrufen kann
- icon: Referenz auf eine Bilddatei, welche mit der Funktion assoziiert wird

Da die ListView als Layout keine Logik enthält, müssen ihr die Listenelemente durch ein Bindeglied zur Verfügung gestellt werden. Dieses Bindeglied wird in Android als Adapter bezeichnet. Der allgemeine Kommunikationsablauf zwischen Adapter und View ist in Abbildung 5.17 zu sehen.

Jedes Mal, wenn die ListView ein neues Listenelement an einer bestimmten Position darstellen soll, ruft diese den Adapter auf (1). Der Adapter bettet dann die Daten, die sich in der zugrunde liegenden Datenstruktur an Position n befinden, in eine View ein (2). Anschließend gibt der Adapter diese View an die ListView zurück, die die View dann an der Position n darstellt (3). Abhängig von der Datenstruktur, in der die darzustellenden Daten gespeichert sind, bietet Android verschiedene Adapter an. Daten aus einer Datenbank werden der View am besten mit einem CursorAdapter zur Verfügung gestellt. Bei Daten, die textuell dargestellt werden können und in einem Array oder einer Liste gespeichert sind, ist die Verwendung des ArrayAdapters empfehlenswert.

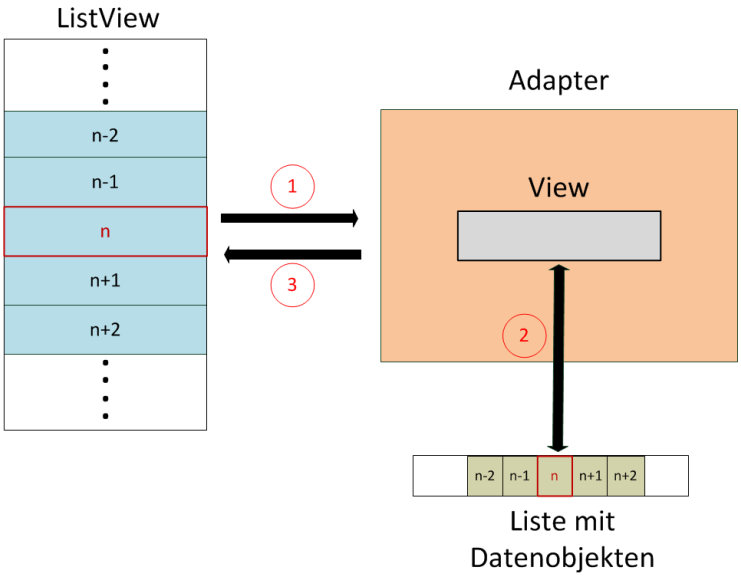


Abbildung 5.17: Allgemeine Funktionsweise eines Adapters am Beispiel einer ListView

Zusätzlich dazu können aber auch individuelle Adapter erstellt werden, wie dies beim Sliding-Menü der Fall ist. Allen Adaptern gemein ist, dass sie von der Klasse *BaseAdapter* erben [Gooa]. Tabelle 5.3 gibt einen Überblick über die Methoden dieser Klasse.

Methode	**Funktion**
CustomSlidingMenuListAdapter (Context c)	Konstruktor
getCount()	Gibt die Zahl der Datenelemente zurück
getItem(int pos)	Gibt das Datenelement an der Stelle pos zurück
getIdemID(int pos)	Gibt die ID des Datenelements an der Stelle pos zurück
getViewTypeCount	Gibt die Anzahl der verschiedenen View-Typen zurück, die durch getView() erzeugt werden können
getItemViewType(int pos)	Gibt den View-Typ jedes Datenelements anhand seiner Position pos zurück
getView(int pos, View view, ViewGroup vg)	Gibt eine View view zurück, in die das Datenelement an Position pos eingebettet ist

Tabelle 5.3: Die wichtigsten Methoden der Klasse *BaseAdapter*

Der letzte Eintrag von Tabelle 5.3 zeigt die Methode *getView*. Der Ablauf dieser Methode wird nun anhand von Listing 5.11 kurz erklärt. Danach werden zwei Aspekte vorgestellt, mit denen die Performanz einer ListView erhöht wird.

Jedes Mal, wenn die ListView ein neues Element darstellen soll, wird die Adapter-Methode *getView* gerufen. Die Position, an welcher sich die Daten der zugrunde liegenden Datenstruktur befinden, wird durch den Parameter *pos* angegeben. Außerdem wird ein *View*-Objekt *view* übergeben. Dieses enthält allerdings nur Daten, wenn die View durch den Adapter bereits schon einmal verarbeitet wurde. Basiert das Listenelement auf einer neuen View muss der Adapter mit einem LayoutInflater die View aus einer XML-Datei laden und daraus ein *View*-Objekt erstellen (Zeile 14 - 17). Anschließend werden die Komponenten der View mit den entsprechenden Daten befüllt (Zeile 34 - 37). Zum Schluss wird die fertige View an die ListView zurückgegeben (Zeile 42).

Listing 5.11: Die Methode *getView* der Adapter-Klasse *BaseAdapter*

```
1  public View getView(int pos, View view, ViewGroup vg) {
2        ListHolder lh = null;
3        int itemType = getItemViewType(pos);
4        if (view == null) {
5              switch(itemType){
6              case 0:
7                 lh = new ListHolder();
8                 view = inflater.inflate
9                 (R.layout.navigator_custom_sliding_menu_list
10                          _item_empty, null);
11             break;
12             case 1:
13                lh = new ListHolder();
14                view =
15                inflater.inflate(R.layout.navigator_custom
16                                    _sliding_menu_list_item,
17                                 null);
18                lh.iv = (ImageView) view.findViewById(R.id.icon);
```

```
19                lh.tv = (TextView) view.findViewById(R.id.title);
20                break;
21             default:
22             break;
23             }
24             view.setTag(lh);
25       }
26       else{
27             lh = (ListHolder) view.getTag();
28       }
29
30       switch(itemType){
31       case 0:
32       break;
33       case 1:
34          lh.iv.setImageResource(slidingMenuItems.
35                                 get(pos).getIcon());
36          lh.tv.setText(slidingMenuItems.
37                                 get(pos).getTitle());
38             break;
39       default:
40       break;
41       }
42    return view;
43    }
44 }
45
46 private class ListHolder{
47     ImageView iv;
48     TextView tv;
49 }
```

Wie eben erwähnt, wird der Methode *getView* ein *View*-Objekt übergeben. Grund dafür ist, dass das Laden einer XML-Datei sowohl CPU als auch Speicher stark beansprucht. Um die Auslastung der Systemressourcen möglichst gering zu halten, gibt es in Android eine Recycler-Funktion, die die zugrunde liegende View eines Listenelements zwischenspeichert, sobald diese durch Scrollen aus dem sichtbaren Bereich verschwindet. Wenn dann ein neues Listenelement angezeigt werden soll, das auf der gleichen View basiert, muss der Adapter die View nicht mehr aus der XML-Datei laden, sondern kann direkt auf das übergebene *View*-Objekt zurückgreifen. Der Recycler macht sich dabei die Tatsache zu Nutze, dass sich beim Scrollen meistens nur der Inhalt einer View ändert, jedoch nicht diese selbst. Das Prinzip des Recyclers ist in Abbildung 5.18 noch einmal dargestellt [Gooh].

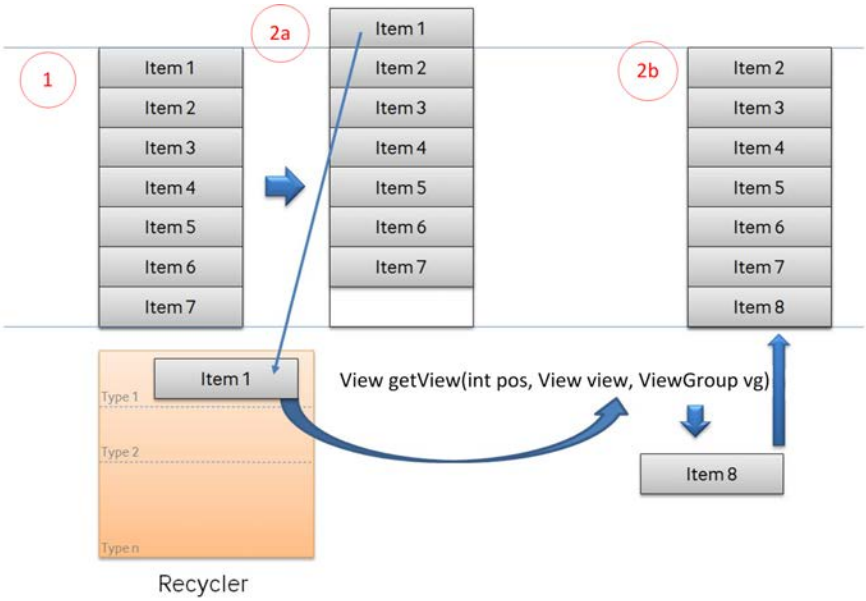


Abbildung 5.18: Funktionsweise des Recyclers

In Schritt 1 ist zunächst der Ausgangszustand der ListView und des Recyclers zu sehen. Durch Scrollen in der ListView verschwindet in Schritt 2a das erste Listenelement *Item1* aus dem sichtbaren Bereich. Da im Recycler das *View*-Objekt von *Item1* noch nicht vorhanden ist, wird dieses nun im Recycler gespeichert. In Schritt 2b wird dann das neue Listenelement *Item8* sichtbar.

Da dieses Element auf der gleichen View basiert wie *Item1*, enthält der Parameter *view* in der Methode *getView* eine Referenz auf dieses *View*-Objekt. Bei jeder anderen View würde der Parameter hingegen den Wert NULL annehmen und der Adapter müsste die View mit dem LayoutInflater aus der XML-Datei laden. Insgesamt muss Android somit immer nur die aktuell sichtbaren Views einer ListView und die Views im Recycler speichern. Der Speicherbedarf einer ListView kann auf diese Weise gering gehalten werden.

Ein weiterer Flaschenhals neben dem Laden einer View aus einer XML-Datei ist der Aufruf der Methode *findViewById*, mit der Komponenten einer View referenziert werden (z.B. Zeile 18). Um die Zahl der Aufrufe dieser Methode möglichst gering zu halten, wurde das View-Holder-Pattern eingeführt, welches in den Zeilen 46 - 49 dargestellt ist. Bei einem View-Holder handelt es sich um eine statische, innere Klasse, die zu allen Komponenten einer View der ListView Referenzen hält. Die Klasse *ListHolder* speichert zum Beispiel eine Referenz zu einer ImageView und eine Referenz zu einer TextView. Anschließend wird diese Referenz über die Methode *setTag* mit der jeweiligen View verknüpft (Zeile 24). Sollen zu einem späteren Zeitpunkt die Komponenten dieser View wieder mit Daten befüllt werden, wird über die Methode *getTag* die Instanz des ViewHolders geladen (Zeile 27). Danach kann auf die Komponenten direkt zugegriffen werden, ohne die Methode *findViewById* rufen zu müssen.

Das fertige Sliding-Menü ist in Abbildung 5.19 dargestellt. Wie darauf zu erkennen ist, müssen Listenelemente in einer ListView nicht alle auf der gleichen View basieren. So besteht der eine View-Typ des Sliding-Menüs aus einem Text, der den Namen der Funktion enthält und aus einem Icon, welches mit der Funktion assoziiert wird. Der zweite View-Typ besteht hingegen lediglich aus einer horizontalen Linie, die als Trennstrich zwischen den einzelnen Funktionsgruppen dient. Welcher View-Typ jeweils verwendet werden soll, wird über die Methode *getItemViewType* bestimmt (Zeile 3). Auf dem Screenshot in Abbildung 5.19 ist außerdem gut zu erkennen, dass das Sliding-Menü nur über den aktuellen Bildschirminhalt geschoben wird und diesen somit nicht ersetzt.

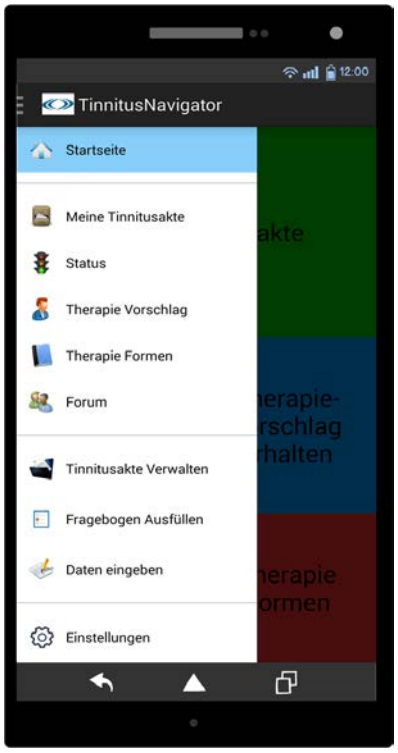


Abbildung 5.19: Sliding-Menü

Fragment Kommunikation

Bei einem Klick auf einen Menüeintrag der ListView wird die zugrunde liegende Funktion aufgerufen, welche jeweils als Fragment implementiert ist [Good]. Die Verwendung von Fragments ist im NavigationDrawer-Konzept empfehlenswert, weil sich bei einem Seitenwechsel in der App immer nur die als Fragment umgesetzte Benutzeroberfläche ändert, nicht jedoch die zugrunde liegende *MainActivity*. Der Programmcode wird dadurch modularer und kann somit besser wiederverwendet werden.
Abbildung 5.20 zeigt den Lebenszyklus eines Fragments in übersetzter Form [Goob]. Die drei wichtigsten Methoden des Fragment-Lebenszyklus werden nun kurz vorgestellt werden. Zunächst werden in der Methode *onCreate* die grundlegenden Komponenten des Fragments initialisiert. Im Anschluss daran wird durch *onCreateView* die Oberfläche des Fragments erstellt und das Fragment geht in den aktiven Zustand über. Sobald das Fragment nicht mehr im Vordergrund dargestellt wird, wird die Methode *onPause* gerufen.

Dies ist zum Beispiel der Fall, wenn das Fragment durch ein anderes Fragment ersetzt wird. Bei diesem Vorgang wird das Fragment nicht zerstört, sondern auf einem Stack gespeichert. Dadurch kann der Zustand des Fragments durch den Zurück-Button sofort wiederhergestellt werden, ohne dass die Komponenten und die darin enthaltenenen Daten erneut initialisiert werden müssen.

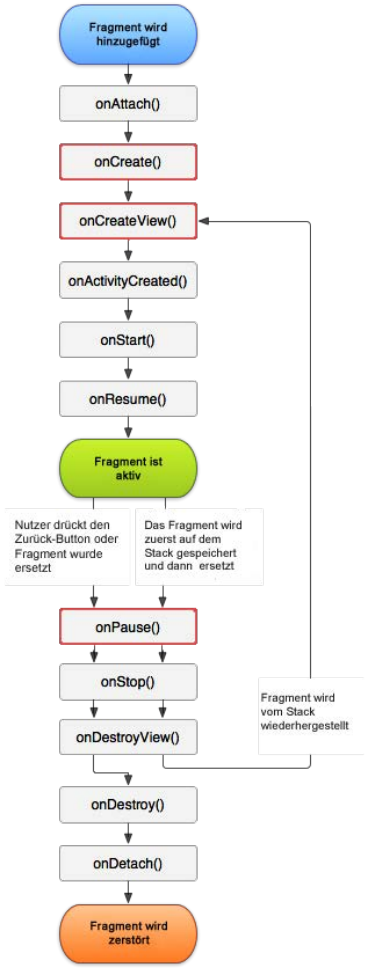


Abbildung 5.20: Lebenszyklus eines Fragments in Android

Da Fragments einen eigenen Lebenszyklus haben und als modularer Teil einer Activity nicht direkt mit anderen Fragments kommunizieren sollen, implementiert die Klasse *MainActivity* ein Interface zur Kommunikation der Fragments, welches als Klassendiagramm in Abbildung 5.21 dargestellt ist.

Die Methode *startfragment* des *FragmentCommunicationInterface* erwartet die ID des zu startenden Fragments. Für den Fall, dass dieses Fragment Daten erhalten soll, kann zusätzlich ein Bundle übergeben werden.

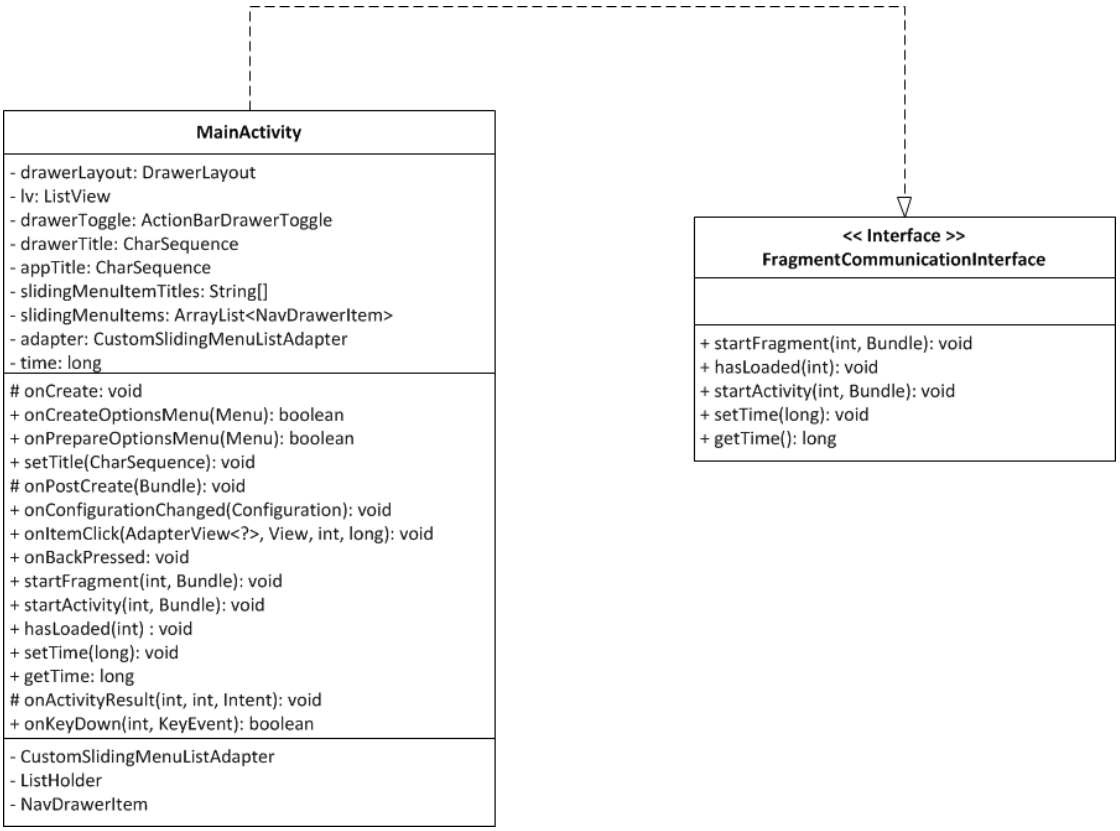


Abbildung 5.21: Klassendiagramm der Klassen *MainActivity* und *FragmentCommunicationInterface*

Soll nun ein Fragment A durch ein anderes Fragment B ersetzt werden, bedarf es eines Fragment-Managers, der sämtliche Fragments einer Actitvity verwaltet. Da alle Vorgänge innerhalb eines Fragment-Managers als Transaktionen durchgeführt werden, muss außerdem eine Fragment-Transaktion gestartet werden. Innerhalb dieser wird das zu ersetzende Fragment A dann auf dem Stack abgelegt und das neue Fragment B geladen. Abgeschlossen wird die Transaktion schließlich durch den Aufruf eines Commits. Zu beachten gilt, dass der Commit nicht sofort ausgeführt wird, sondern in den Ablaufplan des Haupt-Threads integriert wird.

Dort kümmert sich dann ein Scheduler um die rechtzeitige Durchführung der Transaktion. Listing 5.12 zeigt beispielhaft das Laden des Startbildschirms durch die Methode *startFragment*.

In Zeile 3 wird durch Aufruf von *getSupportFragmentManager* zunächst eine neue *FragmentManager*-Instanz erzeugt. Da Android kontinuierlich weiterentwickelt wird, kommt es immer wieder vor, dass neue Programmierkonstrukte entstehen. Dazu gehört auch der FragmentManager, der erst mit Android 3.0 eingeführt wurde [Gooc]. Damit auch ältere Android-Versionen von solchen neuen Strukturen profitieren können, gibt es die Android Support Library, deren Methoden das Präfix „getSupport“ tragen.

Listing 5.12: Ausschnitt aus der Methode *startFragment* des FragmentCommunication-Interfaces

```
1  public void startFragment(int fragmentID, Bundle b) {
2     Fragment f = null;
3     FragmentManager fm = getSupportFragmentManager();
4     switch(fragmentID){
5
6        // HomeScreen
7        case 0:
8           f = new HomeScreen();
9        break;
10
11      /*
12       * 19 weitere Fölle zum Start der einzelnen Fragments
13       */
14
15       // Call last fragment
16       case -1:
17          fm.popBackStack();
18          return;
19       default:
```

```
20          break;
21      }
22
23      if( f!=null ){
24          if( b!=null ) f.setArguments(b);
25          fm.beginTransaction().addToBackStack(null).
26                      replace(R.id.fragment_container,f).commit(); }
27  }
```

Anschließend wird in Zeile 4 anhand von *fragmentID* das neue Fragment bestimmt. Da in diesem Fall der Startbildschirm aufgerufen werden soll, hat *fragmentID* den Wert 0. Bei dieser ID handelt es sich nicht um die androidinterne Fragment-ID, sondern um eine App-spezifische ID. Diese basiert auf derjenigen Position, auf der sich die vom Nutzer aufgerufene Seite im Sliding-Menü befindet. Seiten, die nicht über das Sliding-Menü aufgerufen werden können, erhalten eine *fragmentID*, die größer als 16 ist. Besitzt die *fragmentID* den Wert -1 (Zeile 16), wird das oberste Fragment auf dem Stack wiederhergestellt. Insgesamt wird für den Start eines Fragments somit nur eine einzige Methode benötigt, egal ob der Aufruf über das Sliding-Menü oder über eine Seite der App stattfindet. Anschließend wird in Zeile 25 die Transaktion gestartet. Das alte Fragment wird auf den Stack gelegt und durch das Fragment des Startbildschirms in Zeile 8 ersetzt. Zum Schluss wird die Transaktion mit einem Commit beendet und der Startbildschirm wird geladen.

5.2.3 Startbildschirm

Über den Startbildschirm, der in Abbildung 5.22 links abgebildet ist, kann über eine Kachel unter anderem direkt die persönliche Tinnitusakte aufgerufen werden. Bei den Kacheln handelt es sich um Buttons, deren Standard-Layout durch einen pastellfarbigen Hintergrund verändert wurde. Als Inspiration für die Kacheloptik diente die App Flipboard, die sowohl Nachrichten aus sozialen Medien als auch von anderen Nachrichtenportalen darstellt [Fli]. Zum Vergleich ist diese in Abbildung 5.22 rechts dargestellt.

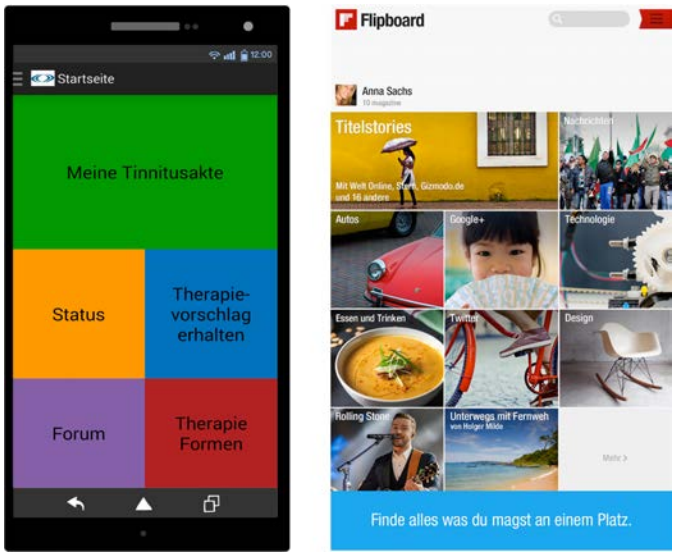


Abbildung 5.22: Kacheloptik des Startbildschirms (links), inspiriert durch die App Flipboard (rechts)

Die Entscheidung, den Startbildschirm in Kacheloptik zu implementieren hat zwei Gründe. Zum einen können in einer zukünftigen Version zusätzlich zu der Beschriftung mnemonische Icons angezeigt werden, welche die Wiedererkennungsrate der zugrunde liegenden Funktion erhöhen. Zum anderen sind Kacheln auch noch auf kleinen Bildschirmen gut bedienbar

5.2.4 Persönliche Tinnitusakte

Die persönliche Tinnitusakte in Abbildung 5.23 ist wie der Startbildschirm in Kacheloptik gehalten und bietet Zugriff auf alle drei Funktionen, die ein Nutzer regelmäßig ausführen muss, um einen Therapievorschlag zu erhalten:

1. Fragebögen ausfüllen
2. Ereignisse eintragen
3. Hörtestergebnisse eintragen

Diese Funktionen werden nun vorgestellt.

Abbildung 5.23: Die persönliche Tinnitusakte, die alle Eingabeoptionen in Kacheloptik darstellt

Auswahl und Ausfüllen von Fragebögen

Durch die immer stärker wachsende Zahl von mobilen Geräten hat die Verwendung von digitalen Fragebögen zur Datengenerierung in den letzten Jahren stark zugenommen [IRLP+13][RLPL+13][CNB+13][SRLP+13][SSP+14]. Auch in Tinnitus Navigator sind Fragebögen ein wichtiges Mittel, um Informationen über die Lebensgewohnheiten des Nutzers und dessen Umgang mit dem Tinnitus zu erhalten. Tinnitus Navigator stellt dazu mehrere Fragebögen zur Auswahl, die dem Nutzer mit Hilfe der externen Bibliothek *JazzyViewpager* von Jeremy Feinstein präsentiert werden [Fei]. Ein ViewPager ist ein Layout, das zur Darstellung mehrerer Views geeignet ist. Wie auf Abbildung 5.24 zu erkennen ist, lassen sich die Views dabei wie Seiten in einem Buch durch eine horizontale Wischbewegung durchblättern. Dieser Vorgang kann zusätzlich durch verschiedene Übergangseffekte visualisiert werden.

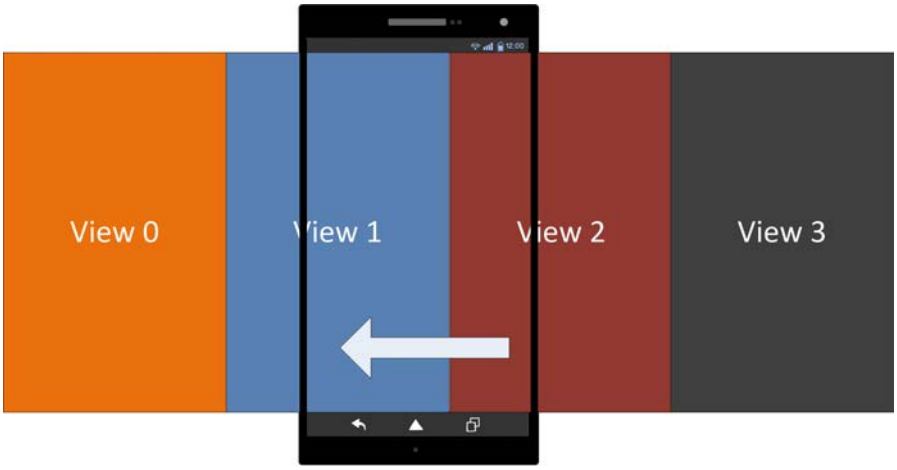


Abbildung 5.24: Funktionsweise eines ViewPagers

Die einzelnen Views werden dem ViewPager durch einen PagerAdapter zur Verfügung gestellt. Wie auch der BaseAdapter wirkt der PagerAdapter als Bindeglied zwischen einer View und den ihr zugrunde liegenden Daten. In einigen Punkten unterscheidet sich allerdings der PagerAdapter vom BaseAdapter. So existiert beim PagerAdapter zum Beispiel keine Recycler-Funktion, mit der eine View wiederverwendet werden kann. Die wichtigsten vier Methoden des PagerAdapters und ihre Aufgaben werden in Tabelle 5.4 gezeigt:

Methode	**Funktion**
instantiateItem(ViewGroup vg, int pos)	Erstellen einer neuen View
destroyItem(ViewGroup vg, int pos, Object obj)	Löschen einer View
getCount()	Anzahl an verfügbaren Views
isViewFromObject(View v, Object obj)	Prüfung, ob View v mit dem korrekten Schlüsselobjekt *obj* verknüpft ist

Tabelle 5.4: Die wichtigsten Methoden der Klasse *PagerAdapters*

Mit Listing 5.13 wird nun der Ablauf der Methode *instantiateItem* vorgestellt, die eine neue View erstellt. Dabei wird auch auf einen besonderen Implementierungsaspekt eingegangen, der speziell beim ViewPager zu finden ist.

Jedes Mal, wenn durch eine Wischbewegung ein anderer Fragebogen dargestellt werden soll, wird die Methode *instantiateItem* aufgerufen. Da der PagerAdapter keine Recycler-Funktion unterstützt, muss zunächst die zugrunde liegende View aus einer XML-Datei geladen und daraus ein *View*-Objekt erzeugt werden (Zeile 4 - 6). Daraufhin werden die einzelnen Komponenten des *View*-Objekts mit dem Titel und der Beschreibung des jeweiligen Fragebogens gefüllt und das *View*-Objekt zu einer ViewGroup hinzugefügt (Zeile 8 – 21). Schließlich wird in Zeile 24 ein Schlüssel-Objekt zurückgegeben. Dieses Objekt muss im Gegensatz zum BaseAdapter nicht zwingend das *View*-Objekt selbst sein, sondern es kann auch ein mit der View assoziiertes Objekt verwendet werden. Das Schlüssel-Objekt dient ähnlich einem Pointer-Objekt in C++ der Referenzierung einer View im Speicher [Jac14]. Dadurch kann der ViewPager jede View im PagerAdapter-Array unabhängig von ihrer Position finden. Um zu überprüfen, ob eine View mit einem bestimmten Schlüssel verknüpft ist, wird die Methode *isViewFromObjekt* in Zeile 27 ff. aufgerufen. Ein Nebeneffekt dieser Methode ist, dass der ViewPager zur Laufzeit immer weiß, wo sich eine bestimmte View befindet.

Listing 5.13: Die Methoden *instantiateItem* und *isViewFromObject* der Klasse *PagerAdapter*

```
1  public Object instantiateItem(ViewGroup container,
2                                final int position) {
3
4      View view = inflater.inflate(R.layout.navigator_
5                           fragment_questionnaire_description
6                           , null);
7
8      TextView tv_title = (TextView)view.findViewById(
9                     R.id.textViewFragmentQuestionnaireTitle);
10     tv_title.setText(questionnaires.get(position).title);
11     TextView tv_description = (TextView) view.findViewById(
12               R.id.textViewFragmentQuestionnaireDescription);
13     tv_description.setText(questionnaires.
```

```
                            get(position).description);
    TextView tv_info =
        (TextView)view.findViewById(
                R.id.textViewFragmentQuestionnaireInfo);
    tv_info.setText("Derzeit gibt es keine persönlichen
                    Statistiken zu diesem Fragebogen");

    container.addView(view);
    jvp.setObjectForPosition(view, position);

    return view;
}

public boolean isViewFromObject(View view, Object obj) {

    if (view instanceof OutlineContainer) {
        return ((OutlineContainer) view).getChildAt(0) == obj;
    } else {
        return view == obj;
    }
}
```

Die Informationen zu den Fragebögen, die über den ViewPager angezeigt werden, erhält der PagerAdapter über die Methode *getQuestionnaires* (Listing 5.14). Diese Methode ruft alle Fragebögen in Form von *Questionnaire*-Objekten über eine *SQLiteOpenHelper*-Subklasse aus der Datenbank der App ab und speichert diese in einer Liste [Gooj].

Listing 5.14: Die Methode *getQuestionnaires* zum Download von Fragebögen

```
private void getQuestionnaires(){
    QuestionnaireDatabaseHelper questionnaireHelper =
                new QuestionnaireDatabaseHelper(getActivity());
    SQLiteDatabase questionnaireDatabase =
```

```
5                questionnaireHelper.getReadableDatabase();
6     questionnaires = questionnaireHelper.
7                      getAllQuestionnaires(questionnaireDatabase);
8     questionnaireDatabase.close();
9     questionnaireHelper.close();
10 }
```

Ein *Questionnaire*-Objekt besteht wie das entsprechende *Questionnaire*-Datenmodell auf dem Server aus folgenden Attributen:

- qid: int - ID des Fragebogens
- title: String - Titel des Fragebogens
- description: String - Beschreibung des Fragebogens
- one_answer: int – Flag, wenn Antworttyp aller Fragen gleich ist
- type: String – Antworttyp, falls one_answer-Flag gesetzt
- configuration: String – Antwortformat, falls one_answer-Flag gesetzt

Hat sich der Nutzer für einen Fragebogen entschieden, kann er diesen durch eine Berührung des Bildschirms auswählen. Dieser Auswahlvorgang ist mit einem Gesture-Detector umgesetzt, der Benutzergesten wie beispielsweise Wischbewegungen über den Bildschirm erkennt. Listing 5.15 zeigt die Methode *onSingleTapConfirmed*, die auf einzelne Bildschirmberührungen reagiert.

In Android löst eine Berührung des Bildschirms zwei Ereignisse aus. Das Ereignis ACTION_DOWN beschreibt den Vorgang, bei dem der Finger den Bildschirm berührt. Das Ereignis ACTION_UP wird ausgelöst, wenn der Finger wieder angehoben wird. Aus diesem Grund prüft Zeile 2 zunächst, um welche der beiden Aktionen es sich handelt. Nur wenn der Nutzer auf eine Stelle des Bildschirms kurz drückt, wird das Fragment *QuestionnaireShowSelected* mit dem selektierten Fragebogen gestartet (Zeile 3 - 7). Drückt der Nutzer zu lange auf den Bildschirm oder rutscht er leicht ab, reagiert der GestureDetector nicht (Zeile 9).

Listing 5.15: Die Methode *onSingleTapConfirmed* zum Auswählen eines Fragebogens

```
1  public boolean onSingleTapConfirmed(MotionEvent e) {
2     if( e.getAction()==MotionEvent.ACTION_DOWN){
3        Bundle b = new Bundle();
4        b.putParcelable(Container.questionnaire,
5                    questionnaires.get(jvp.getCurrentItem()));
6        fci.startFragment(91, b);
7        return true;
8     }
9     else return false;
10 }
```

In den Zeilen 4 - 6 wird der Fragebogen in einem Bundle an das neue Fragment übergeben. Dies ist in Android bei Objekten nicht ohne weiteres möglich, da mit den Standardmethoden nur einfache Datentypen wie Strings oder Integers übertragen werden können. Um auch Objekte zwischen Activities bzw. Fragments austauschen zu können, müssen diese Objekte als Parcel verpackt werden. Dies ist ein Container, der beliebige Daten enthalten kann. Damit ein Objekt als Parcel übertragen werden kann, muss seine Klasse das Interface *Parcelable* implementieren. Listing 5.16 zeigt die Methoden, die dafür implementiert werden müssen am Beispiel der Modell-Klasse *Questionnaire*.

Listing 5.16: Die Klasse *Questionnaire*, die das Interface *Parcelable* implementiert

```
1  public class Questionnaire implements Parcelable{
2
3        public int qid;
4        public String title;
5        public String description;
6        public String type;
7        public String configuration;
8        public int one_answer;
9
10       public Questionnaire(){
```

```
11
12        }
13        public Questionnaire(Parcel parcel){
14            qid = parcel.readInt();
15            title = parcel.readString();
16            description = parcel.readString();
17            type = parcel.readString();
18            configuration = parcel.readString();
19            one_answer = parcel.readInt();
20        }
21        public void writeToParcel(Parcel dest, int flags) {
22            dest.writeInt(qid);
23            dest.writeString(title);
24            dest.writeString(description);
25            dest.writeString(type);
26            dest.writeString(configuration);
27            dest.writeInt(one_answer);
28        }
29        public static final Parcelable.Creator<Questionnaire>
30            CREATOR = new Parcelable.Creator<Questionnaire>() {
31                public Questionnaire createFromParcel(
32                                                final Parcel parcel){
33                    return new Questionnaire(parcel);
34                }
35
36                public Questionnaire[] newArray(final int size) {
37                    return new Questionnaire[size];
38                }
39        };
40    }
```

Zu dem Interface gehört die Methode *writeToParcel*, die alle Felder des *Questionnaire*-Objektes mit einem *write*-Befehl in ein Parcel packt (Zeile 21 - 28). Damit die Klasse *QuestionnaireShowSelected* das *Questionnaire*-Objekt anschließend wieder verwenden kann, muss das Parcel entpackt werden. Dessen Methode *createFromParcel* erstellt durch den Aufruf eines Parcel-spezifischen Konstruktors ein neues Objekt der Klasse *Questionnaire*. Dabei werden die Daten des Parcels mit dem *read*-Befehl ausgelesen und entsprechend den Feldern des *Questionnaire*-Objekts zugeordnet (Zeile 13 - 20).

Nachdem das *Questionnaire*-Objekt des Fragebogens erfolgreich entpackt werden konnte, werden alle Fragen dieses Fragebogens aus der Datenbank in einer Liste gespeichert. Jede Frage wird durch ein *Question*-Objekt repräsentiert, das folgende Felder enthält:

- qid: int – ID der Frage
- questionnaire_id: int - ID des zugehörigen Fragebogens
- question: String – Fragetext
- description: String – optionale Beschreibung der Frage
- type: String – Antworttyp der Frage
- configuration: String – Antwortformat
- position: int – Position der Frage im Fragebogen
- optional: int – Flag, ob Frage optional ist
- answer: Answer – Antwort auf die Frage in Form eines *Answer*-Objekts

Da in der Regel nicht alle Fragen eines Fragebogens auf einem Smartphone-Bildschirm angezeigt werden können, wird zur Darstellung des Fragebogens eine ListView verwendet. Die Views dieser ListView enthalten jeweils den Fragetext der Frage und das damit verknüpfte Antwortformat. Wie am Beispiel des Tinnitus Sample Case History Questionnaire-Fragebogens (TSCHQ) in Abbildung 5.25 zu sehen ist, enthält ein Fragebogen nicht immer nur Fragen, die auf einem einzigen Antworttyp basieren.

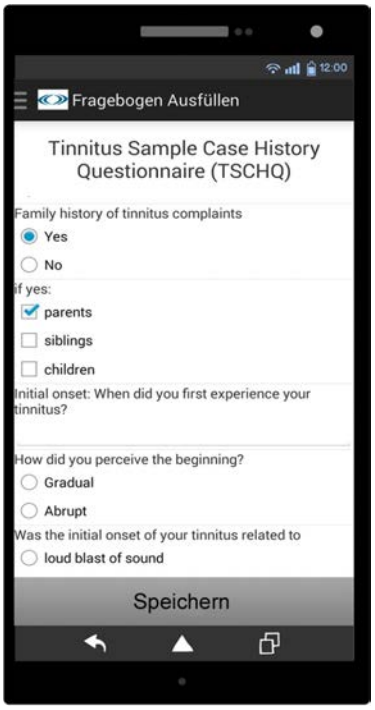


Abbildung 5.25: Darstellung eines Fragebogens in Tinnitus Navigators

Aus diesem Grund sind in Tinnitus Navigator die folgenden fünf Antworttypen enthalten:

- einzeiliger Text
- mehrzeiliger Text
- Einfachauswahl durch Radiobuttons
- Mehrfachauswahl durch Checkboxen
- Skala mit Schieberegeler

Jeder dieser Antworttypen basiert auf einer eigenen View, die der ListView wie schon beim Sliding-Menü über einen Adapter zur Verfügung gestellt wird. Damit der Adapter weiß, welche View an einer bestimmten Position dargestellt werden muss, müssen die beiden Methoden *getItemViewType* und *getViewTypeCount* zwingend implementiert werden. Diese sind in Listing 5.17 dargestellt. Die Methode *getViewTypeCount* gibt an, wie viele unterschiedliche Views in der gesamten ListView insgesamt existieren (Zeile 1 – 3).

In der Methode *getItemViewType* wird das mit der Frage verknüpfte *Question*-Objekt bestimmt (Zeile 5). Über das Attribut *type* dieses Objekts kann dann die der Frage zugrunde liegende View ermittelt werden (Zeile 6 ff).

Listing 5.17: Methoden der Adapter-Klasse *BaseAdapter* zur Darstellung mehrerer View-Typen

```
1  public int getViewTypeCount(){
2      return MAX_NUMBER_OF_TYPES;
3  }
4  public int getItemViewType(int position){
5      Question rowQuestion = questions.get(position);
6      if (rowQuestion.type.equals("radio_button")) {
7          return TYPE_RADIOBUTTON;
8      }
9      else if (rowQuestion.type.equals("checkbox")) {
10         return TYPE_CHECKBOX;
11     }
12     else if (rowQuestion.type.equals("text_multiline")) {
13         return TYPE_TEXTMULTILINE;
14     }
15     else if (rowQuestion.type.equals("int_scale")) {
16         return TYPE_SCALE;
17     }
18     else return TYPE_TEXT;
19 }
```

Tinnitus Navigator erlaubt es dem Nutzer, einen Fragebogen jederzeit zu unterbrechen und diesen zu einem späteren Zeitpunkt fertigzustellen. Damit die vom Nutzer eingegebenen Antworten bei einer solchen Unterbrechung nicht verloren gehen, registriert sich jede View an einem ChangeListener, der jedes Mal aktiv wird, wenn eine neue Antwort eingegeben wird. Antworten werden in TinnitusNavigator als Objekte der Klasse *Answer* gespeichert.

Ein *Answer*-Objekt besteht aus folgenden Feldern:

- answer_id: int – ID der Antwort
- questionnaire-id: int – ID des Fragebogens
- question_id: int – ID der Frage
- answer: String – Antwort
- sentsuccess: int - Flag zum Kennzeichnen einer erfolgreichen Speicherung auf dem Server

Listing 5.18 zeigt den Speichervorgang am Beispiel einer Antwort, die in ein Textfeld eingetragen wurde. Zunächst wird anhand der Position in der ListView die zugehörige Frage bestimmt (Zeile 2). Hat der Nutzer diese Frage bereits schon einmal beantwortet, kann die Antwort aus dem Textfeld direkt in das Feld *answer* des *Answer*-Objekts eingetragen und dann im zugehörigen *Question*-Objekt gespeichert werden (Zeile 10). Andernfalls muss zuerst ein neues *Answer*-Objekt erzeugt werden (Zeile 3 – 8).

Listing 5.18: Die Methode *onFocusChange* zum direkten Speichern von Antworten

```
1  public void onFocusChange(View v, boolean hasFocus){
2     Question question = questions.get(position);
3     if (question.answer == null) {
4        question.answer = new Answer();
5        question.answer.questionnaire_id =
6                          question.questionnaire_id;
7        question.answer.question_id = question.qid;
8     }
9     EditText textfield = (EditText) v;
10    question.answer.answer = textfield.getText().toString();
11 }
```

Mit der Methode *saveAnswersToDatabaseAndUpload* werden schließlich alle Antworten in der Tabelle *answers* gespeichert und anschließend zum Server übertragen. Schlägt der Upload der Antwort fehl, wird dies im Feld sentsuccess vermerkt.

5.2.5 Kalender

Neben dem Ausfüllen von Fragebögen spielt die Eingabe von Ereignissen bei der Bestimmung eines Therapievorschlags eine entscheidende Rolle. Der zentrale Ort, an dem sowohl zeitabhängige Ereignisse wie zum Beispiel eine Tinnitusattacke als auch zeitunabhängige Ereignisse wie Vorerkrankungen verwaltet werden können, ist der Kalender. Dieser wurde für Tinnitus Navigator eigens implementiert. Grund dafür ist, dass es kein bereits existierendes Kalenderframework gibt, welches die Anforderungen an den Kalender in der Tinnitusakte vollständig erfüllen kann. Durch die eigene Implementierung konnten gezielt die Funktionen in den Kalender integriert werden, die auch wirklich benötigt werden. Dies spart nicht nur Speicherplatz, sondern sorgt auch für eine höhere Performanz der App. Auch die Möglichkeit, die Zellen des Kalenders flexibel zu gestalten, hat diese Entscheidung beeinflusst.
Für die Darstellung des Kalenders kommen zwei Layouts in Frage, die beide zur Darstellung von Daten in zweidimensionaler Form geeignet sind:

- TableLayout
- GridView

Ein großer Nachteil des TableLayouts ist allerdings, dass für diesen Layout-Typ kein Adapter existiert. Für jede Zelle müsste daher die zugrunde liegende View geladen und mit Daten befüllt werden. Ähnlich einer ListView ohne Recycler-Funktion würde jede Aktualisierung des Kalenders so zu einer hohen Auslastung von CPU und Speicher führen. Alternativ wäre es zwar auch möglich, einen individuellen Adapter für das TableLayout zu entwickeln, dies hätte allerdings den Rahmen dieser Arbeit gesprengt. Einer GridView hingegen werden die einzelnen Views durch einen Adapter zur Verfügung gestellt. Mit einem ViewHolder, der Referenzen auf die einzelnen Komponenten der Views hält, kann die Zugriffszeit zudem weiter reduziert werden. Insgesamt empfiehlt es sich somit für die Umsetzung eines Kalenders auf eine GridView zurückzugreifen, deren Schema in Abbildung 5.26 dargestellt ist.
Der Kalender in Tinnitus Navigator zeigt jeweils einen Monat an. Jede Zelle der GridView steht dabei für einen Tag des Monats und zeigt die Titel von bis zu zwei Ereignissen eines Tages direkt an.

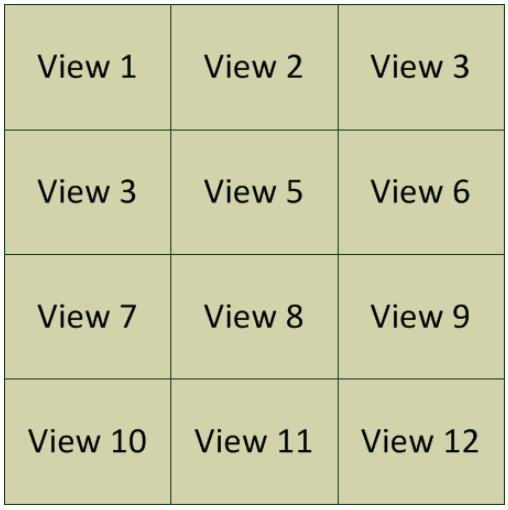


Abbildung 5.26: Struktur einer GridView

Durch einen längeren Druck auf eine Zelle können alle Ereignisse eingesehen und bearbeitet werden. Neue Ereignisse können durch zweimalige Berührung einer Zelle eingetragen werden. Unterhalb des Kalenders befinden sich außerdem zwei Buttons, über die Vorerkrankungen und Medikamente eingegeben werden können. Möchte der Nutzer einen anderen Monat auswählen, so kann er dies auf mehrere Wege umsetzen. Zum einen kann der Vormonat bzw. der nächste Monat über die beiden Buttons oberhalb des Kalenders aufgerufen werden. Zum anderen kann aber auch durch eine Wischgeste nach links bzw. nach rechts ein Monat vor- bzw. zurückgesprungen werden. Soll ein größerer Zeitraum übersprungen werden, kann über das Textfeld in der Mitte oben ein DatePicker aufgerufen werden, über den dann ein beliebiges Datum ausgewählt werden kann. In Abbildung 5.27 sind die beiden Funktionen Datumsauswahl und Ereignisübersicht dargestellt. Die Möglichkeit durch Wischgesten einen Monat vor- oder zurückzuspringen wird durch organgene Pfeile im Kalender angedeutet.
Jedes Mal, wenn der Nutzer einen anderen Monat im Kalender aufruft, müssen die Daten entsprechend aktualisiert werden. Dazu wird zunächst die Reihenfolge der Tage in der GridView aktualisiert. Insgesamt besteht die GridView aus 42 Zellen. Die Zahl der Zellen der GridView ist somit immer größer als ein Monat Tage hat.

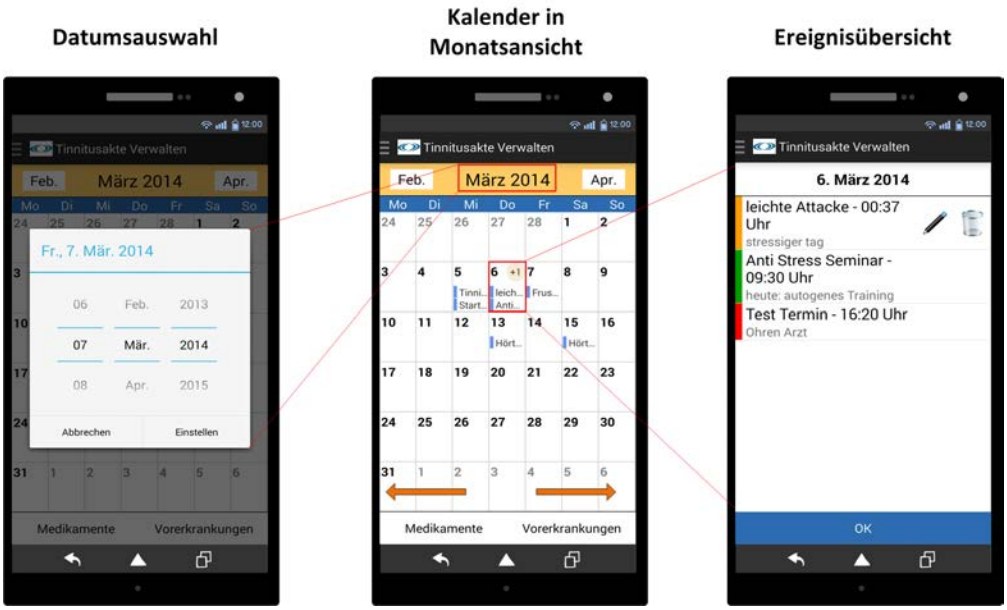


Abbildung 5.27: Datumsauswahl und Ereignisübersicht im Kalender

Damit dennoch alle Zellen belegt sind, werden die übrigen Zellen sowohl auf Tage des Vormonats als auch auf Tage des darauffolgenden Monats verteilt. Wie diese Berechnung erfolgt, wird anhand von Abbildung 5.28 am Beispiel vom Mai 2014 dargestellt.

Zunächst wird die Anzahl der Tage des Monats bestimmt (1). Diese beträgt im Mai 31. Anschließend wird die Position des Monatsersten in der GridView berechnet. Ist der Monatserste ein Montag, wird dieser der besseren Übersicht wegen erst in der zweiten Zeile dargestellt. Ansonsten werden die Monatsersten direkt in der ersten Zeile dargestellt. Der Monatserste im Mai 2014 ist ein Donnerstag. Ausgehend von dieser Berechnung werden dann die Tage des Vormonats ermittelt (2). Da durch den Donnerstag drei Zellen in der ersten Zeile noch frei sind, werden die Tage 28 – 30 vom April eingetragen. Anschließend werden alle Mai-Tage eingetragen (3). Die restlichen, verbleibenden Zellen werden schließlich mit Tagen des Junis aufgefüllt (4).

Nachdem die Kalendertage der GridView bestimmt sind, müssen dieser anschließend noch die Indikatoren, durch die der Nutzer sofort erkennen kann, ob an einem bestimmten Tag Ereignisse stattgefunden haben, hinzugefügt werden.

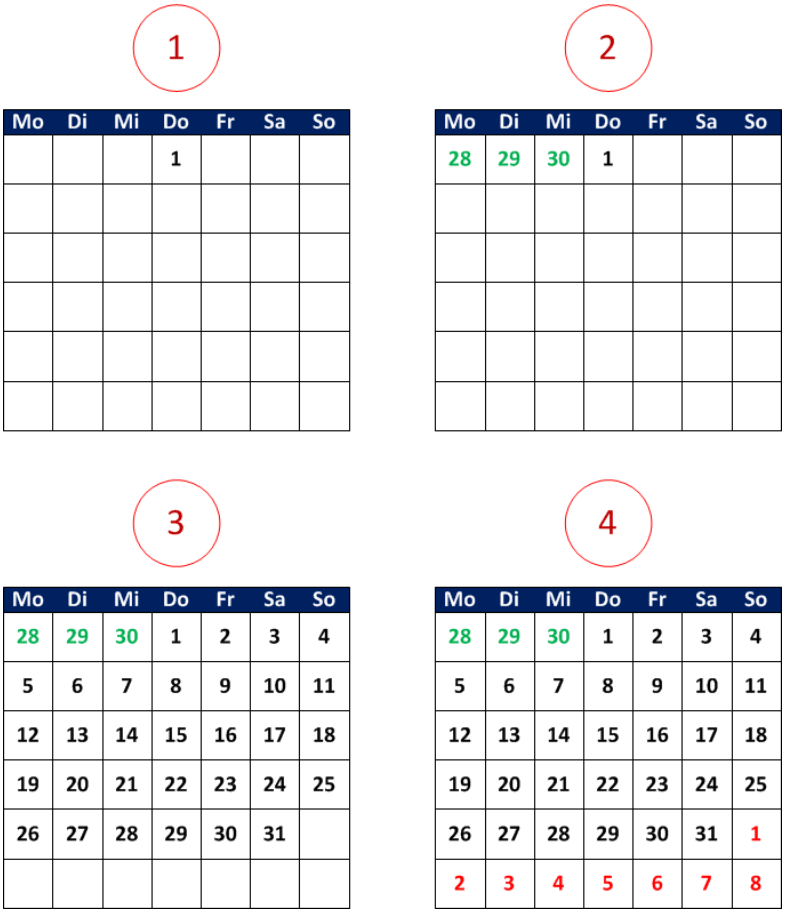


Abbildung 5.28: Berechnung der dargestellten Tage in der Monatsansicht

Aus diesem Grund werden zu jedem Tag des berechneten Zeitraums Informationen aus der Tabelle *calendarevents* geladen:

1. Titel des ersten gefundenen Ereignisses eines Tages
2. Titel des zweiten gefundenen Ereignisses eines Tages
3. Anzahl der weiteren Ereignisse eines Tages

Die fertige View einer Zelle ist skizzenhaft in Abbildung 5.29 dargestellt. Sie zeigt an, dass am 15. des Monats insgesamt fünf Ereignisse stattgefunden haben. Neben den zwei direkt angezeigten Ereignissen gibt es noch drei weitere Ereignisse, auf die der Indikator rechts oben hinweist.

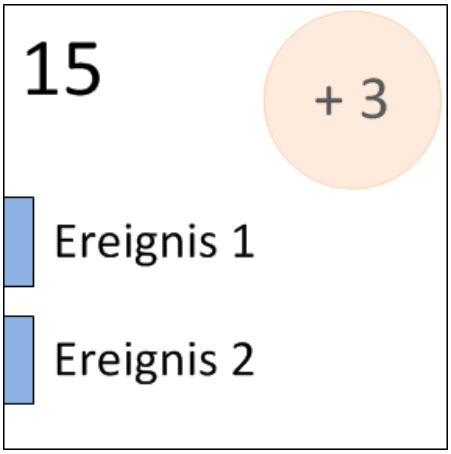


Abbildung 5.29: Layout einer Zelle im Kalender

Bislang wurde nur der Aufbau des Kalenders betrachtet, und erklärt, wie die Kalenderansicht aktualisiert wird. Im Folgenden werden nun die Strukturen beschrieben, die für das Eingeben eines neuen Ereignisses notwendig sind. Ein Ereignis wird in Tinnitus Navigator durch ein Objekt der Klasse *Event* repräsentiert und enthält folgende Felder:

- time: long –Datum des Ereignisses in Milisekunden
- type: int – ID des Ereignistyps
- title: String – Titel des Ereignisses
- description: String – Genaue Beschreibung des Ereignisses

Wie bereits erwähnt, kann ein neues Ereignis durch zweimalige Berührung einer Zelle im Kalender eingetragen werden. Nach der ersten Berührung verfärbt sich der Hintergrund der Zelle zunächst blau, sodass der Nutzer das selektierte Datum noch einmal überprüfen kann. Nur wenn der Nutzer innerhalb von fünf Sekunden erneut auf die voraktivierte Zelle drückt, öffnet sich das Ereignis-Eingabefenster. Dieser Vorgang, durch den die Zahl der Fehleingaben möglichst gering gehalten werden soll, ist in Abbildung 5.30 noch einmal dargestellt. Für die Umsetzung dieser Funktion bietet sich in Android die Verwendung eines Handlers an, mit dem unter anderem zeitgesteuert Threads ausgeführt werden können. Listing 5.19 zeigt den entsprechenden Code-Ausschnitt der Listener-Klasse, in der diese Funktion umgesetzt ist.

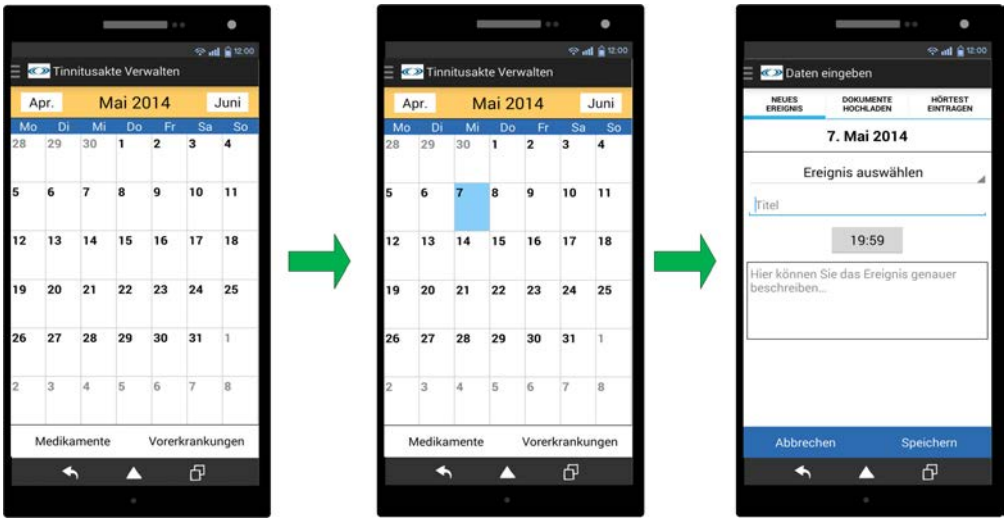


Abbildung 5.30: Selektierung eines Kalendertages

In den Zeilen 9 und 10 wird zunächst die ausgewählte Zelle referenziert und mit einem blauen Hintergrund versehen. Anschließend wird in Zeile 11 eine neue *Handler*-Instanz erzeugt, die den Thread in den Zeilen 13 – 16 nach fünf Sekunden ausführt.

Listing 5.19: Implementierung der Selektierung eines Kalendertages

```
1  if( lastSelected!=null && lastSelected==v ){
2        fci.setTime(c.getTimeInMillis());
3        fci.startFragment(10, null);
4  }
5  else{
6     if( lastSelected!=null )
7         lastSelected.setBackgroundColor(0xffffffff);
8
9     lastSelected = v;
10    v.setBackgroundColor(0xff87CEFA);
11    new Handler().postDelayed(new Runnable() {
12       @Override
13       public void run() {
14          lastSelected = null;
```

```
15            v.setBackgroundColor(0xffffffff);
16        }
17    }, 5000);
18 }
```

Wird die Zelle in dieser Zeit nicht mehr berührt, färbt sich der Hintergrund wieder weiß ein und die Referenz auf die zuletzt gedrückte Zelle wird wieder entfernt. Andernfalls wird bei einem erneuten Drücken auf die Zelle das Fragment *EventTabHost* gestartet, über welches Ereignisse, Dokumente und Hörtestergebnisse eingegeben werden können (Zeile 1 – 4).

Das Layout, das diesem Fragment zugrunde liegt, ist ein so genannter TabHost [Gook]. Dabei handelt es sich um einen Layout-Typ, mit dem mehrere, ähnliche Funktionen über eine einzige View dargestellt werden können. Dieser besteht, wie Abbildung 5.31 zeigt, aus einer Tab-Leiste und einem Container, der den Inhalt des jeweils selektierten Tabs anzeigt. Der TabHost ist somit ideal, um die Eingabemasken von Ereignissen, Dokumenten und Hörtestergebnissen in einer View zusammenzufassen.

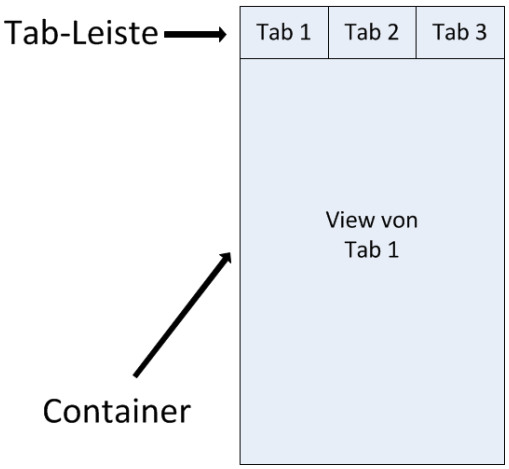


Abbildung 5.31: Layout eines TabHosts

In seiner Grundansicht zeigt *EventTabHost*, wie auf Abbildung 5.32 links zu sehen ist, die Eingabemaske für Ereignisse an. Da für einen erfolgreichen Therapievorschlag jedes Detail entscheidend sein kann, enthält die Eingabemaske ein Dropdown-Menü.

Dieses stellt zum Stand dieser Arbeit fünf Ereignistypen zur Auswahl bereit:

- Tinnitus-Attacke
- Psychisches Problem
- Arzttermin
- Therapie
- Sonstiges

Diese Auswahl soll dem Nutzer das Gefühl geben, dass jedes Ereignisse, das nur im entferntesten mit einem dieser Ereignistypen assoziiert werden kann, für einen guten Therapievorschlag von hoher Relevanz sein können. Zusätzlich dazu soll ein mehrzeiliges Textfeld den Nutzer animieren, das Ereignis genauer zu beschreiben. Damit das Ereignis im Kalender angezeigt werden kann, muss es schließlich über den Button *Speichern* in die Tabelle *calendarevents* geschrieben werden. Anschließend wird der Nutzer wieder zur Kalenderansicht weitergeleitet.

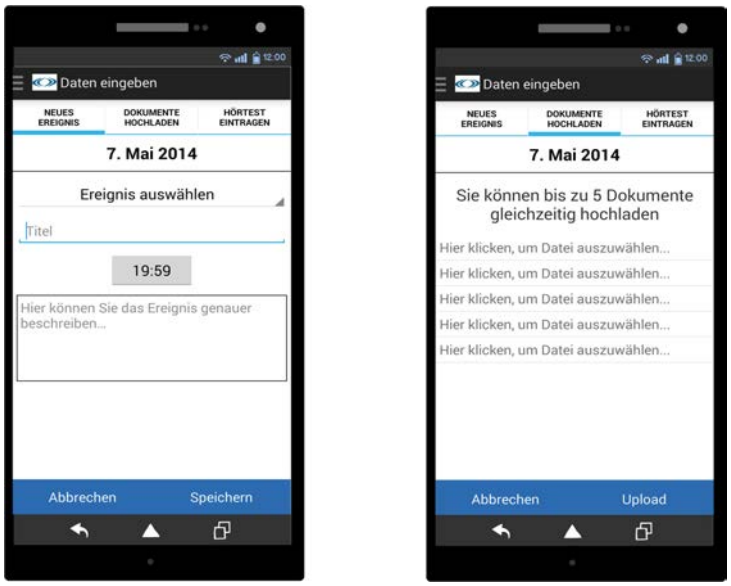


Abbildung 5.32: Eingabemaske für ein neues Ereignis in der Grundansicht des Tab-Hosts

Zusätzlich zu den fünf Ereignistypen des oben erwähnten Auswahlmenüs gibt es in Tinnitus Navigator weitere fünf Ereignistypen. Diese sind allerdings nicht frei wählbar, sondern an die Ausführung der ihnen zugeordneten Funktionen geknüpft:

- Hörtest
- Dateiupload
- Fragebogen
- Medikamente
- Vorerkrankungen

Die obersten zwei Funktionen können dabei direkt über EventTabHost aufgerufen werden. Über Dokumente hochladen können Dokumente wie Hörtestergebnisse oder Röntgenaufnahmen in den Dateiformaten PDF oder JPEG der App hinzugefügt und anschließend auf den Server hochgeladen werden. Die Anzahl der Dokumente, die gleichzeitig auf den Server hochgeladen werden können, ist dabei von der aktuellen Verbindungsqualität abhängig. Wenn wie in Abbildung 5.32 rechts eine Wifi-Verbindung besteht, können bis zu fünf Dokumente gleichzeitig hochgeladen werden. Besteht lediglich eine mobile Datenverbindung schwankt die Zahl zwischen drei (bei HSPA) und null (bei GSM) Dokumenten. Um das Datenvolumen nicht unnötig zu belasten, können bei einer mobilen Datenverbindung außerdem immer nur maximal 5MB auf einmal hochgeladen werden. Da zum Stand der Arbeit allerdings keine Software existiert, die diese Dokumente auswerten kann, ist der Dateiupload für die Erstellung eines Therapievorschlags irrelevant und wird daher nicht weiter betrachtet. Von großer Bedeutung für einen Therapievorschlag ist hingegen die Eingabe von Hörtestergebnissen, die im nächsten Abschnitt vorgestellt wird.

5.2.6 Eingabe von Hörtestergebnissen

Dieser Abschnitt stellt die wichtigsten Konstrukture vor, die zur Eingabe von Hörtestergebnissen benötigt werden. Wie bereits im Grundlagen-Kapitel 2 erwähnt wurde, wird bei einem Hörtest der Hörverlust auf beiden Ohren auf den gleichen Frequenzen geprüft.

Ein Hörtest wird in Tinnitus Navigator durch ein Objekt der Klasse *Audiometry* repräsentiert und enthält deshalb folgende Attribute:

- time: long – Datum des Hörtest in Milisekunden
- frequencies: ArrayList<ArrayList<String» - Frequenzen
- valuesLeft: ArrayList<ArrayList<String» - Ergebnissen des linken Ohrs
- valuesRight: ArrayList<ArrayList<String» - Ergebnissen des rechten Ohrs

Die Eingabemasken zum Eintragen von Hörtestergebnissen werden über das Tab *Hörtest* eintragen im *EventTabHost* erreicht, welches in Abbildung 5.33 links dargestellt ist. Zusätzlich dazu enthält das Tab auch noch einen Button, über den die Hörtestergebnisse in einem Liniendiagramm visualisiert werden können. Dieser ist standardmäßig deaktiviert und kann erst gedrückt werden, wenn die Ergebnisse für beide Ohren eingetragen sind (Abbildung 5.33 rechts). Wie dies geprüft wird, wird im nächsten Abschnitt *Visualisierung von Hörtestergebnissen* erläutert.

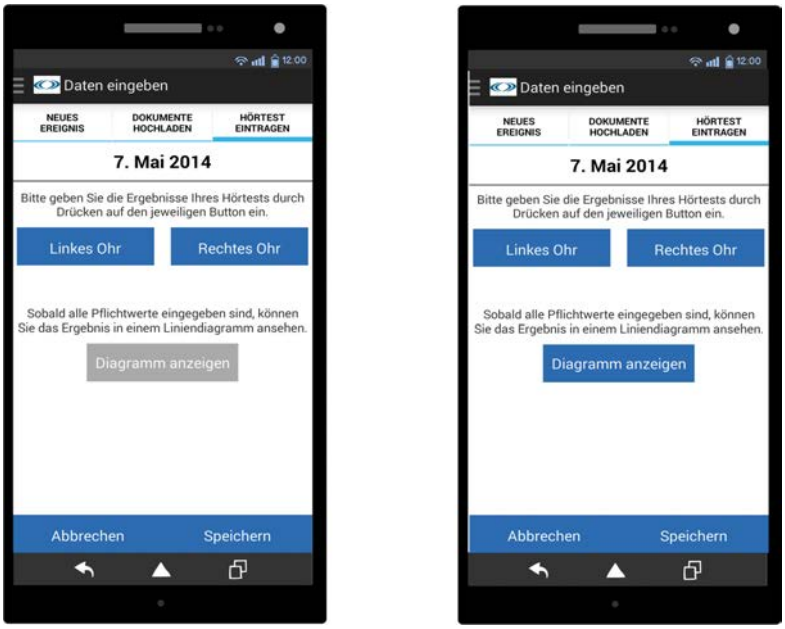


Abbildung 5.33: Das Tab *Hörtest Eintragen* zum Eintragen und Visualisieren von Hörtestergebnissen

Im Folgenden wird nun zunächst das Layout, das der Eingabemaske zugrunde liegt, vorgestellt. Anschließend werden einzelne Anforderungen angesprochen, die es bei der Implementierung der Eingabemaske zu beachten gilt.

Unter den „Standardfrequenzen“ werden alle Frequenzen zusammengefasst, die in jedem Hörtest geprüft werden. „Erweiterte Frequenzen“ werden getestet, wenn sich der Hörverlust in höheren Frequenzen des Hörbereichs befindet. In seltenen Fällen können zur Messung auch Frequenzen herangezogen werden, die den Hörverlust eines Patienten noch detaillierter bestimmen lassen. Diese werden der Gruppe „Eigene Frequenzen“ zugeordnet. Tabelle 5.5 stellt die drei Frequenzgruppen nochmals übersichtlich dar.

Frequenzgruppe	**Frequenzspektrum**
Standardfrequenzen	zwischen 125 Hz und 8000 Hz
Erweiterte Frequenzen	zwischen 10000 Hz und 18000 Hz
Eigene Frequenzen	beliebig

Tabelle 5.5: Überblick über die Frequenzgruppen in einem Hörtest

Um alle drei Frequenzgruppen auf einer Seite der App darstellen zu können, wird eine ExpandableListView verwendet. Diese spezielle View ist in Abbildung 5.34 links dargestellt und kann im Vergleich zu einer ListView Daten in zwei Ebenen darstellen. Die Namen der Frequenzgruppen befinden sich dabei in der oberen Ebene. Aktiviert der Nutzer den Schalter auf der rechten Seite einer Frequenzgruppe, wird diese aufgeklappt und der Nutzer kann die entsprechenden Ergebnisse in der unteren Ebene eintragen (Abbildung 5.34 rechts). So können auf einer Seite der App alle Messwerte eines Ohrs eingegeben werden.

Wie bei der ListView bestehen auch die einzelnen Listenelemente der ExpandableListView aus Views. Da die ExpandableListView allerdings zwei Ebenen enthält, muss für jede dieser Ebenen getrennt implementiert werden, wie die jeweiligen Listenelemente dargestellt werden sollen. Aus diesem Grund erbt der Adapter von der Klasse *BaseExpandableListAdapter*, die die Adapter-Methoden für beide Ebenen jeweils getrennt zur Verfügung stellt.

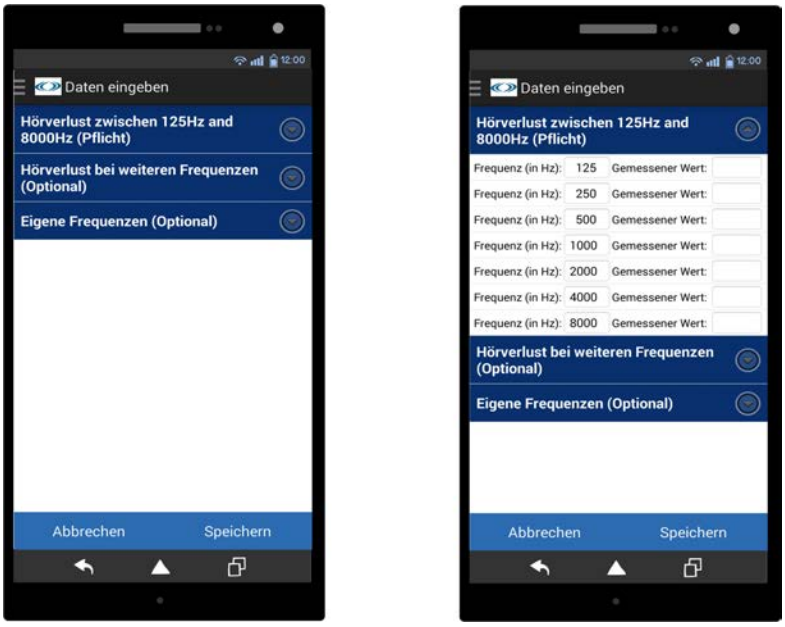


Abbildung 5.34: ExpandableListView im geschlossenen (links) und im offenen Zustand (rechts)

Die Methode *getChildView* beschreibt dabei, wie die Views der unteren Ebene auszusehen haben. Für die obere Ebene übernimmt *getGroupView* diese Aufgabe. Da der Vorgang an sich jedocht der gleiche ist wie bei einer ListView, wird auf eine weitere Beschreibung an dieser Stelle verzichtet und auf den Abschnitt Sliding-Menü verwiesen.

Als nächstes werden nun einzelne Anforderungen aufgelistet, die an die Eingabe von Hörtestergebnissen geknüpft sind. Dazu wird jede Anforderung zunächst beschrieben. Anhand eines Code-Ausschnitts wird dann gezeigt, wie die Anforderung umgesetzt ist.

1. **Editieren von Frequenzen**
   Da Frequenzen in den Gruppen „Standardfrequenzen“ und „Erweiterte Frequenzen“ fest vorgegeben sind, dürfen diese nicht geändert werden. Die Methode *getChildView* überprüft daher die Frequenzgruppe und lässt die Bearbeitung von Frequenzen nur zu, wenn das Element in der Gruppe der „Eigenen Frequenzen“ enthalten ist (Zeile 3 & 4 in Listing 5.20).

Listing 5.20: Prüfung, ob Frequenz editiert werden kann oder nicht

```
1 // only EditTexts in section "custom frequencies'
2 // should be editable
3 if( groupPosition!=2 )
4    holder.editFrequencies.setEnabled(false);
5 else
6    holder.editFrequencies.setEnabled(true);
```

2. **Direktes Speichern der Eingabewerte**

   Zum Speichern der Daten kommen zwei Zeitpunkte in Frage. Entweder direkt nach jeder Eingabe oder wenn die Eingabemaske geschlossen wird. In letzterem Fall muss jedes Eingabefeld einzeln aufgerufen werden. Dies geschieht mit der Methode *findViewById*. Wie bereits im Abschnitt ListView erwähnt, wirkt sich ein häufiger Aufruf dieser Methode negativ auf die Performance aus. Aus diesem Grund werden die eingegebenen Daten immer direkt nach der Eingabe gespeichert. Zu diesem Zweck wird ein TextWatcher verwendet, dessen Implementierung in Listing 5.21 dargestellt ist. Dieser Listener ist um die Positionsangaben *groupPosition* und *childPosition* erweitert, sodass bei jeder Eingabe klar ist, an welcher Position der Ergebnisliste *inputList* der eingegebene Wert gespeichert werden muss (Zeile 11 & 12). Außerdem wird dem Listener das Flag *isFrequency* übergeben, das angibt, ob es sich bei dem eingegeben Wert um eine zusätzlich eingetragene Frequenz oder um ein Hörtestergebnis handelt (Zeile 4). Zusätzliche Frequenzen werden in der Liste *childrenList* gespeichert.

Listing 5.21: TextWatcher zur direkten Speicherung der Messwerte

```
1 public void onTextChanged(CharSequence s, int start,
2                                int before, int count) {
3    try{
4       if( isFrequency ){
5       childrenList.get(groupPosition).
6                    set(childPosition,
7                    s.toString().trim()); }
```

```
8         else{
9         float valueAsFloat = Float.parseFloat(
10                              s.toString().trim());
11        inputList.get(groupPosition)
12                  .set(childPosition,
13                  String.format(Locale.US,
14                              "%.2f", valueAsFloat));
15        }
16    }
17    catch(NullPointerException npe){
18        inputList.get(groupPosition).
19                   set(childPosition, "");
20        Log.e("InputValuesForChart",
21                    "NullPointerException");
22    }
23    catch(NumberFormatException nfe){
24        inputList.get(groupPosition).
25                   set(childPosition, "");
26        Log.e("InputValuesForChart",
27                    "NumberFormatException");
28    }
29 }
```

3. **Speichern der Eingabewerte als String**
   Wie beispielsweise die Zeilen 13 und 14 in Listing 5.21 zeigen, werden alle Eingaben als String gespeichert. Dies ist darin begründet, da die Funktion zum Auslesen eines Textfeldes in Android stets einen String zurückgibt. Da die eingegebenen Werte außerdem nur in zwei Fällen in den Datentyp Float konvertiert werden müssen, lohnt eine Speicherung der Werte als Float nicht.

4. **Überprüfen der Eingaben**

   Sobald die Eingabemaske verlassen wird, werden die eingegebenen Werte auf Korrekheit geprüft. Wie Listing 5.22 zeigt, werden dazu alle Eingabewerte, die in der Liste *inputList* gespeichert sind, auf folgende zwei Bedingungen geprüft

   - Alle Eingabefelder der Standardfrequenzen müssen einen Wert enthalten (Zeile 5 - 8)
   - Die Werte aller Eingabefelder müssen sich im Wertebereich [0;100] befinden (z.B. Zeile 20)

   Ist der Wert ungültig oder ist das Eingabefeld einer Standardfrequenz leer, wird das Flag *isFilled* auf false gesetzt (Zeile 6). Dadurch wird verhindert, dass die Eingabemaske geschlossen wird. Zusätzlich dazu wird die fehlerhafte Position durch Setzen eines Flags in der Liste *checkValues* markiert (Zeile 7).
   Jedes Mal, wenn das Flag *isFilled* nicht gesetzt ist, wird der Adapter über die geänderten Flags in der Liste *checkValues* informiert (Zeile 51). Die einzelnen Views der ExpandableListView werden daraufhin neu geladen und zeigen abhängig davon, ob das Flag an der entsprechenden Position in *checkValues* gesetzt ist, einen Warnhinweis an.

   Listing 5.22: Ausschnitt aus der Listener-Methode, die alle eingegebenen Messwerte auf Korrektheit prüft

```
1  boolean isFilled = true;
2
3  for(int i = 0; i< inputList.get(0).size(); i++){
4     String val = inputList.get(0).get(i);
5     if( val.length()==0 ){
6        isFilled=false;
7        checkValues.get(0).set(i, false);
8     }
9     else if ( Float.valueOf(val)>100 ){
10       isFilled=false;
```

```
11          checkValues.get(0).set(i, false);
12      }
13      else{
14          checkValues.get(0).set(i, true);
15      }
16  }
17
18  for(int j = 0; j < inputList.get(1).size(); j++){
19      String val = inputList.get(1).get(j);
20      if( val.length()>0 && Float.valueOf(val)>100 ){
21          isFilled=false;
22          checkValues.get(1).set(j, false);
23      }
24      else{
25          checkValues.get(1).set(j, true);
26      }
27  }
28
29
30  for(int j = 0; j < inputList.get(2).size(); j++){
31      String freq = childrenList.get(2).get(j);
32      String val = inputList.get(2).get(j);
33      if( (freq.length()==0 && val.length()>0) ||
34                  (freq.length()>0 &&val.length()==0) ){
35          isFilled = false;
36          checkValues.get(2).set(j, false);
37      }
38      else if( val.length()>0 && Float.valueOf(val)>100 ){
39          isFilled=false;
40          checkValues.get(2).set(j, false);
41      }
```

```
42      else{
43          checkValues.get(2).set(j, true);
44      }
45  }
46
47  if(isFilled){
48      returnToEditAudiometry(false);
49  }
50  else{
51      caa.notifyDataSetChanged();
52      Toast.makeText(getActivity(),
53                      "Bitte achten Sie darauf,dass
54                      alle Pflichtfelder ausgeföllt
55                      sind und dass kein gemessener
56                      Wert grösser als 100 sein darf!",
57                      Toast.LENGTH_LONG).show();
58  }
```

5.2.7 Visualisierung von Hörtestergebnissen

Sobald die Ergebnisse eines Hörtests für beide Ohren eingetragen sind, werden die Angaben auf Vollständigkeit geprüft. Ein Hörtest kann nämlich nur dargestellt bzw. gespeichert werden, wenn folgende Voraussetzungen erfüllt sind:

1. Alle Pflichtfelder sind ausgefüllt
2. Alle eingegebenen Werte liegen im Wertebereiche [0; 100]
3. Zu jeder Frequenz sind in beiden Ohren entweder Werte eingetragen oder keine Werte eingetragen.

Während die Überprüfung der ersten beiden Punkte bereits beim Schließen der Eingabemaske erfolgt, wird Voraussetzung (3) in der Methode *isEqual* der Klasse *Audiometry* verifiziert. Listing 5.23 zeigt diese Methode, die in zwei Abschnitte unterteilt ist.

In den Zeilen 2 - 12 wird nochmals Voraussetzung (1) geprüft. So wird sichergestellt, dass ein neues bzw. unvollständiges *Audiometry*-Objekt nicht visualisiert werden kann. Ist Voraussetzung (1) gültig, werden die Zeilen 14 - 28 ausgeführt, in denen Voraussetzung (3) überprüft wird. Erst wenn auch dieser Test erfolgreich ist, kann der Hörtest in einem Liniendiagramm dargestellt werden.

Listing 5.23: Die Methode *isEqual*, die überprüft, ob ein Hörtest dargestellt werden kann

```
1  public boolean isEqual() {
2     if( valuesLeft.size()==valuesRight.size() ){
3        if( valuesLeft.get(0).size()==valuesRight.get(0)
4                                                .size() ){
5           for(int i = 0; i < valuesLeft.get(0).size(); i++){
6              int lengthLeft = valuesLeft.get(0).get(i)
7                                               .length();
8              int lengthRight = valuesRight.get(0).get(i)
9                                                 .length();
10             if (lengthLeft * lengthRight == 0) return false;
11          }
12       } else return false;
13
14       for(int i = 1; i < valuesLeft.size(); i++){
15          if( valuesLeft.get(i).size()==valuesRight.get(i)
16                                                  .size() ){
17             for(int j = 0; j< valuesLeft.get(i).size();j++){
18                int lengthLeft = valuesLeft.get(i).get(j)
19                                                .length();
20                int lengthRight = valuesRight.get(i).get(j)
21                                                  .length();
22                if ((lengthLeft > 0 && lengthRight == 0) ||
23                   (lengthLeft == 0 && lengthRight > 0))
24                return false;
```

```
25                }
26            }
27            else return false;
28        }
29    }
30    else return false;
31    return true;
32 }
```

Die Darstellung des Hörtest wird durch die Klasse *AudiometryShowChart* realisiert, die bewusst als Activity und nicht als Fragment implementiert ist. Zum einen soll das Liniendiagramm den kompletten Bildschirm des Smartphones ausnutzen. Die Klasse *MainActivity* hätte dazu so verändert werden müssen, dass nur dieses eine Fragment im Vollbildmodus ausgeführt wird. Diese spezielle Darstellungsform hat Google in Android allerdings nicht vorgesehen. Zum anderen wird *AudiometryShowChart* nur für die Darstellung von Hörtestergebnissen verwendet. Damit grenzt sie sich hinsichtlich ihrer Aufgabe so stark von der *MainActivity* ab, dass eine Auslagerung als eigenständige Activity sinnvoll erscheint.

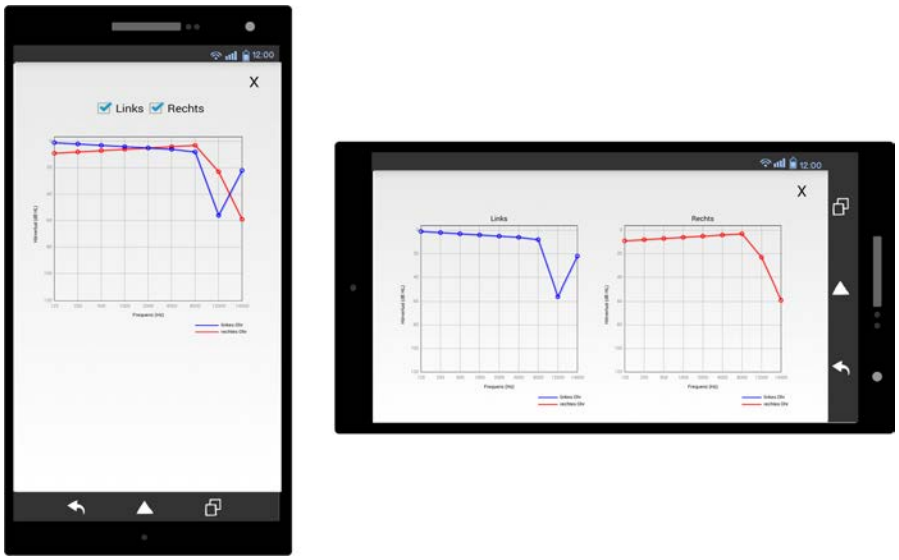


Abbildung 5.35: Liniendiagramm eines Hörtests im Hochformat und im Querformat

Wie bereits am Anfang dieses erwähnt wurde, können Hörtestergebnisse auch im Querformat betrachtet werden. Der Vorteil dabei ist, dass die Ergebnisse beider Ohren im Querformat nebeneinander dargestellt werden können. Abbildung 5.35 zeigt das Liniendiagramm eines Hörtests jeweils im Hochformat und im Querformat.

Im Code umgesetzt ist diese variable Darstellungsform durch die Methode *setupLayout*, die in Listing 5.24 zu sehen ist. Darin wird in den Zeilen 10 und 11 zunächst die Lage des Smartphones bestimmt. Wird es im Querformat gehalten, werden zwei Diagramme nebeneinander dargestellt (Zeile 14 ff.). Das linke Diagramm zeigt dabei das Ergebnis des linken Ohrs, das rechte Diagramm entsprechend das Ergebnis des rechten Ohrs (Zeile 16 & 18).

Listing 5.24: Die Methode *setupLayout*

```
1  private void setupLayout(){
2      LinearLayout layoutForVerticalAlignedCharts =
3                      (LinearLayout) findViewById(
4                       R.id.LinearLayoutLineChart);
5      LinearLayout.LayoutParams param =
6          new LinearLayout.LayoutParams(
7                      LayoutParams.MATCH_PARENT,
8                      LayoutParams.WRAP_CONTENT, 1);
9
10     int orientation = getResources().getConfiguration()
11                                      .orientation;
12     Log.d(getClass().getName(), "Orientation:" +orientation);
13     //Landscape
14     if( orientation == 2){
15         LineChartView lcv = new LineChartView(this);
16         lcv.setData(new int[]{Color.BLUE},"Links");
17         LineChartView lcv1 = new LineChartView(this);
18         lcv1.setData(new int[]{Color.RED},"Rechts");
19
20         LinearLayout layoutForHorizontalAlignedCharts =
```

```
21            new LinearLayout(this);
22        layoutForHorizontalAlignedCharts
23                              .setGravity(Gravity.CENTER);
24        layoutForHorizontalAlignedCharts
25                              .addView(lcv, param);
26        layoutForHorizontalAlignedCharts
27                              .addView(lcv1, param);
28        layoutForVerticalAlignedCharts
29                    .addView(layoutForHorizontalAlignedCharts);
30    }
```

Im Hochformat wird der Verlauf beider Ergebnisse in einem Liniendiagramm dargestellt (Listing 5.25, Zeile 4). Wie auf Abbildung 5.35 links zu erkennen ist, ist es jedoch auch möglich, einzelne Graphen ein bzw. auszublenden. Hierzu sind in den Zeilen 6 – 11 zwei Checkboxes implementiert, die nach jedem Klick eine erneute Zeichnung des Diagramms veranlassen.

Listing 5.25: Implementierung von Checkboxen, um einzelne Graphen ein- und auszublenden

```
1    //Portrait
2    else{
3         final LineChartView lcv = new LineChartView(this);
4         lcv.setData(new int[]{Color.RED, Color.BLUE},"");
5
6         CheckBox leftEar = new CheckBox(this);
7         leftEar.setText("Links");
8         leftEar.setChecked(true);
9         CheckBox rightEar = new CheckBox(this);
10        rightEar.setText("Rechts");
11        rightEar.setChecked(true);
12
```

```
13          CustomCheckedChangeListener listener =
14              new CustomCheckedChangeListener(lcv,
15                                              leftEar, rightEar);
16          leftEar.setOnCheckedChangeListener(listener);
17          rightEar.setOnCheckedChangeListener(listener);
18
19          LinearLayout layoutForHorizontalAlignedCharts =
20              new LinearLayout(this);
21          layoutForHorizontalAlignedCharts
22                                      .setGravity(Gravity.CENTER);
23          layoutForHorizontalAlignedCharts.addView(leftEar);
24          layoutForHorizontalAlignedCharts.addView(rightEar);
25          layoutForVerticalAlignedCharts
26                      .addView(layoutForHorizontalAlignedCharts);
27          layoutForVerticalAlignedCharts.addView(lcv);
28      }
29  }
```

Für die Zeichnung des Liniendiagramms wurde bewusst auf die Verwendung einer vorgefertigten Diagramm-Bibliothek verzichtet. Dies hat mehrere Gründe.

- Unnötiger Speicherplatzbedarf aufgrund einer Vielzahl nicht genutzter Funktionen
- Höhere Performance der eigenen View
- Flexiblere Gestaltung der eigenen View
- Ausschließen von Fehlverhalten externer Bibliotheken nahezu unmöglich

Aus diesem Grund wird das Liniendiagramm in der inneren Klasse *LineChartView* mit Hilfe der androidinternen Graphics-Bibliothek erstellt [Gooe]. Die wichtigsten beiden Komponenten dieser Bibliothek sind die Klassen *Canvas* und *Paint*. Während mit *Canvas* bestimmt wird, was gezeichnet wird, legt *Paint* fest, wie dies gezeichnet wird.

LineChartView erbt direkt von der Klasse *View*. Dies ist notwendig, da in Android nur Komponenten dargestellt werden können, die eine Subklasse von *View* sind oder direkt von einer View erben. Der Konstruktor von *LineChartView* ist in den Zeilen 1 - 4 von Listing 5.26 zu sehen. Dieser fügt an jeder Seite ein Padding ein, sodass zwischen dem Bildschirmrand und der Beschriftung der Achsen bzw. der Legende ein gewisser Abstand eingehalten wird.

Listing 5.26: Konstruktor und wichtige Methoden der inneren Klasse *LineChartView*

```
1  public LineChartView(Context context) {
2      super(context);
3      setPadding(100,100,100,150);
4  }
5
6  private void setData(int[] color, String title ) {
7      this.graphColor = color;
8      this.title = title;
9      invalidate();
10 }
11
12 protected void onDraw(Canvas canvas) {
13     drawBackground(canvas);
14     drawGraph(canvas);
15 }
```

Listing 5.26 zeigt außerdem die Methoden *setData* (Zeile 6 – 10), *onDraw* (Zeile 12 – 15). Wie diese beiden Methoden mit der Methode *onMeasure* in Listing 5.27 zusammenhängen, wird nun im Folgenden beschrieben. Nachdem ein neues *LineChartView*-Objekt erstellt ist, wird die Methode *setData* gerufen. In dieser werden die Farbe des Graphen und die Überschrift des Diagramms definiert. Über die Länge des Arrays *graphColor* kann dabei bestimmt werden, ob das Liniendiagramm im Hochformat nur das Hörtestergebnis eines Ohres oder die Hörtestergebnisse beider Ohren anzeigen soll (Zeile 7).

Für den Fall, dass *graphColor* nur ein Element enthält, gibt *title* an, um welches Ohr es sich handelt (Zeile 8). Der Aufruf von *invalidate* in der Methode *setData* triggert implizit die Methoden *onDraw*, welche die Zeichnung der View übernimmt, und *onMeasure*, welche die Größe der View anhand der Bildschirmgröße festsetzt. Damit die View immer vollständig auf dem Bildschirm angezeigt werden kann, ist die Größe des Diagramms immer von der kürzeren Bildschirmlänge abhängig. Listing 5.27 zeigt dies in den Zeilen 11 bis 15.

Listing 5.27: Die Methode *onMeasure* zur Berechnung der Größe der View

```
1  protected void onMeasure(int widthMeasureSpec,
2                           int heightMeasureSpec) {
3      super.onMeasure(widthMeasureSpec, heightMeasureSpec);
4
5      int size = 0;
6      int width = getMeasuredWidth();
7      int height = getMeasuredHeight();
8
9      // always ensure the chart's length depends on
10     // the smaller length of the display
11     if ( width > height ) {
12           size = height;
13     } else {
14           size = width;
15     }
16     //  the form of the chart is a square
17     setMeasuredDimension(size , size);
18 }
```

Der Zeichenvorgang des Liniendiagramms wird in zwei Schritten durchgeführt. Zunächst wird der Hintergrund des Diagramms mit der Methode *drawBackground* gezeichnet. Anschließend wird der Graph mit der Methode *drawGraph* in das Diagramm eingezeichnet. Wie der Ablauf der Methoden im Einzelnen ist, wird nun im Folgenden beschrieben.

Wie Abbildung 5.36 zeigt, wird in *drawBackground* zunächst der Rahmen des Diagramms gezeichnet. Nachdem dieser erstellt ist, werden in den Schritten (2) und (3) die horizontalen und vertikalen Linien in das Diagramm eingetragen. Zusätzlich dazu werden die entsprechenden Achsen beschriftet. Während die Y-Achse den prozentualen Hörverlust angibt, beschreibt die X-Achse die Frequenz, auf welcher sich der Hörverlust befindet. Damit diese Zuordnung dem Nutzer jederzeit bewusst ist, werden dann an der linken und an der unteren Seite des Diagramms entsprechende Textfelder platziert (4). Zuletzt wird noch die Legende eingezeichnet (5).

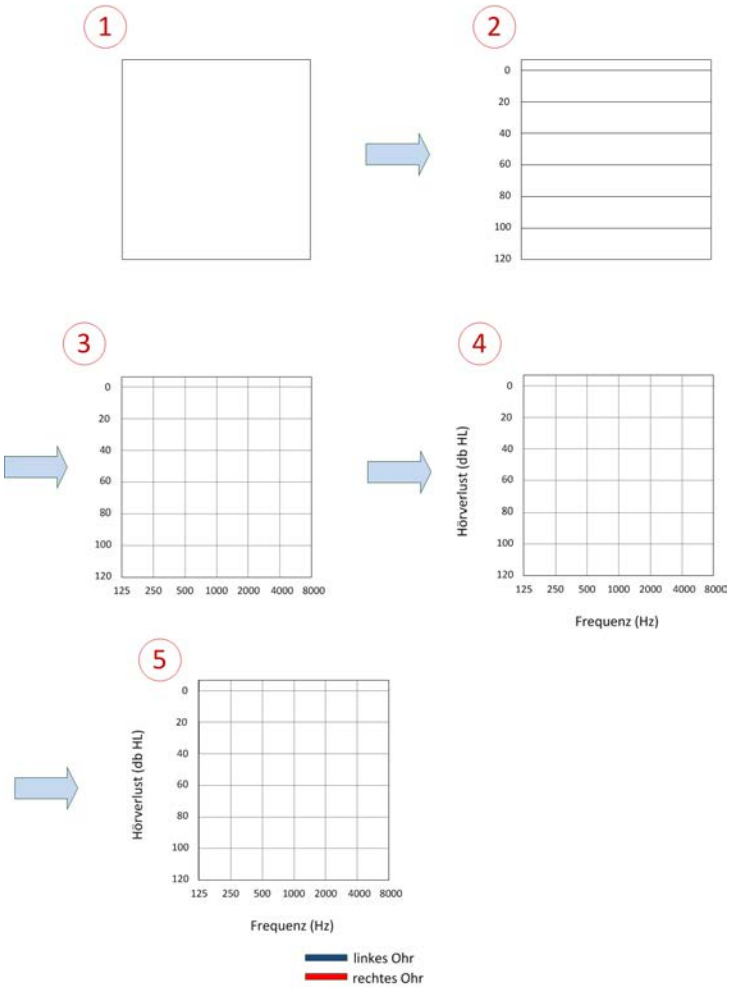


Abbildung 5.36: Ablauf des Zeichenvorgangs des Liniendiagramms

Nachdem nun bekannt ist, wie *drawBackground* arbeitet, werden nun einige wichtige Implentierungsdetails anhand Listing 5.28 vorgestellt. Jedes graphische Objekt besitzt bestimmte Eigenschaften, die das Aussehen näher beschreiben. Wenn ein Objekt in einer View eingezeichnet werden soll, müssen diese Eigenschaften ebenfalls definiert werden. Dies geschieht mit Hilfe eines *Paint*-Objekts. Am Beispiel des Rahmens des Diagramms ist ein solches Objekt in den Zeilen 2 - 5 in Listing 5.28 dargestellt. Dieser Rahmen besteht aus einer schwarzen Linie, die 2 Pixel breit ist. Erst wenn diese Eigenschaften definiert sind, kann das graphische Objekt mit Hilfe eines *Canvas*-Objekts in die View eingezeichnet werden. Auch die Eigenschaften von Texten, die in das Diagramm eingezeichnet werden sollen, müssen mit einem *Paint*-Objekt beschrieben werden. So kann zum Beispiel mit der Methode *setAntiAlias* die Kantenglättung aktiviert werden, um einen Text sauber darstellen zu können (Zeile 25).

Listing 5.28: Ausschnitte der Methode *drawBackground*

```
1  // draw frame of the chart
2  paint.setStyle(Style.STROKE);
3  paint.setStrokeWidth(2);
4  paint.setColor(Color.BLACK);
5  paint.setAntiAlias(false);
6  final int space = 20;
7  canvas.drawRect(getPaddingLeft(),getPaddingTop()-space,
8                  getWidth()-getPaddingRight(),
9                  getHeight()-getPaddingBottom(), paint);
10
11 // draw horizontal lines and scale of the chart
12 paint.setStyle(Style.FILL);
13 paint.setTextAlign(Align.RIGHT);
14 paint.setTextSize(20);
15 paint.setColor(Color.GRAY);
16 paint.setStrokeWidth(1);
17 for (int y = maxValueY; y >=0; y -= 20) {
18      final float yPos = calculateYPos(y);
```

```
19
20      paint.setAntiAlias(false);
21      canvas.drawLine(getPaddingLeft(), yPos,
22                      getWidth()-getPaddingRight(),
23                                      yPos, paint);
24
25      paint.setAntiAlias(true);
26      canvas.drawText(String.valueOf(y), getPaddingLeft()-10,
27                                              yPos+5, paint);
28  }
```

Neben der der X-Achse wird in Schritt (4) auch die Y-Achse beschriftet. Um den Text dabei in vertikaler Form darstellen zu können, muss mit der Methode *save* zunächst die aktuelle Position des *Canvas*-Objektes gespeichert (Listing 5.29, Zeile 9). Daraufhin wird das *Canvas*-Objekt um 90 Grad im Uhrzeigersinn gedreht, sodass die zu beschriftende Achse nach oben zeigt. Anschließend kann der Text oberhalb dieser Achse eingezeichnet werden (Zeile 11 & 12).

Listing 5.29: Weiterer Ausschnitt der Methode *drawBackground*

```
1  // draw labels
2  paint.setColor(Color.BLACK);
3  paint.setTextSize(30);
4  canvas.drawText(title, getWidth()/2,
5                  getPaddingTop()-space-20, paint);
6  paint.setTextSize(20);
7  canvas.drawText("Frequenz (Hz)", getWidth()/2,
8                   getHeight()-getPaddingBottom()+70, paint);
9  canvas.save();
10 canvas.rotate(-90,getWidth()/2, getHeight()/2);
11 canvas.drawText("Hörverlust (dB HL)", (getWidth()+space)/2,
12                                  getPaddingTop()-70, paint);
13 canvas.restore();
```

Der Befehl *restore* in Zeile 13 stellt die mit *save* gespeicherte Position des *Canvas*-Objekts schließlich wieder her, ohne den eingezeichneten Text zu löschen. Zum besseren Verständnis ist dieser Vorgang ist in Abbildung 5.37 noch einmal dargestellt.

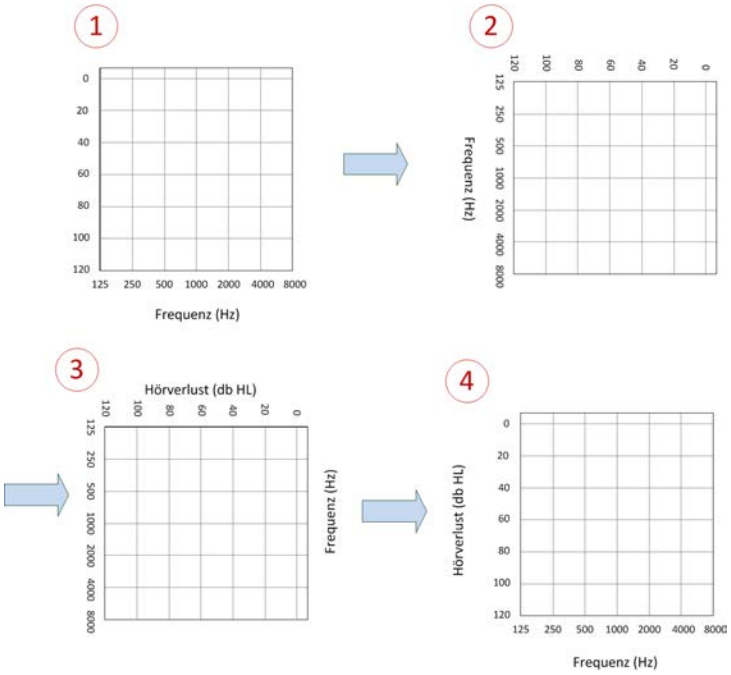


Abbildung 5.37: Darstellung eines vertikalen Textes

Nachdem der Hintergrund des Diagramms fertiggestellt ist, werden in der Methode *drawGraph* die Hörtestergebnisse des linken bzw. rechten Ohres in das Diagramm eingezeichnet. Dazu wird ein *Path*-Objekt verwendet, das in Zeile 13 in Listing 5.30 einsehbar ist. Zunächst wird der Startpunkt des Pfades festgelegt (Zeile 14). Nacheinander werden dann Punkte zum Pfad hinzugefügt, deren erste Komponente den gemessenen Hörverlust und deren zweite Komponente die korrespondierende Frequenz angibt. Wie die Zeilen 22 und 23 zeigen, werden dabei jeweils zwei Punkte durch eine gerade Linie miteinander verbunden. Um die eingegebenen Messdaten hervorzuheben, werden diese in den Zeilen 24 - 26 zusätzlich durch einen Kreis markiert. Abschließend wird der Pfad in Zeile 31 durch ein *Canvas*-Objekt in das Diagramm eingezeichnet wird. Die Verwendung eines Pfad-Objekts erleichtert das Einzeichnen des Graphen deutlich.

Alternativ wäre es allerdings auch möglich, zunächst die Datenpunkte einzeln in das Diagramm einzuzeichnen und diese dann in einem weiteren Schritt durch Linien zu verbinden.

Listing 5.30: Die Methode *drawGraph*, die Graphen in das Liniendiagramm einzeichnet

```
1  private void drawGraph(Canvas canvas) {
2
3     for(int x = 0; x < graphColor.length; x++){
4
5       int ear = graphColor[x]==Color.BLUE ? 0 : 1;
6
7       paint.setStyle(Style.STROKE);
8       paint.setStrokeWidth(4);
9       paint.setColor(graphColor[x]);
10      paint.setAntiAlias(true);
11      paint.setShadowLayer(4, 2, 2, 0x80000000);
12
13      Path path = new Path();
14      path.moveTo(calculateXPos(0),calculateYPos(
15                        Float.valueOf(result.get(ear).get(0))));
16      canvas.drawCircle(calculateXPos(0), calculateYPos(
17                        Float.valueOf(result.get(ear).get(0))),
18                                                  8, paint);
19
20      for (int i = 1; i < result.get(ear).size(); i++) {
21         try{
22            path.lineTo(calculateXPos(i),calculateYPos(
23                        Float.valueOf(result.get(ear).get(i))));
24            canvas.drawCircle(calculateXPos(i),calculateYPos(
25                        Float.valueOf(result.get(ear).get(i))),
26                                                  8, paint);
27         }
```

```
28            catch(NumberFormatException nfe){ }
29            catch(NullPointerException npe){}
30        }
31        canvas.drawPath(path, paint);
32        paint.setShadowLayer(0, 0, 0, 0);
33        }
34    }
```

Abgeschlossen werden soll dieser Abschnitt schließlich mit den Methoden *calculateXPos* und *calculateYPos*, auf die bislang nicht eingegangen wurde. Aufgrund des Paddings, das zwischen dem Bildschirmrand und dem Liniendiagramm einen gewissen Abstand generiert, würde das Liniendiagramm in der View nicht zentral angezeigt werden. Um diesen Fehler zu korrigieren, skalieren die beiden Funktionen alle Datenpunkte im Liniendiagramm derart, dass das Liniendiagramm zum einen zentral dargestellt wird und zum anderen die ganze Fläche der View ausfüllt.

5.2.8 Therapievorschlag

Das Hauptziel dieser App ist die Generierung von mindestens einem Therapievorschlag. Hierzu müssen drei Voraussetzungen erfüllt sein:

1. Einpflegen von Regeln und Therapievorschlägen in der Webanwendung
2. Abruf der Regeln und Therapievorschläge durch die App
3. Ausreichend große Datenmenge, sodass mindestens eine Regel darauf anwendbar ist

Die Voraussetzungen (1) und (2) werden für dieses Kapitel als gegeben betrachtet. Im Folgenden werden nun zunächst die Methoden und Strukturen erläutert, die zur Bestimmung eines Therapievorschlags eingesetzt werden. Anschließend wird erklärt, warum die Wahl gerade auf diese Strukturen gefallen ist und welche Alternative es gibt.

Die Bestimmung von Therapievorschlägen erfolgt in der Methode *checkDisposability* der Klasse *Recommendation*, die Therapievorschlag-Objekte verwaltet. Jedes *Recommendation*-Objekt besitzt folgende Felder:

- recommendation_id: int - ID des Therapievorschlags
- title: String - Titel des Therapievorschlags
- description: String - detaillierte Beschreibung des Therapievorschlags
- contradicted_recommendation: int - ID eines Therapievorschlags, der im Widerspruch zu jenem in recommendation_id steht

Zu Beginn von checkDisposability werden in den Zeilen 7 - 9 von Listing 5.31 zunächst alle *Rule*-Objekte aus der Datenbank in der Liste *rulesList* gespeichert. Ein *Rule*-Objekt besteht dabei aus folgenden Feldern:

- rule_id: int - ID der Regel
- recommendation_id: int - Therapievorschlag, der mit dieser Regel verbunden ist
- table: String - Datenbanktabelle, die durchsucht werden soll
- column: String - Spalte der Tabelle, die durchsucht werden soll
- filter: String - Operatoren wie „=“, „<“, oder „LIKE“, „IN“
- value: String - Suchwert
- connected_rule: int - ID einer Regel, die mit rule_id verknüpft ist
- connection_type: String - Art der Verknüpfung: „AND“, „OR“, „NAND“, „NOR“

Listing 5.31: Beginn der Methode *checkDisposability*

```
1 public static ArrayList<Recommendation>
2                         checkDisposability(Context context){
3    RulesDatabaseHelper rulesDbHelper =
4                         new RulesDatabaseHelper(context);
5    SQLiteDatabase rulesDatabase =
6                         rulesDbHelper.getReadableDatabase();
```

```
7    ArrayList<Rule> rulesList =
8                            rulesDbHelper.getAllRuleObjects(
9                                                rulesDatabase,context);
```

Aus den Eigenschaften des *Rule*-Objekts geht hervor, dass ein *Rule*-Objekt R1 mit einem anderen *Rule*-Objekt R2 verknüpft werden kann. Dazu muss, wie Abbildung 5.38 zeigt, die ID von R2 im Feld *connected_rule* von R1 eingetragen sein. Außerdem muss die Art der Verknüpfung im Feld *connection_type* von R1 angegeben sein.

R1

rule_id	recommendation_id	table	column	filter	value	connected_rule	connection_type
4711	1234	audiometries	values_left	>	40	4712	AND

R2

rule_id	recommendation_id	table	column	filter	value	connected_rule	connection_type
4712	1234	audiometries	values_left	<	40	null	null

Abbildung 5.38: Zwei miteinander verknüpfte *Rule*-Objekte

Für jede Regel, die mit R1 verknüpft werden soll, muss somit ein neues *Rule*-Objekt mit gleicher *rule_id*, aber unterschiedlicher *connected_rule* erstellt werden. Aus diesem Grund erfolgt die Speicherung der *Rule*-Objekte in der Liste *rulesList* nach aufsteigender *rule_id*. Damit ist garantiert, dass miteinander verknüpfte Regeln immer hintereinander stehen. Die Auswertung verknüpfter Regeln bedarf somit keiner aufwändigen Suche in *rulesList* und erfolgt effizient in linearer Zeit.

Der Prozess, der Therapievorschläge auf Basis der eingegebenen Daten generiert, ist in 3 Abschnitte unterteilt, die nun im Folgenden vorgestellt werden:

1. Überprüfung der Gültigkeit aller Regeln
2. Erstellen eines Gesamtwahrheitswertes für jede Regel
3. Entfernen von widersprüchlichen Therapievorschlägen aus der Ergebnisliste

1. Überprüfung der Gültigkeit aller Regeln (Listing 5.32)

Zunächst wird jede Regel in *rulesList* für sich auf Gültigkeit untersucht. Regeln mit gleicher *rule_id* werden dabei nur einmal geprüft (Zeile 57 - 61). Denn wie bereits erklärt, unterscheiden sich zwei Regeln mit gleicher *rule_id* nur im Feld *connected_rule*, das keinen Einfluss auf die Gültigkeit einer Regel hat. Anschließend wird aus den Feldern *table*, *column*, *filter* und *value* jedes *Rule*-Objekts eine SQL-Query geformt, die auf der in *table* gespeicherten Tabelle ausgeführt wird. Abbildung 5.39 stellt diesen Vorgang noch einmal am Beispiel einer Regel dar, die auf Hörtests angewandt wird.

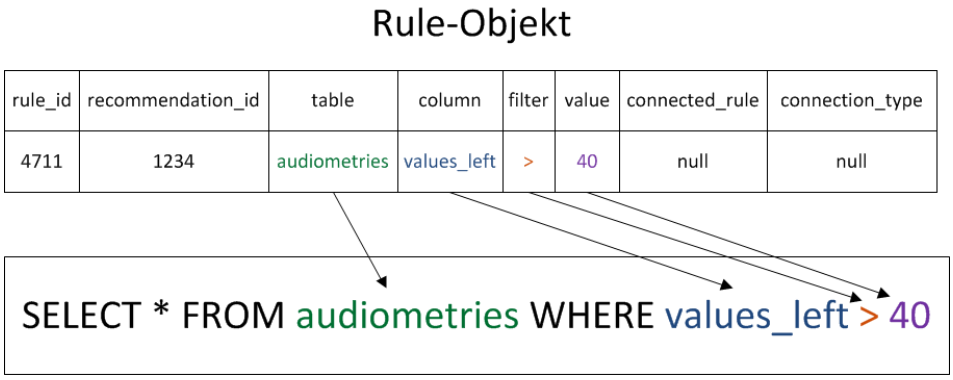


Abbildung 5.39: Erstellen eines SQL-Statements aus einem *Rule*-Objekt

Jede Query ist stets so geformt, dass ihr Ergebnis einen booleschen Wahrheitswert zurückgibt. Liefert die Query mindestens einen Datensatz, so ist Regel gültig. Der Wahrheitswert jeder Regel wird im SparseBooleanArray *rulesCheck* in der Form <rule_id, isFullfilled> gespeichert (Listing 5.32, Zeile 53). In einem SparseBooleanArray können wie in einer HashMap <Key,Value>-Paare gespeichert werden. Die Performance eines *SparseBooleanArrays* ist allerdings höher, da bei den Keys kein Autoboxing durchgeführt wird und Values beim Mapping nicht auf ein Entry-Objekt angewiesen sind [Gooi].

Listing 5.32: Überprüfung der Gültigkeit jeder Regel

```
1 if( rulesList.isEmpty() )
2     return new ArrayList<Recommendation>();
3 else{
4    SparseBooleanArray rulesCheck = new SparseBooleanArray();
```

```
5     // first loop through recommender list for determining
6     // every rules logical value
7     for(int i = 0; i < rulesList.size(); i++){
8         Rule r = rulesList.get(i);
9         boolean isFullfilled = false;
10
11        SQLiteDatabase db = null;
12        if( r.table.equals(
13                  AudiometryDatabaseHelper.TABLE_AUDIOMETRY) ){
14            AudiometryDatabaseHelper helper = new
15                                AudiometryDatabaseHelper(context);
16            db = helper.getReadableDatabase();
17            String query = "SELECT 1 FROM " + r.table +
18                           " WHERE CAST(" + r.column +
19                           " AS REAL) " + r.filter + " " +
20                           Float.parseFloat(r.value);
21            isFullfilled = helper.checkContent(db, query);
22            db.close();
23            helper.close();
24         }
25         else if( r.table.equals(
26                         AnswersDatabaseHelper.TABLE_ANSWERS) ){
27            AnswersDatabaseHelper helper = new
28                                 AnswersDatabaseHelper(context);
29            db = helper.getReadableDatabase();
30            String query = "SELECT 1 FROM " + r.table +
31                           " WHERE " + r.column + " " +
32                           r.filter + " " + r.value;
33            isFullfilled = helper.checkContent(db, query);
34            db.close();
35            helper.close();
```

```
36          }
37          else if( r.table.equals(
38                      CalendarDatabaseHelper.TABLE_CALENDAR) ){
39             CalendarDatabaseHelper helper = new
40                                 CalendarDatabaseHelper(context);
41             db = helper.getReadableDatabase();
42             String query = "SELECT 1 FROM " + r.table +
43                            " WHERE " + r.column + " " +
44                            r.filter + " " + r.value;
45             isFullfilled = helper.checkContent(db, query);
46             db.close();
47             helper.close();
48          }
49          else{
50          Log.e("ERROR IN RECOMMENDER.JAVA", "undefined table
51                                    name or table name is null");
52          }
53          rulesCheck.append(r.rule_id, isFullfilled);
54
55          // save unnecessary database queries for
56          // multiple connected rules
57          while( i+1 < rulesList.size() &&
58                 rulesList.get(i).rule_id==rulesList.
59                                              .get(i+1).rule_id )
60                 i++;
61          }
```

Wird eine Query auf die Tabelle *audiometries* angesetzt, so müssen die Werte der Spalten in den Datentyp REAL gecastet werden (Zeile 17 - 20). Grund dafür ist, dass Frequenzen in der App ohne Dezimalpunkt, die Messwerte allerdings mit Dezimalpunkt dargestellt werden.

Da Regeln aber sowohl auf Frequenzen als auch auf Messwerte in der Tabelle angewandt werden dürfen, müsste für jede der beiden Spalten eine eigene Query erstellt werden. Diese zusätzliche Abfrage erübrigt sich durch den erwähnten Cast.

2. Erstellen eines Gesamtwahrheitswertes für jede Regel (Listing 5.33)

Da Regeln miteinander verknüpft werden können, wird im zweiten Abschnitt für jede Regel ein Gesamtwahrheitswert ermittelt. Hierzu sei nochmals erwähnt, dass miteinander verknüpfte Regeln stets als Block in der Liste *rulesList* gespeichert sind. Die while-Schleife, mit der diese Blöcke durchlaufen werden, ist in den Zeilen 18 - 53 zu sehen.

Listing 5.33: Erstellen eines Gesamtwahrheitswertes

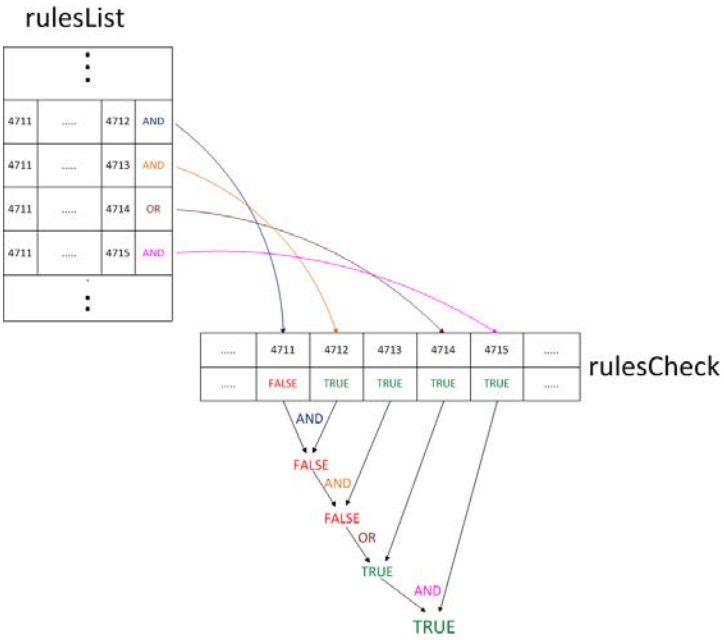
```
1   // second loop through recommender list for determining
2   // logical value of all connected rules containing same
3   // rule_id
4   boolean connectedRulesCheck = false;
5   ArrayList<Integer> recommendationList = new
6                                           ArrayList<Integer>();
7   for(int i = 0; i < rulesList.size(); i++){
8
9      int recommendation_id = rulesList.get(i)
10                                        .recommendation_id;
11
12     // recommendation could already been added due to an
13     // other rule that already come true
14     if( !(recommendationList.contains(recommendation_id)) ){
15        connectedRulesCheck = rulesCheck.get(rulesList.get(i)
16                                                .rule_id);
17
18        while( rulesList.get(i).connected_rule!=-1 ){
19           if(rulesList.get(i).connection_type.equals(
```

```
20                                                    "AND")){
21          connectedRulesCheck = connectedRulesCheck &&
22                                rulesCheck.get(rulesList
23                                     .get(i).connected_rule);
24      }
25      else if(rulesList.get(i).connection_type.equals(
26                                                     "OR")){
27          connectedRulesCheck = connectedRulesCheck ||
28                                rulesCheck.get(rulesList
29                                .get(i).connected_rule);
30      }
31      else if(rulesList.get(i).connection_type.equals(
32                                                "AND NOT")){
33          connectedRulesCheck = !(connectedRulesCheck &&
34                               rulesCheck.get(rulesList
35                               .get(i).connected_rule));
36      }
37      else if(rulesList.get(i).connection_type.equals(
38                                                 "OR NOT")){
39          connectedRulesCheck = !(connectedRulesCheck ||
40                               rulesCheck.get(rulesList
41                               .get(i).connected_rule));
42      }
43      else{Log.e("ERROR IN RECOMMENDER.JAVA",
44              "undefined connection type or connection
45              type is null");
46      }
47
48      if( i+1 < rulesList.size() &&
49          rulesList.get(i).rule_id==rulesList.get(i+1)
50                                                .rule_id)
```

```
51                  i++;
52              else break;
53          }
54
55          if(connectedRulesCheck)
56              recommendationList.add(recommendation_id);
57      }
58      connectedRulesCheck = false;
59  }
```

In jedem Durchlauf wird zunächst die Art der Verknüpfung bestimmt. Tinnitus Navigator kennt zum Stand der Implementierung die Verknüpfungsoperatoren „AND“ und „OR“ und deren Inverse „NAND“ und „NOR“. Anschließend wird der Wahrheitswert der zu verknüpfenden Regel anhand deren ID bestimmt und mit dem bereits bestehenden Wert verknüpft. Dieser Vorgang wird so lange wiederholt, bis entweder die nächste Regel eine andere *rule_id* besitzt oder das Ende der Liste *rulesList* erreicht wurde. Das Beispiel in Abbildung 5.40 soll diesen Vorgang noch einmal verinnerlichen.

Untersucht werden soll dabei der Gesamtwahrheitswert der Regel 4711. Zunächst wird der Wahrheitswert dieser Regel in der Liste *rulesCheck* nachgeschlagen. Anschließend wird der erste Eintrag in der *rulesList* gesucht, dessen *rule_id* ebenfalls 4711 ist. Dieser zeigt eine AND-Verknüpfung mit der Regel 4712 an. Daraufhin wird in der Liste *rules-Check* der Wahrheitswert der Regel 4712 nachgeschlagen und mit dem Wahrheitswert von Regel 4711 verundet. Die Wahrheitstabelle in Abbildung 5.40 zeigt das Zwischenergebnis nach dem ersten Durchlauf an. Anschließend wird mit der Regel 4713 die nächste Regel in der *rulesList* bestimmt, die mit Regel 4711 verknüpft ist. Daraufhin wird wieder der Wahrheitswert in der List *rulesCheck* bestimmt und mit dem Zwischenergebnis des ersten Durchlaufs verundet. Dieser Vorgang wird so lange wiederholt, bis mit Regel 4715 die letzte Regel in der Liste *rulesList* gefunden und mit dem Zwischenergebnis von Durchlauf 3 verknüpft wurde.



Wahrheitstabelle

Durchlauf	Zwischenergebnis alt	Verknüpfte Regel	Verknüpfungsoperator	Wahrheitswert	Zwischenergebnis neu
0	-		-	FALSE	FALSE
1	FALSE	4712	AND	TRUE	FALSE
2	FALSE	4713	AND	TRUE	FALSE
3	FALSE	4714	OR	TRUE	TRUE
4	TRUE	4715	AND	TRUE	TRUE

Abbildung 5.40: Ermittlung des Gesamtwahrheitswertes einer verknüpften Regel

Der Gesamtwahrheitswert kann dann in der untersten, rechten Zelle der Wahrheitstabelle abgelesen werden. Daraus ist zu erkennen, dass Regel 4711 in dieser verknüpften Form gültig ist und in der vorläufigen Ergebnisliste *recommendationList* gespeichert wird.

3. Entfernen von widersprüchlichen Therapievorschlägen aus der Ergebnisliste (Listing 5.34)

Nachdem alle potentiellen Therapievorschläge ermittelt sind, müssen in einem letzten Schritt alle Therapievorschläge wieder entfernt werden, die im Widerspruch zu einem anderen Therapievorschlag in der *recommendationList* stehen (Listing 5.34)

Dazu werden aus der Tabelle *recommendations* alle Therapievorschläge geladen, zu denen dieser unvereinbar ist (Zeile 19 - 27). Ist die Ergebnismenge leer, werden in den Zeilen 40 - 48 anhand von *recommendation_id* Titel und Beschreibung des Therapievorschlags aus der Datenbank ermittelt. Schließlich wird das daraus erstellte *Recommendation*-Objekt der endgültigen Ergebnisliste hinzugefügt (Zeile 50 & 51). Im Fall, dass die Anfrage an die Tabelle *recommendations* IDs widersprüchlicher Therapievorschläge zurückgibt, wird die *recommendationList* in den Zeilen 29 – 38 nach diesen durchsucht. Nur wenn *recommendationList* keine dieser IDs enthält, wird der Therapievorschlag ebenfalls der Ergebnisliste hinzugefügt.

Listing 5.34: Entfernen von widersprüchlichen Therapievorschlägen

```
1  // now test whether there are contradicted
2  // recommendations in the solution set
3  ArrayList<Recommendation> resultList = new
4                                  ArrayList<Recommendation>();
5
6  if( recommendationList.isEmpty() ) return resultList;
7  else{
8     RecommendationDatabaseHelper helper = new
9                          RecommendationDatabaseHelper(context);
10    SQLiteDatabase recDB = helper.getReadableDatabase();
11
12    for(int recommendation_id : recommendationList){
13
14    // first check if there is at least one record containing
15    // current recommendation_id as a contradicted
16    // recommendation.if the cursor is empty, query associated
17    // recommendation record and add it to resultList
18
19    Cursor contradictedCursor =
20       recDB.query(RecommendationDatabaseHelper
21             .TABLE_RECOMMENDATION, new
```

```
22              String[]{RecommendationDatabaseHelper
23              .COLUMN_RECOMMENDATION_ID},
24              RecommendationDatabaseHelper
25              .COLUMN_CONTRADICTED_RECOMMENDATION+
26              "="+recommendation_id, null, null, null,
27              null);
28      boolean isContradicted = false;
29      if( contradictedCursor.moveToFirst() ){
30         while( !contradictedCursor.isAfterLast() ){
31            if(recommendationList.contains(
32                           contradictedCursor.getInt(0)) ){
33               isContradicted = true;
34               break;
35            }
36            contradictedCursor.moveToNext();
37         }
38      }
39      if( !isContradicted ){
40         Cursor recCursor =
41               recDB.query(RecommendationDatabaseHelper
42               .TABLE_RECOMMENDATION, new
43               String[]{RecommendationDatabaseHelper
44               .COLUMN_TITLE, RecommendationDatabaseHelper
45               .COLUMN_RECOMMENDATION},
46               RecommendationDatabaseHelper
47               .COLUMN_RECOMMENDATION_ID+"="+
48               recommendation_id, null, null, null, null);
49         recCursor.moveToFirst();
50         resultList.add(new Recommendation(recommendation_id,
51               recCursor.getString(0), recCursor.getString(1)));
52         recCursor.close();
```

```
53          }
54          contradictedCursor.close();
55        }
56        recDB.close();
57        helper.close();
58      }
59      return resultList;
60  }
```

Sofern die Ergebnisliste nicht leer ist, öffnet sich, nachdem der Algorithmus terminiert hat, eine neue Seite in der App. Diese stellt alle gültigen Therapievorschläge über einen ViewPager dar. Mehrere Therapievorschläge können somit wie in einem Buch durch eine Wischgeste nach links oder rechts durchblättert werden. Wie Abbildung 5.41 veranschaulicht, enthält jede Seite des ViewPagers den Titel und die Beschreibung eines Therapievorschlags.

Abbildung 5.41: Darstellung eines Therapievorschlags in Tinnitus Navigator

5.2.9 ECA-Regeln

Der vorgestellte Algorithmus zur Generierung von Therapievorschlägen basiert auf dem Konzept von ECA-Regeln. Der Name ECA, der für „Event Condition Action“ steht, bezieht sich dabei auf die Struktur von aktiven Regeln in ereignisgesteuerten Systemen bzw. in aktiven Datenbanken. Allgemein ist der Ablauf einer ECA-Regel dergestalt, dass zunächst ein bestimmtes Ereignis E eintreten muss, welches die Regel auslöst. Sobald dies geschehen ist, wird eine Bedingung C geprüft. Nur wenn diese Bedingung erfüllt ist, wird die Aktion A schließlich ausgeführt.

Formal wird eine ECA-Regel als ein Tupel E x C x A definiert. Das Ereignis E kann entweder ein einzelnes oder ein zusammengesetztes Ereignis sein. Letzteres besteht aus Einzelereignissen, die durch logische Operatoren miteinander verknüpft sind. Die Bedingung C ist eine boolesche Funktion, die beispielsweise überprüft, ob ein SQL-Statements Datensätze zurückliefert oder nicht. Ist die Bedingung erfüllt, werden alle Aktionen in A ausgeführt. Eine solche Aktion kann zum Beispiel das Einfügen oder das Ändern von Daten in einer Tabelle sein. Gleichzeitig kann die Ausführung von A auch ein Trigger für weitere ECA-Regeln sein, die somit beliebig verschachtelt werden können. Dadurch ist es möglich, dass die Auswertung dieser Regeln niemals terminiert. Verhindert werden kann dies zum Beispiel durch eine obere Grenze der Rekursionstiefe, deren Überschreiten den Abbruch der Regel zur Folge hat. Da die ECA-Regeln bei der Bestimmung von Therapievorschlägen immer terminieren, wird auf dieses Problem nicht weiter eingegangen und auf die Quellen [Pat99] und [WC95] verwiesen.

Dennoch kann es auch bei der Generierung von Therapievorschlägen zu einer Verschachtelung von ECA-Regeln kommen. Wie aus dem vorangehenden Kapitel bekannt ist, ist dieser Vorgang in drei Abschnitte aufgeteilt. Abhängig von diesen Abschnitten nehmen die verwendeten ECA-Regeln unterschiedliche Formen an. Zunächst wird im ersten Abschnitt die Gültigkeit jeder Regel für sich überprüft. Um auch alle miteinander verknüpften Regeln zu erfassen, wird dann im zweiten Abschnitt ein Gesamtwahrheitswert für jede Regel gebildet. Die ECA-Regel, die für diese beiden Abschnitte verwendet wird, hat die folgende Form:

On **Event**(Benutzer möchte Therapievorschläge erhalten)

If **Condition**(Auswertung der Regel (ggf. aus mehreren Regeln bestehend) erfolgt zu true))

Do **Action**(Füge den mit der Regel verknüpften Therapievorschlag vorläufig in die Ergebnisliste ein)

Ist mindestens eine Regel wahr und der Aktionsteil A wird ausgeführt, so triggert diese Aktion das Ereignis, welches Voraussetzung für die Ausführung des dritten Abschnitt ist. Dabei werden all diejenigen Therapievorschläge wieder herausgefiltert, die im Widerspruch zu einem anderen Therapievorschlag in der Ergebnismenge stehen. Die ECA-Regeln sehen wie folgt aus:

On **Event**(Therapievorschlag ist in vorläufiger Ergebnisliste vorhanden)

If **Condition**(Therapievorschlag verträgt sich mit mindestens einem anderen Therapievorschlag aus der Ergebnismenge nicht)

Do **Action**(Lösche Therapievorschlag aus der Ergebnisliste)

5.2.10 ECA-Regeln für XML-Dateien

Neben Datenbanken können ECA-Regeln auch auf XML-Dateien angewandt werden [Pap07]. Dies erfordert die Verwendung der beiden Abfragesprachen XPath und XQuery, die beide vom W3C standardisiert sind. Während mit XPath Teile eines XML-Dokuments adressiert und ausgewertet werden können, dient XQuery zur Abfrage der XML-Datenbanken, in denen die XML-Dokumente gespeichert sind.

Ein Ereignis einer ECA-Regel für XML-Dateien hat dabei eine der beiden Formen:

INSERT e
DELETE e

Die Variable e steht hierbei für einen beliebigen XPath-Ausdruck, dessen Ergebnis eine Menge von Knoten ist. Aufgrund der beiden Formen gibt es zwei Ereignisse, die die ECA-Regel triggern können. Bei einem INSERT-Ereignis wird die Regel ausgeführt, wenn mindestens ein neuer Kindknoten in der Knotenmenge enthalten ist.
Entsprechend wird die Regel bei einem DELETE-Ereignis ausgeführt, wenn die Knotenmenge mindestens einen gelöschten Kindknoten enthält.

Auch Bedingungen werden in Form eines XPath-Ausdrucks geschrieben. Ein XPath-Ausdruck kann hierbei aus mehreren XPath-Ausdrücken bestehen, die durch logische Operatoren miteinander verknüpft sind. Zusätzlich dazu kann die Bedingung auch konstant den Wert TRUE annehmen.

Wenn die Bedingung erfüllt ist, werden die im Aktionsteil spezifizierten Aktionen ausgeführt. Diese können wie folgt geformt sein:

INSERT r BELOW e [BEFORE| AFTER] q
DELETE e

Während e wiederum einen beliebigen XPath-Ausdruck bezeichnet, steht die Variable r für einen XQuery-Ausdruck. Schließlich gibt es noch die Variable q, welche einen sogenannten Qualifier beschreibt. Als Qualifier wird in XPath ein boolescher Ausdruck bezeichnet, der stets auf jeden Knoten einer Knotenmenge angewandt wird und diese dementsprechend filtert.
Bei einer INSERT-Aktion werden die Kindknoten r direkt unter der Knotenmenge e eingefügt. Mit den Anweisungen BEFORE und AFTER kann die Einfügestelle weiter spezifiziert werden. AFTER q drückt aus, dass die Kindknoten nach dem letzten Geschwisterknoten eingefügt werden, für den q TRUE ausgibt. Entsprechend bedeutet BEFORE q, dass die Kindknoten vor dem ersten Geschwisterknoten eingefügt werden, für den q TRUE ausgibt.
Bei einer DELETE-Aktion gibt e die Knotenmenge an, die mit all ihren Kindknoten gelöscht wird

Mit den beschriebenen Konzepten ist es möglich, Informationen aus XML-Dokumenten auszutauschen und zu speichern. Diese Anforderungen sind vor allem stark in den Bereichen e-Learning und e-Commerce gefragt. Aus diesem Grund eignen sich ECA-Regeln in XML-basierten Datenbanken besonders im semantischen Web. Für die Bestimmung von Therapievorschlägen ist diese Art von ECA-Regeln somit eher ungeeignet.

5.2.11 Entscheidungsbäume als Alternative zu ECA-Regeln

Neben ECA-Regeln können zur Ermittlung von Therapievorschlägen auch andere Datenstrukturen verwendet werden. Eine dieser Strukturen sind Präfixbäume, deren Kanten jeweils mit einem bestimmten Label markiert sind. Eine besondere Form des Präfixbaums ist der Entscheidungsbaum, in dem jeder Knoten, außer er ist ein Blattknoten, immer genau zwei Nachfolger besitzt. Der linke Nachfolgerknoten steht dabei immer für eine negative Entscheidung, der rechte Nachfolgerknoten immer für eine positive Entscheidung. Dementsprechend sind die Kanten, die einen Knoten mit seinen beiden Nachfolgern verbinden, mit den booleschen Wahrheitswerten 0 und 1 beschriftet. Jeder Pfad von der Wurzel bis zu einem Blatt steht somit für eine Reihe von 0-1-wertigen Entscheidungen, die mit einer bestimmten Aktion A abgeschlossen werden. Jedes Blatt speichert dazu immer genau eine Aktion. Abbildung 5.42 stellt einen solchen Baum dar.

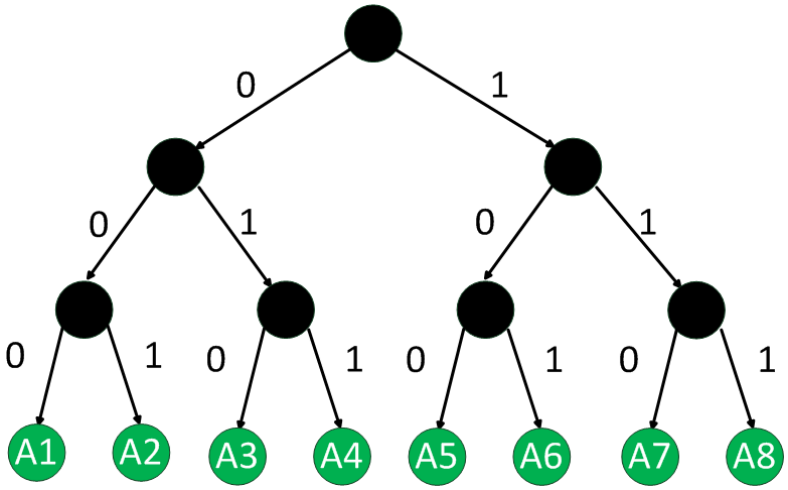


Abbildung 5.42: Entscheidungsbaum

Angewandt auf die Ermittlung von Therapievorschlägen ist ein Entscheidungsbaum G mit G = (V, E) ein azyklisch, gerichteter Graph, der zusätzlich folgende Eigenschaften besitzt:

- Jeder Knoten der Knotenmenge V, der kein Blatt ist, speichert jeweils ein SQL-Statement, mit dem die Existenz eines bestimmten Datensatzes in der Datenbank der App überprüft wird
- Jeder Pfad von der Wurzel zu einem Blatt des Entscheidungsbaumes wird durch einen 0-1-wertigen String repräsentiert
- Jedes Blatt speichert genau einen Therapievorschlag, der mit genau einem 0-1-wertigen String assoziiert ist

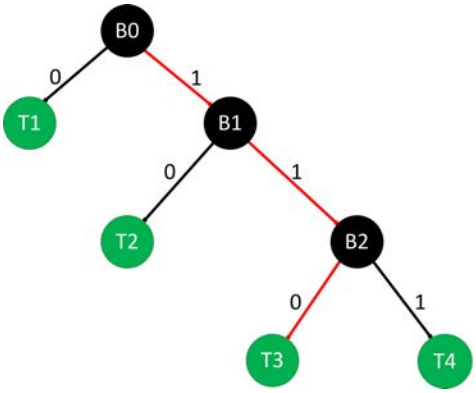
Anhand des Entscheidungsbaums in Abbildung 5.43 soll dieses Prinzip nochmals veranschaulicht werden. Dieser Baum besteht aus drei Entscheidungen, die sich auf die Ergebnisse eines Hörtests beziehen, welche in Tabelle 5.6 abgebildet sind. Zum besseren Verständnis sind die Bedingungen, die den Entscheidungen zugrunde liegen in diesem Beispiel allerdings in Textform und nicht als SQL-Statement formuliert.

Frequenz	**Hörverlust**
125 Hz	15 %
250 Hz	11 %
500 Hz	18 %
1000 Hz	37 %
2000 Hz	43 %
4000 Hz	59 %
8000 Hz	48 %

Tabelle 5.6: Hörtestergebnisse

Zunächst wird Bedingung B0 geprüft. Da diese gültig ist, wird die Kante mit Label 1 rot markiert und als nächste Bedingung B1 getestet. Nach Tabelle 5.6 trifft auch diese Bedingung zu, da zum Beispiel bei 2000 Hz ein Hörverlust von 43% gemessen wurde. Aus diesem Grund wird wieder die Kante mit Label 1 rot markiert und Bedingung B2 untersucht. Da im Beispiel-Hörtest jedoch bei keiner Frequenz ein Hörverlust von mehr als 60% des Hörvermögens aufgetreten ist, wird dieses Mal die linke Kante mit Label 0 rot markiert.

Über diese Kante wird schließlich das Blatt T3 erreicht, welches dem Nutzer aufgrund eines mittelschweren Hörverlustes das Tragen eines Hörgeräts empfiehlt. Der Therapievorschlag T3 kann somit über den Pfad „110“ im Entscheidungsbaum gefunden werden.



Bedingung	textuelle Beschreibung
B0	Es wurde mindestens ein Hörtest durchgeführt
B1	Auf mindestens einer Frequenz ist der Hörverlust größer als 40% des normalen Hörvermögens
B2	Auf mindestens einer Frequenz beträgt der Hörverlust mehr als 60% des normalen Hörvermögens

Therapievorschlag	Beschreibung
T1	Hörtest durchführen
T2	-
T3	Mittelschwerer Hörverlust. Um den Hörverlust auszugleichen, wird das Tragen eines Hörgeräts empfohlen
T3	Schwerer Hörverlust. Es wird dringend empfohlen, ein Hörgerät zu tragen!

Abbildung 5.43: Bestimmen eines Therapievorschlags durch einen Entscheidungsbaum

Aus diesem Beispiel lassen sich zwei positive Eigenschaften ableiten. Zum einen müssen sich nicht alle Blätter auf einer bestimmten Ebene befinden. Lässt sich ein Sachverhalt durch weniger Entscheidungen beschreiben, kann auf diese Weise der Pfad abgekürzt und dadurch die Performanz der Abfrage gesteigert werden. Zum anderen zeigt dieses Beispiel, dass sich mit Entscheidungsbäumen komplexere Datenbankanfragen besonders einfach modellieren lassen. Jede Ebene steht dabei für eine Teilanfrage, die mit der darunterliegenden Ebene, sofern es sich nicht um die Blätter-Ebene handelt, verundet wird.

Dennoch gibt es auch einige Nachteile bei der Verwendung eines Entscheidungsbaumes. Um beispielsweise zu überprüfen, ob die Therapievorschläge zutreffen, müssen jeweils alle Pfade eines Baumes durchlaufen werden. Dies wirkt sich besonders dann negativ auf die Performanz aus, wenn in einzelnen Blättern kein Therapievorschlag gespeichert ist. Dies kann vorkommen, da in einem Entscheidungsbaum wie oben beschrieben jeder Knoten stets zwei Nachfolger besitzen muss. Zwar müssen auch alle ECA-Regeln einzeln auf Gültigkeit geprüft werden, da eine Regel jedoch so definiert ist, dass diese immer genau einen Therapievorschlag enthalten muss, kann dieser Effekt nicht auftreten.

Ebenfalls zeitaufwendig sind die Einfüge- und Löschoperationen im Baum. So muss zum Beispiel beim Einfügen eines neuen Knotens zunächst der Baum durchlaufen werden, bis die richtige Stelle gefunden ist. Nach dem Einfügen muss schließlich überprüft werden, ob die Voraussetzungen aller unterhalb des neuen Knotens befindlichen Therapievorschläge weiterhin erfüllt sind. Auch in dieser Hinsicht eignen sich ECA-Regeln besser, da neue Regeln immer nur dort eingefügt werden, wo es der mit der Regel verknüpfte Therapievorschlag verlangt.
Ein letzter negativer Punkt bezieht sich auf den hohen Speicherplatzbedarf von Entscheidungsbäumen. Denn bereits mit der Wurzel des Baumes wird der Bereich festgelegt, auf den sich die Bedingungen in den einzelnen Knoten beziehen. Damit ist es nur schwer möglich, zwei unterschiedliche Sachverhalte in einem Entscheidungsbaum so zu integrieren, dass jeder Knoten nur einmal verwendet wird. Im schlimmsten Fall müssen sogar mehrere Entscheidungsbäume erzeugt und verwaltet werden. Diese Nachteile waren letztendlich dafür ausschlaggebend, den Algortihmus zur Ermittlung von Therapievorschlägen nicht mit einem Entscheidungsbaum, sondern auf Basis von ECA-Regeln zu implementieren.

6

Anforderungsabgleich

Nachdem in den letzten beiden Kapiteln die Architektur und die Oberfläche der Webanwendung und der App erläutert wurden, sollen nun die in Kapitel 3 gesteckten funktionalen und nichtfunktionalen Anforderungen damit abgeglichen werden. Die Tabellen 6.1 und 6.2 stellen dazu in der ersten Spalte jeweils die Anforderung dar. Die zweite Spalte gibt an, ob die Anforderung erfüllt ist. Dabei wird zwischen „erfüllt“, „teilweise erfüllt“ und „nicht erfüllt“ unterschieden. Anschließend folgt in der dritten Spalte eine kurze Begründung. In der vierten Spalte wird schließlich eine Schulnote zwischen 1 und 5 vergeben, um das Ergebnis der einzelnen Anforderungen untereinander besser vergleichen zu können. Abgeschlossen wird dieses Kapitel mit einer kurzen Zusammenfassung der Ergebnisse.
Zunächst werden nun die funktionalen Anforderungen mit dem Istzustand verglichen.

Anforderung	Erfüllt?	Begründung	Note
FA 1	Ja	Die App bietet entweder über den Kalender oder direkt über das Menü Eingabemöglichkeiten für tagesaktuelle Ereignisse, Dokumente oder Hörtestergebnisse	1
FA 2	Ja	Im Kalender kann an ein beliebiges Datum gesprungen werden. Zusätzlich kann in der aktuellen Monatsansicht durch eine Wischgeste nach links bzw. rechts jeweils ein Monat vor bzw. zurück gesprungen werden. Editierfunktion durch langes Drücken auf einen Tag erreichbar. Funktionsweise ähnlich dem Kalender S-Planner von Samsung	1
FA 3	Ja	Sowohl Medikamente als auch Vorerkrankungen können unterhalb des Kalenders eingetragen und auch bearbeitet werden.	1
FA 4	Ja	Nutzer kann Dokumente sowohl direkt hochladen, als auch Hörtestergebnisse in die vorgesehene Eingabemaske eintragen. Dort können auch spezielle Frequenzen zusätzlich eingegeben werden. Nach Eingabe aller Daten kann Hörtestergebnis in Liniendiagramm betrachtet werden.	1
FA 5	Teilweise	Möglichkeit zum Ausfüllen von Fragebögen ist gegeben. Ein fertiger Fragebogen kann aktuell aber noch nicht im Kalender angezeigt bzw. erneut ausgefüllt werden	3
FA 6	Teilweise	Bislang nur Synchronisierung von App zu Server	3
FA 7	Ja	Synchronisation in beide Richtungen	1
FA 8	Teilweise	App kann offline genutzt werden. Synchronisierung wird allerdings noch nicht nachgeholt	3
FA 9	Nein	Status aktuell noch nicht umgesetzt	5

FA 10	Ja	Sowohl Regeln als auch Therapievorschläge können erstellt werden. Regeln können außerdem miteinander verknüpft werden	1
FA 11	Nein	Aktuell werden nur Regeln verwendet, die beim ersten Start der App heruntergeladen wurden	4
FA 12	Ja	Therapievorschläge werden angezeigt. Außerdem wird begründet, warum der Vorschlag gegeben wurde.	1
FA 13	Nein	Übersicht existiert in der App noch nicht. Texte, die Therapiearten beschreiben, sind allerdings schon vorhanden	4

Tabelle 6.1: Bewertung der funktionalen Anforderungen

Als nächstes folgt nun der Anforderungsabgleich mit den nichtfunktionalen Anforderungen.

Anforderung	**Erfüllt?**	**Begründung**	**Note**
NFA 1	Ja	Datenschutzerklärung fehlt in der App	5
NFA 2	Ja	Alle Funktionen in der App sind nach den aktuellen Android-Bedienkonzepten umgesetzt. Dazu gehört die Menüstruktur, die durch eine Wischbewegung aktiviert bzw. deaktiviert werden kann. Das Bedienkonzept des Kalenders ist dem integrierten Kalender nachempfunden. Außerdem lassen sich viele Seiten der App mit Wischeffekten bedienen. Schließlich stellt die Verwendung der Kompatibilitätsbibliothek sicher, dass diese Funktionen bereits auf Geräten mit Android 2.3.3 verfügbar sind.	1

NFA 3	Ja	Alle wichtigen Eingabemöglichkeiten können über drei verschiedene Wege erreicht werden. Über das Menü, über den Kalender und über die persönliche Tinnitusakte	1
NFA 4	Ja	Die Speicherung der Daten erfolgt über die gleiche API wie Track Your Tinnitus. Dementsprechend werden die Daten auch in der gleichen Datenbank gespeichert. Die Eingabemöglichkeit für Therapievorschläge und Regeln ist in die Webseite integriert, die für das Track Your Tinnitus Projekt entwickelt wurde. In der App selbst ist eine Funktion zum Wechsel zwischen Tinnitus Navigator und Track Your Tinnitus angedacht, allerdings noch nicht implementiert. Ähnliche Bedienelemente sind zum Beispiel das Sliding-Menü und die Eingabe von Fragenbögen.	2

Tabelle 6.2: Bewertung der nichtfunktionalen Anforderungen

Aus den beiden Tabellen 6.1 und 6.2 geht hervor, dass alle Anforderungen, die benötigt werden, um einen Therapievorschlag zu generieren, vollständig umgesetzt sind. Bei den meisten Anforderungen, die mit einer befriedigenden oder schlechteren Note bewertet sind, handelt es sich um wünschenswerte, zusätzliche Funktionen. Diese konnten aufgrund des großen Aufwandes zur Implementierung der Grundfunktionen von Webanwendung und App nicht mehr implementiert werden. Allerdings lassen sich diese Funktionen mit geringem Aufwand nach Abgabe dieser Arbeit erfüllen, sodass sich die App insgesamt mit einer guten bis sehr guten Note bewerten lässt.

7

Fazit

Mit Beendigung dieser Masterarbeit ist ein erster, großer Meilenstein in der Entwicklung von Tinnitus Navigator abgeschlossen. Dieses Kapitel fasst daher die Ergebnisse dieser Arbeit in Abschnitt 7.1 zusammen. Abschnitt 7.2 gibt abschließend einen Ausblick darüber, wie Tinnitus Navigator weiter verbessert werden kann.

7.1 Zusammenfassung

Das Ziel dieser Masterarbeit war die Entwicklung einer mobilen Anwendung, die Tinnitus-Patienten einen geeigneten Therapievorschlag erstellen soll. Mit der App Tinnitus Navigator ist dieses Ziel erfolgreich umgesetzt worden.

Bei der Entwicklung der Oberfläche von Tinnitus Navigator wurde viel Wert darauf gelegt, bekannte Bedienkonzepte aus anderen Apps einzubauen, sodass Nutzer nur wenig Eingewöhnungszeit benötigen. So können Seiten, die nur Informationen anzeigen, mit Wischgesten wie ein Buch durchblättert werden. Dazu gehört beispielsweise die Seite zur Auswahl von Fragebögen oder die Seite zur Darstellung der Therapievorschläge. Ein weiteres Beispiel ist die Kacheloptik im Startmenü und in der persönlichen Tinnitusakte. Diese ist an die Nachrichten-App Flipboard angelehnt und ermöglicht es, dass die App auch noch auf kleinen Bildschirmen gut bedient werden kann.
Über die persönliche Tinnitusakte können alle wichtigen Funktionen erreicht werden, die zur Ermittlung eines Therapievorschlags regelmäßig ausgeführt werden müssen. Dazu gehören die Beantwortung von Fragebögen, die Eingabe von Ereignissen und das Eintragen von Hörtestergebnissen. Nach Eingabe aller Hörtestwerte können diese zudem in einem Liniendiagramm visualisiert betrachtet werden. Schließlich können über die App auch Dokumente wie Untersuchungsergebnisse oder Röntgenaufnahmen auf den Server hochgeladen werden. Aufgrund fehlender Auswertungssoftware trägt diese Funktion aktuell allerdings nicht zur Ermittlung von Therapievorschlägen bei.
Die zentrale Stelle zur Verwaltung aller eingegebenen Daten ist der eigens für die App entwickelte Kalender, der ebenfalls über die persönliche Tinnitusakte aufgerufen werden kann. Der Kalender stellt jeweils einen Monat dar und hebt alle Tage optisch hervor, an denen Daten eingetragen wurden. Diese können dann aufgelistet und, falls gewünscht, bearbeitet oder gelöscht werden. Auch neue Daten lassen sich über den Kalender eingeben. Bedienen lässt sich der Kalender unter anderem durch Wischgesten, mit denen monatsweise vor- oder zurückgesprungen werden kann. Eine Funktion zur Überbrückung größerer, zeitlicher Abstände ist ebenfalls integriert.
Viele Funktionen der App können außerdem auf mehrere Wege erreicht werden. Dazu trägt auch die Verwendung eines Sliding-Menüs bei, über welches die wichtigsten Funktionen der App direkt aufrufbar sind.
Zu diesen Funktionen gehört zweifelsohne die Ermittlung von Therapievorschlägen aus den eingegebenen Daten mit Hilfe von ECA-Regeln. Dazu wurde ein Algorithmus entwickelt, der diese Regeln zuerst auswertet und anschließend alle Therapievorschläge ermittelt, die mit den gültigen Regeln assoziiert sind.

Da sich Therapievorschläge auch widersprechen können, enthält der Algorithmus eine Funktion, die widersprüchliche Therapievorschläge wieder aus der Ergebnismenge herausfiltert. Um auch komplexere Sachverhalte mit den Regeln modellieren zu können, ist es außerdem möglich, diese miteinander zu verknüpfen. Der Algorithmus ist daher so implementiert, dass er diese Verknüpfungen erkennen und daraus einen Gesamtwahrheitswert berechnen kann.
Die Regeln werden, wie auch die Therapievorschläge, über eine Webapplikation verwaltet. Um neue Regeln bzw. Therapievorschläge eintragen zu können, gibt es für beide jeweils eine separate Eingabemaske. Bereits existierende Regeln und Therapievorschläge können jeweils auf einer Übersichtsseite betrachtet werden. Über diese Seiten ist es jeweils auch möglich, Regeln bzw. Therapievorschläge zu bearbeiten oder zu löschen.

Zum Stand dieser Arbeit ist die Entwicklung der App soweit vorangeschritten, dass Nutzer Therapievorschläge auf Basis der eingegebenen Daten erhalten können. Die Implementierung der wichtigsten Funktionen ist somit abgeschlossen. Um die App zu veröffentlichen, bedarf es allerdings noch ein paar kleiner Ergänzungen, die im Anschluss an diese Arbeit umgesetzt werden.

7.2 Ausblick

Im Rahmen dieser Arbeit sind in vielen Meetings immer wieder neue Anforderungen diskutiert worden, die die Funktionalität von Tinnitus Navigator sinnvoll erweitern. Aufgrund der begrenzten Zeit konnten allerdings nicht alle diese Anforderungen in diese erste Version von Tinnitus Navigator integriert werden. Aus diesem Grund stellt dieser Abschnitt nun einige der Ideen vor, die langfristig in Tinnitus Navigator umgesetzt werden sollen.

1. **Unterstützung weiterer Displaygrößen und Betriebssysteme**
   Die Entwicklung und das Testen der App erfolgten auf einem Samsung Galaxy S4. Der Bildschirm dieses Smartphones ist nicht nur hochauflösend, sondern im Vergleich zu anderen Smartphones auch ziemlich groß.

Das bedeutet, dass Layout-Elemente wie Buttons oder Texte auf Smartphones mit kleineren Bildschirmen unter Umständen nicht so dargestellt werden, wie dies eigentlich gewünscht ist. Durch eine Unterstützung der gängigsten Bildschirmgrößen und Bildschirmauflösungen soll eine hohe Reichweite von Tinnitus Navigator gewährleistet werden. Um diese Reichweite noch weiter zu erhöhen, soll Tinnitus Navigator außerdem auf andere mobile Betriebssysteme wie zum Beispiel Apple iOS oder Windows Phone portiert werden.

2. **Erstellen eines Forums**
   Tinnitus Navigator soll nicht nur geeignete Therapievorschläge erstellen, sondern Nutzern auch hilfreiche Informationen zum Thema Tinnitus zur Verfügung stellen. Zum Beispiel ist in die App bereits eine Seite integriert, die einen Überblick über die wichtigsten Therapiearten geben soll. Um den Informationswert der App weiter zu erhöhen, ist ein Forum geplant, in dem Nutzer gegenseitig ihre Erfahrungen mit Tinnitus austauschen können. Dieses Forum soll sowohl über die Webseite als auch über die App erreichbar sein. Eine Kachel, über die das Forum aufgerufen werden kann, ist im Startmenü der App bereits enthalten.

3. **Verbesserung des Empfehlungssystems**
   In der aktuellen Implementierung werden in Tinnitus Navigator nur Therapievorschläge angezeigt, die von Experten über die Webapplikation eingetragen wurden. Eine wichtige Komponente bei der Empfehlung einer Therapie sind allerdings Erfahrungswerte anderer Tinnitus-Patienten. Aus diesem Grund soll ein Ratingsystem eingeführt werden, über das Nutzer nach Abschluss einer von Tinnitus Navigator empfohlenen Therapie bewerten können, wie hilfreich diese Therapieempfehlung war. Je nachdem, wie ähnlich der Verlauf des Tinnitus zweier Nutzer ist, kann eine solche Bewertung dann mit in die Berechnung eines Therapievorschlags einfließen. Ergibt sich zum Beispiel aus dem Ratingsystem, dass bei Patientinnen zwischen 30 und 40 Jahren mit subakutem Tinnitus ein regelmäßiges Entspannungsbad mit klassischer Musik den Tinnitus stark lindert, so könnte diese Empfehlung auch einer 32-jährigen Patientin mit subakutem Tinnitus gegeben werden. Bei einem 50 Jahre alten Patienten ist es hingegen wahrscheinlicher, dass eine andere Therapie mehr Erfolg verspricht.

4. **Einbindung von Facharztinformationen**

   Auf der Empfehlungsseite der App werden bislang nur der Titel der Empfehlung und eine genaue Beschreibung der Therapie angezeigt. Gerade für Patienten mit akutem Tinnitus kann es jedoch hilfreich sein, zusätzlich Kontaktinformationen zu HNO-Ärzten in der Nähe aufzulisten, die sich auf Tinnitus spezialisiert haben. Diese Funktion soll Nutzer eine langwierige Suche nach einem geeigneten Facharzt ersparen. Gleichzeitig soll diese Funktion Nutzer aber auch motivieren, schnellst möglich Kontakt zu einem Spezialisten aufzunehmen, der ihnen bei ihrem Problem helfen kann.

5. **Annäherung an die App Track Your Tinnitus**

   Tinnitus Navigator ist nicht die erste App dieses Projekts. Mit der App Track Your Tinnitus von Jochen Herrmann können Schwankungen der Tinnituswahrnehmung erfasst und besser verstanden werden [Her]. Damit Nutzer nicht immer zwischen den Apps hin- und herwechseln müssen, sollen beide Apps zukünftig zusammengeführt werden. Vorstellbar wäre zum Beispiel ein Portal, das die zentrale Anmeldung übernimmt und dann Zugriff auf die Funktionen beider Apps bietet. Aus diesem Grund wurden in Tinnitus Navigator bereits einige Vorkehrungen getroffen, die diesen Vorgang vereinfachen. So können sich Nutzer, die sich bereits im Tinnitus-Projekt registriert haben, mit dem gleichen Account auch in Tinnitus Navigator anmelden. Des Weiteren verwendet Tinnitus Navigator dieselbe API, um Daten mit dem Server auszutauschen. Dementsprechend wird auch dieselbe Datenbank auf dem Server verwendet, um die Daten langfristig zu speichern. Auch in der App finden sich bereits solche Ansätze. Das Sliding-Menü enthält zum Beispiel einen Eintrag, über den zukünftig Track Your Tinnitus aufgerufen werden kann.

6. **Mobile Prozesse**

   Schließlich kann auch die Interaktion zwischen Tinnitus-Patienten, Ärzten und Therapeuten mit Hilfe von mobilen Prozessen verbessert werden. Beispiele, in denen mobile Anwendungen durch mobile Prozesse erfolgreich unterstützt wurden, finden sich in [PTKR10][PTR10][PLRH12][PMLR14].

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[Biq] *Balsamiq Mockups*. http://balsamiq.com/products/mockups/. – zuletzt besucht am 13.05.2014

[CNB+13] CROMBACH, Anselm ; NANDI, Corina ; BAMBONYE, Manassé ; LIEBRECHT, Martin ; PRYSS, Rüdiger ; REICHERT, Manfred ; ELBERT, Thomas ; WEIERSTALL, Roland: Screening for mental disorders in post-conflict regions using computer apps - a feasibility study from Burundi. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013, S. 70–70

[Cos] *European Cooperation in Science and Technology*. http://cost.tinnitusresearch.org/. – zuletzt besucht am 13.05.2014

[ECV+98] E., Biesinger ; C., Heiden ; V., Greimel ; T., Lendle ; R., Hoing ; K., Albegger: Strategien in der ambulanten Behandlung des Tinnitus. In: *HNO* (1998)

[Fei] *ViewPager von Jeremy Feinstein*. https://github.com/jfeinstein10/JazzyViewPager. – zuletzt besucht am 13.05.2014

[FKS06] FRANK, Wilhelm ; KONTA, Brigitte ; SEILER, Gerda: *Therapie des unspezifischen Tinnitus ohne organische Ursache*. Deutsches Institut für Medizinische Dokumentation und Information (DIMDI), 2006

[Fli] *Flipboard*. https://play.google.com/store/apps/details?id=flipboard.app&hl=de. – zuletzt besucht am 13.05.2014

[Gooa] *BaseAdapter*. http://developer.android.com/reference/android/widget/BaseAdapter.html. – zuletzt besucht am 13.05.2014

[Goob] *Fragment Lebenszyklus*. http://developer.android.com/images/fragment_lifecycle.png. – zuletzt besucht am 13.05.2014

[Gooc] *Fragment-Manager*. http://developer.android.com/reference/android/app/FragmentManager.html. – zuletzt besucht am 13.05.2014

[Good] *Fragment*. http://developer.android.com/guide/components/fragments.html. – zuletzt besucht am 13.05.2014

[Gooe] *Graphics Bibliothek von Android*. http://developer.android.com/reference/android/graphics/package-summary.html. – zuletzt besucht am 13.05.2014

[Goof] *ListView*. http://developer.android.com/guide/topics/ui/layout/listview.html. – zuletzt besucht am 13.05.2014

[Goog] *Navigation Drawer*. https://developer.android.com/design/patterns/navigation-drawer.html. – zuletzt besucht am 13.05.2014

[Gooh] *Recycler einer ListView*. http://android.amberfog.com/?p=296. – zuletzt besucht am 13.05.2014

[Gooi] *SparseBooleanArray*. http://developer.android.com/reference/android/util/SparseBooleanArray.html. – zuletzt besucht am 13.05.2014

[Gooj] *SQLiteOpenHelper-Klasse*. http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html. – zuletzt besucht am 13.05.2014

[Gook] *TabHost*. http://developer.android.com/reference/android/widget/TabHost.html. – zuletzt besucht am 13.05.2014

[GPSR13] GEIGER, Philip ; PRYSS, Rüdiger ; SCHICKLER, Marc ; REICHERT, Manfred: Engineering an Advanced Location-Based Augmented Reality Engine for Smart Mobile Devices / University of Ulm. University of Ulm, 2013. – Technical Report

[GSP+14] GEIGER, Philip ; SCHICKLER, Marc ; PRYSS, Rüdiger ; SCHOBEL, Johannes ; REICHERT, Manfred: Location-based Mobile Augmented Reality Applications: Challenges, Examples, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014, S. 383–394

[Her] *Track Your Tinnitus Projekt*. `https://secure.dbis.info/da/tinnitus/`. – zuletzt besucht am 13.05.2014

[IRLP+13] ISELE, Dorothea ; RUF-LEUSCHNER, Martina ; PRYSS, Rüdiger ; SCHAUER, Maggie ; REICHERT, Manfred ; SCHOBEL, Johannes ; SCHINDLER, Arnim ; ELBERT, Thomas: Detecting adverse childhood experiences with a little help from tablet computers. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013, S. 69–70

[Jac14] JACKSON, Wallace: *Pro Android UI*. Apress, 2014

[KVL13] KREUZER, Peter M. ; VIELSMEIER, Veronika ; LANGGUTH, Berthold: *Chronischer Tinnitus - eine interdisziplinäre Herausforderung*. 2013

[Lara] *Laravel - A Framework For Web Artisans: Controller*. `http://three.laravel.com/docs/controllers`. – zuletzt besucht am 13.05.2014

[Larb] *Laravel - A Framework For Web Artisans: Database*. `http://three.laravel.com/docs/database/eloquent#one-to-one`. – zuletzt besucht am 13.05.2014

[Larc] *Laravel - A Framework For Web Artisans: Startseite*. `http://three.laravel.com/docs/home`. – zuletzt besucht am 13.05.2014

[Pap07] PAPAMARKOS, George: *Event-Condition-Action Rule Languages over Semistructured Data*, School of Computer Science Information Systems Birkbeck College University of London, Diss., 2007

[Pat99] PATON, N.: *Active Rules in Database Systems*. Springer-Verlag, 1999

[PLRH12] PRYSS, Rüdiger ; LANGER, David ; REICHERT, Manfred ; HALLERBACH, Alena: Mobile Task Management for Medical Ward Rounds - The MEDo Approach. In: *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*, Springer, 2012, S. 43–54

[PMLR14] PRYSS, Rüdiger ; MUNDBROD, Nicolas ; LANGER, David ; REICHERT, Manfred: Supporting medical ward rounds through mobile task and process management. In: *Information Systems and e-Business Management* (2014)

[PTKR10] PRYSS, Rüdiger ; TIEDEKEN, Julian ; KREHER, Ulrich ; REICHERT, Manfred: Towards Flexible Process Support on Mobile Devices. In: *Proc. CAiSE'10 Forum - Information Systems Evolution*, Springer, 2010, S. 150–165

[PTR10] PRYSS, Rüdiger ; TIEDEKEN, Julian ; REICHERT, Manfred: Managing Processes on Mobile Devices: The MARPLE Approach. In: *CAiSE'10 Demos*, 2010

[RES] *Representational State Transfer (REST)*. `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`. – zuletzt besucht am 13.05.2014

[RLPL+13] RUF-LEUSCHNER, Martina ; PRYSS, Rüdiger ; LIEBRECHT, Martin ; SCHOBEL, Johannes ; SPYRIDOU, Andria ; REICHERT, Manfred ; SCHAUER, Maggie: Preventing further trauma: KINDEX mum screen - assessing and reacting towards psychosocial risk factors in pregnant women with the help of smartphone technologies. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013, S. 70–70

[RPR11] ROBECKE, Andreas ; PRYSS, Rüdiger ; REICHERT, Manfred: DBIScholar: An iPhone Application for Performing Citation Analyses. In: *CAiSE Forum-2011*, CEUR Workshop Proceedings, 2011 (Proceedings of the CAiSE'11 Forum at the 23rd International Conference on Advanced Information Systems Engineering)

[SKH+10] SCHAETTE, Roland ; KÖNIG, Ovidiu ; HORNIG, Dirk ; GROSS, Manfred ; KEMPTER, Richard: *Acoustic stimulation treatments against tinnitus could be most effective when tinnitus pitch is within the stimulated frequency range*. 2010

[SRLP+13] SCHOBEL, Johannes ; RUF-LEUSCHNER, Martina ; PRYSS, Rüdiger ; REICHERT, Manfred ; SCHICKLER, Marc ; SCHAUER, Maggie ; WEIERSTALL, Roland ; ISELE, Dorothea ; NANDI, Corina ; ELBERT, Thomas: A generic questionnaire framework supporting psychological studies with smartphone technologies. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013, S. 69–69

[SSP+13] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; NIENHAUS, Hans ; REICHERT, Manfred: Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. In: *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, 2013, S. 509–518

[SSP+14] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; MAIER, Fabian ; REICHERT, Manfred: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014, S. 371–382

[WC95] WIDOM, J. ; CERI, S.: *Active Database Systems*. Morgan-Kaufmann, 1995

Name: Michael Lindinger Matrikelnummer: 661795

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Michael Lindinger