
ulm university universität
**u**ulm

**Fakultät für Ingenieurwissenschaften, Informatik und Psychologie**
Institut für Datenbanken und Informationssysteme

# Entwicklung progressiver Eingabekonzepte zur Datenerfassung auf mobilen Endgeräten

Abschlussarbeit an der Universität Ulm

**Vorgelegt von:**
Jost Jonas
jost.jonas@uni-ulm.de
925652

**Gutachter:**
Prof. Dr. Manfred Reichert
Prof. Dr. Franz Hauck

**Betreuer:**
Michael Stach

2019

Fassung 15. Oktober 2019



Satz: PDF-LaTeX 2ε

# Kurzfassung

TrackYourTinnitus ist ein universitäres Forschungsprojekt. Es hat zum Ziel, die Symptome eines Tinnitus mittels statistischer Erhebung besser zu verstehen. Die betroffenen Personen geben ihre Daten über die TrackYourTinnitus App ein. Die Dateneingabe stellt aus diesem Grund einen bedeutsamen Teil der Anwendung dar. Eine Neukonzipierung der Dateneingabe des Programms wurde seit der Erstversion nicht veröffentlicht und allgemeine Änderungen an der App wurden zuletzt vor zwei Jahren vorgenommen.[1] Mit einem neuen Konzept könnte die Dateneingabe wesentlich schneller stattfinden. Die Usability und das Eingabekonzept der App sind daher nicht optimal.

In dieser Arbeit wird für die TrackYourTinnitus App ein neues Eingabekonzept entworfen. Der Entwurf wird iterativ implementiert. Er umfasst eine nutzerorientierte Gestaltung, eine neue Gestensteuerung, und eine moderne Navigation. Die Implementierung des Entwurfs wird mit Flutter realisiert. Die neue Herangehensweise im Programmierstil des Frameworks und die Plattformunabhängigkeit überzeugten bei der Auswahl des Tools. Das Ergebnis ist eine progressive TrackYourTinnitus App mit einem neuen Eingabekonzept bei dem die Datenerfassung zu einem Benutzererlebnis wird.

[1] Gemäß dem Versionsverlauf im AppStore.

# Vorwort

Die Idee und das Konzept zu dieser Bachelorarbeit habe ich gemeinsam mit meinem Betreuer Michael Stach erarbeitet. So ziemlich alle Themen waren Neuland für mich. Mobile Programming spielt in meinem Studiengang keine Rolle und auf Konzepte der Usability und User Experience wurde im Verlauf meines Studiums wenig eingegangen. Um so interessanter, aber auch arbeitsintensiver war die Aneignung der notwendigen Kenntnisse und Fähigkeiten. Im Nachhinein bin ich sehr dankbar, dass ich ein Thema für die Ausarbeitung der Bachelorarbeit gewählt habe, bei dem die App-Konzeptionierung auf mobilen Endgeräten im Vordergrund stand. Ein Traum von mir ist es, eines Tages eine eigene App zu veröffentlichen. Diesem Traum bin ich nach den sieben Kapiteln dieser Arbeit sehr viel näher gekommen.

Bei der Arbeit mit Flutter fühlt man sich angesichts der Neuheit des Frameworks wie ein Pioneer. Es wird sich zeigen ob plattformübergreifende App-Entwicklung langfristig einen Durchbruch erleben wird. Für kleinere Projekte ist die Technologie meiner Meinung nach ein ungeheurer Mehrwert, da der Aufwand eine App für die beiden großen mobilen Betriebssysteme zu entwickeln halbiert wird. Für größere Projekte kann ich keine Aussage treffen. Die Idee der Zusammensetzung der Widgets ist ein denkbar einfaches Konzept und gefällt mir. Den deklarativen Programmierstil sehe ich als einen großen Vorteil, weil man schnell sehen kann, welche Eigenschaften das jeweilige Widget besitzt. Fehlt ein gewünschtes Attribut, wrapt man das Widget einfach mit demjenigen, das diese Eigenschaft besitzt und schon ist das Problem gelöst.

Abschließend möchte ich mich gerne bei meinem Betreuer für seine Unterstützung und konstruktive Kritik während meine Arbeit bedanken. Ohne ihn wäre diese Bachelorarbeit nicht in diesem Maße entstanden. Außerdem möchte ich an dieser Stelle meine Freunde und meine Familie erwähnen, die mich in dieser besonders intensiven Zeit unterstützt haben.

Diese Bachelorarbeit stellt lediglich meinen persönlichen Anfang in der Programmierung mobiler Anwendungen dar. Flutter und Usability werden mich weiter im Leben begleiten.

# Inhaltsverzeichnis

# 1 Einleitung

Die Anzahl der Anwendungen für mobile Endgeräte ist über die letzten Jahre stark gewachsen. [32] Es ist zu erwarten, dass die Anzahl weiter zunehmen wird, weil die Anzahl der mobilen Endgeräte ein starke Wachstumsrate zu verzeichnen hat. [24] Die Gründe hierfür liegen klar auf der Hand:

- Einerseits haben viele Entwicklungsländer die Phase des Ausbaus teurer Glasfasernetze bis direkt in das eigene Wohnzimmer übersprungen und streben jetzt mit großen Schritten auf die Vernetzung mittels mobiler Verbindungen zur Internetnutzung zu. Dies ist gleichbedeutend mit einer reinen quantitativen Erhöhung der Endnutzer aufgrund der Technologie eines neuen Links. [34]
- Andererseits müssen sich Einrichtungen und Unternehmen, falls sie mit der Entwicklung der Technologie mitgehen wollen, den Präferenzen der Benutzer anpassen. Die Erfahrung großer Internetversandhändler zeigt beispielsweise, dass Kunden signifikant mehr Zeit damit verbringen, sich Waren über eine App anzuschauen, als über eine mobile Website. Interessanter noch: Die Kaufbereitschaft der Endkonsumenten ist bei mobilen Apps anderthalb mal höher als bei Desktopanwendungen und sogar dreimal höher im Vergleich zu mobilen Browserseiten. [1, 2]

Selbst bei der Behandlung von Krankheiten können heutzutage Anwendungen für mobile Endgeräte helfen. Eine fundamentale Grundvoraussetzung für die Funktionstüchtigkeit dieser Apps ist die Möglichkeit der Dateneingabe. Nur auf diese Weise kann die App Daten generieren, die dem Anwender langfristig helfen.
2013 wurde TrackYourTinnitus als universitäres Projekt entworfen. Es existieren eine Webseite mit wichtigen Informationen über das Projekt und Applikationen für Smartphones der beiden großen Plattformen iOS und Android. Das Projekt mit den TrackYourTinnitus Apps hat zum Ziel, Daten über die Betroffenen und ihre Krankheit

zu sammeln. Auf diese Weise kann die Tinnitus Research Initiative, die hinter dem Projekt steht, Erkenntnisse aus dem Forschungsprojekt gewinnen und die Krankheit besser verstehen. Ebenso ist die Anwendung für den Benutzer ein nützliches Kontrollmittel für den eigenen Tinnitus.

Auf der Tack Your Tinnitus Webseite des Instituts für Datenbanken und Informationssysteme der Universität Ulm (DBIS) sind die fortschreitenden Forschungen an dem Projekt anhand der Vielzahl der Veröffentlichungen ersichtlich.[1] Diese Bachelorarbeit mit dem Titel *Entwicklung progressiver Eingabekonzepte zur Datenerfassung auf mobilen Endgeräten* reiht sich dort ein. Wie im Titel erkenntlich ist, liegt der Fokus hierbei jedoch nicht auf der Erforschung des Tinnitus, sondern auf der Erfassung der Daten über den Tinnitus.

## 1.1 Problemstellung

Neue Technologien tauchen in der Informatik fortwährend auf. Für bestehende Apps bedeutet das, dass überprüft werden muss, ob die neuen Technologien auch für sie geeignet sind.
Die im AppStore verfügbare TrackYourTinnitus App ist mit ihrem letzten Update von vor zwei Jahren obsolet. Darüberhinaus sind in der App wenig Elemente enthalten, die die Benutzung der Anwendung zu einem Erlebnis macht.

Die Dateneingabe der Anwendung gleicht dem Ausfüllen eines Fragebogens auf Papier. Die Möglichkeiten, die die Schnittstelle *mobiler Endgeräte* offerieren, finden wenig Aufmerksamkeit. Auch wenn die Daten in erster Linie dem Zweck der Wissenschaft dienen sollen, können durch *progressive Eingabekonzepte* die Freude an der Benutzung der App erheblich gesteigert werden.

## 1.2 Zielsetzung

In der vorliegenden Bachelorarbeit sollen progressive Eingabekonzepte entwickelt werden, um die Datenerfassung der bestehenden Anwendung auf mobilen Endgeräten zu verbessern. So soll ermöglicht werden, dass die TrackYourTinnitus App

[1] https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/trackyourtinnitus/

mit neuen Merkmalen ausgestattet wird. Ziel ist es, dem Nutzer eine ansprechende Oberfläche zur Verfügung zu stellen, in der die Datenerfassung zum Erlebnis wird. Auch die veraltete App-Navigation soll dabei moderner gestaltet werden.

Mit Flutter ist es möglich fortschrittliche Benutzeroberflächen zu entwickeln. Das Tool Flutter soll im Rahmen dieser Bachelorarbeit ausgiebig analysiert und getestet werden. Das TrackYourTinnitus Projekt stellt mit seinem veralteten Eingabekonzept ein geeignetes Ziel der Verbesserung dar.

## 1.3 Struktur der Arbeit

Um die genannten Probleme effizient zu beheben und damit die geforderten Ziele umzusetzen, soll zunächst die TrackYourTinnitus App im Kapitel 2 der Verwandten Arbeiten analysiert werden. Dabei soll auf die aktuellen Eingabekonzepte zur Datenerfassung eingegangen werden.
Anschließend werden im Kapitel 3 die Grundlagen von Flutter erläutert. Der Programmierstil der mit der Nutzung des Frameworks einhergeht und die Zustandsverwaltung wird dargelegt. Auf die besonderen Aspekte der Usability wird in dem Kapitel 4 eingegangen. Anschließend wird die TrackYourTinnitus App neu entworfen. Der iterative Entwurfsprozess vom Papier bis zur finalen Codeversion wird in Kapitel 5 dokumentiert.
Im Kapitel 6 der Implementierung wird der Code des finalen Entwurfs stückweise publiziert und erläutert. Die Überführung der Gedanken vom Papier in Code werden dann deutlich sein.
Schließlich wird in Kapitel 7 ein Fazit gezogen. Die Arbeit wird zusammengefasst und es wird ein Ausblick für die weitere Entwicklung der TrackYourTinnitus App gegeben.

# 2 Verwandte Arbeiten

Das TrackYourTinnitus Projekt wurde ins Leben gerufen, um die Symptome einer Tinnituserkrankung systematisch über mehrere Wochen verfolgen zu können. Betroffene Personen wissen oft, wann der Tinnitus etwa zeitlich auftritt. Ohne eine dauerhafte Aufzeichnung der Ausprägung sind jedoch keine wissenschaftlichen Aussagen möglich. Ein Zusammenschluss zwischen der Universität Ulm, der Universität Regensburg und der Otto-von-Guericke Universität Magdeburg forscht und arbeitet aus diesem Grund seit mehreren Jahren an dem Projekt. [8, 18]

## 2.1 Überblick über die TrackYourTinnitus Umgebung

Damit die Erhebung der Daten möglich ist, existieren Apps für iOS und Android und eine Browser-Variante. Der Ablauf der Erhebung der Fragebogendaten und die Einstellungsmöglichkeiten innerhalb der mobilen Anwendung sind in Abbildung 2.1 zu sehen. Der User hat die Wahl zwischen einer Browser- und einer mobilen Variante des Programms. Er muss jeweils die Registrierung durchlaufen, seine Email Adresse bestätigen und sich dann anmelden bzw. in der App einloggen.
Danach beginnt der Teil der Dateneingabe. Der Patient muss zunächst Fragen zur eigenen Person beantworten. Hat er sie ausgefüllt, gelangt er zu dem eigentlichen Standardfragebogen über die Tinnitusbeschwerden, die beobachtet werden sollen. Der Standardfragebogen ist nur in der mobilen Anwendung verfügbar. Neben den mobilen Anwendungen gehört auch noch der Server dazu. Die Clients greifen allerdings nicht direkt auf die Informationen auf dem Server zu. Eine REST-ähnliche JSON-API wickelt die Kommunikation zwischen App und Browseranwendung. Das Datenmodell, das in TrackYourTinnitus angewendet wird, stützt sich auf ein relationales Datenmodell. Diese ist auf dem Server in MySQL implementiert. [7, 16]

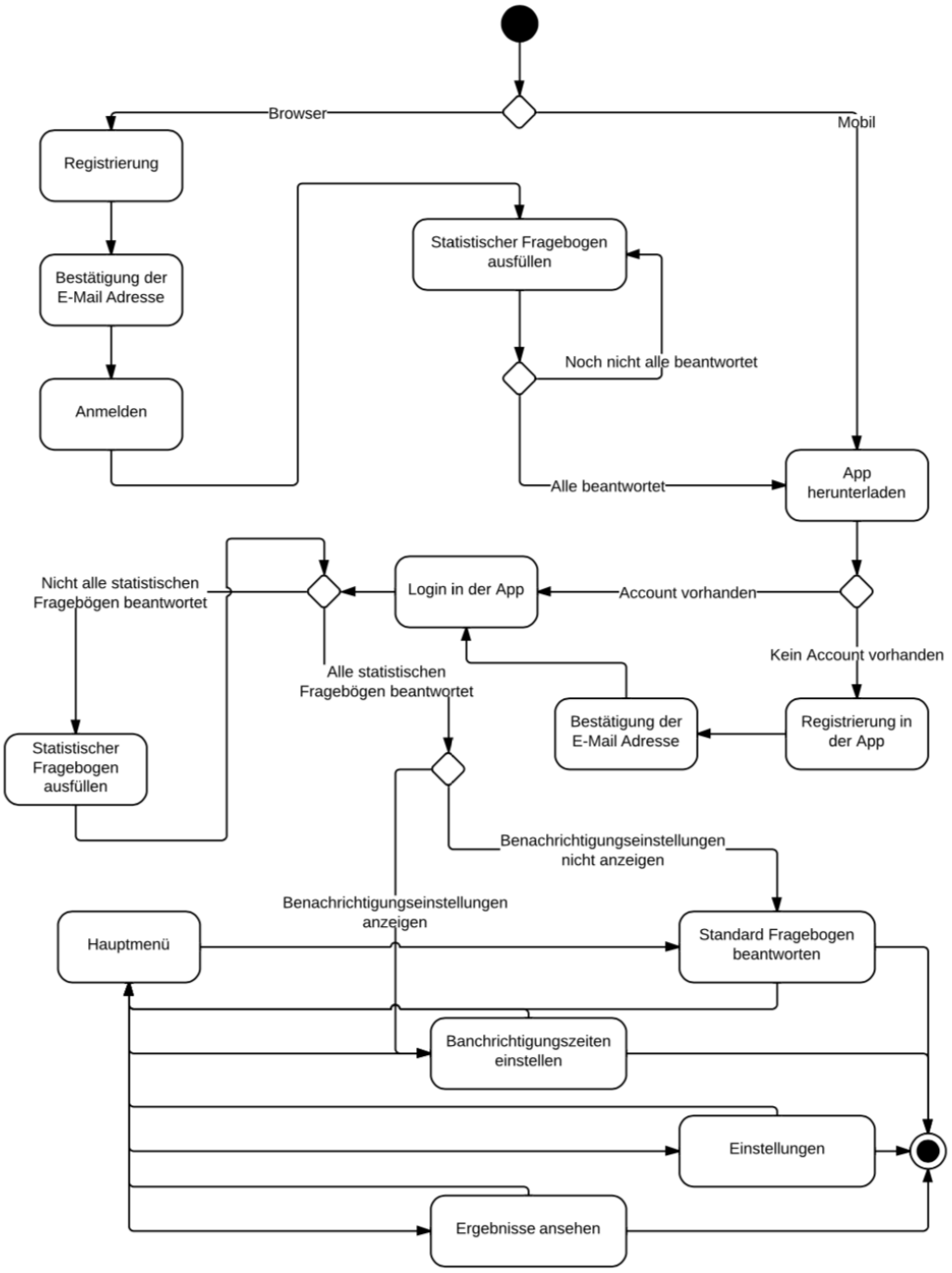



Abbildung 2.1: Technischer Ablauf der TrackYourTinnitus Datenerhebung. Quelle: [7]

## 2.2 Dateneingabe in TrackYourTinnitus

Das Eingabekonzept zur Datenerfassung dieser Bachelorarbeit stützt sich auf das Datenmodell von TrackYourTinnitus und die Möglichkeit wie dort Daten erfasst werden. Die App die im AppStore heruntergeladen werden kann, umfasst die folgenden acht Fragen. [15, 17]

- Haben Sie gerade den Tinnitus bewusst wahrgenommen?
- Wie laut ist der Tinnitus momentan?
- Wie belastend empfinden Sie den Tinnitus im Moment?
- Wie ist Ihre aktuelle Stimmungslage?
- Wie aufgeregt sind Sie gerade?
- Wie gestresst fühlen Sie sich gerade?
- Wie sehr haben Sie sich auf das konzentriert, was Sie gerade tun?
- Fühlen Sie sich gerade gereizt?

Die erste Frage erfordert vom Patienten eine binäre Eingabe, da er sich entscheiden muss, ob er den Tinnitus wahrgenommen hat oder nicht. Das aktuelle UI-Element für die Dateneingabe ist in Abbildung 2.2 zu sehen.

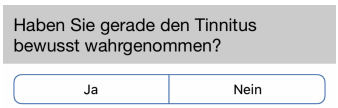



Abbildung 2.2: Binärer Antworttyp

Die Abbildung 2.3 zeigt den zweiten, dritten, sechsten und siebten Eingabetyp für die Datenerfassung im Fragebogen. Der Patient soll mithilfe eines Sliders seinen aktuelllen Wert zwischen Minimal- und Maximalwert eingeben.

Bezüglich der Fragen vier und fünf muss der Anwender seine Eingaben über den emotionalen Zustand auf neun Radiobuttons verteilen. Die Radiobuttons schließen sich bei der Eingabe aus, sodass nur ein Wert markiert werden kann. Abbildung 2.4 zeigt das UI-Element.

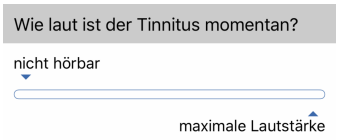



Abbildung 2.3: Slider Antworttyp

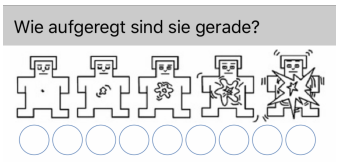



Abbildung 2.4: Radio-Button Antworttyp

Zu erwähnen ist, dass alle Skalen initial keine voreingestellten Werte anzeigen. Dies ist beabsichtigt, da es sich bei dem TrackYourTinnitus Projekt um die Eingabe subjektiver Daten handelt. Für die Eingabeelemente bedeutet das, dass sie den Benutzer nicht beeinflussen sollen. Er könnte durch etwaige Voreinstellungen gelenkt sein und einen Wert in unmittelbarer Nähe des voreingestellten Bereiches wählen.

# 3 Flutter

Flutter ist ein von Google entwickeltes Framework mit dem plattformübergreifend Anwendungen geschrieben werden können. Nach der Einführung des iPhones im Jahr 2008 und dem Erfolg der Smartphones haben sich langfristig zwei Plattformen auf dem Markt der mobilen Endgeräte entwickelt. Auf der einen Seite existiert Apples Plattform mit seinem mobilen Betriebsystem iOS und auf der anderen Seite ist Google mit dem mobilen Betriebssystem Android nachgezogen. Während diese Bipolarität im Softwarebereich der Entwicklung neuer Features und vor allem den Endverbrauchern zu Gute kommt, erschwert es Unternehmen und Entwicklern das Leben dahingehend, dass für beide Plattformen unterschiedliche Programmierkenntnisse gebraucht werden und mit der Programmierung einer nativen Anwendung nicht alle Benutzer erreicht werden können. Aus diesem Grund arbeiten seit einiger Zeit große Softwarefirmen wie Facebook mit React Native[1] oder auch Google mit Flutter daran, die Entwicklung von Apps für beide Plattformen auf eine Codebasis zu reduzieren. [11]

Flutter wird mit Googles hauseigener Programmiersprache Dart bedient und basiert im Grunde darauf einen Widget-Baum aufzubauen. Hierbei wird ein deklarativer Ansatz gewählt wird, um die Benutzeroberfläche aufzubauen.

**Definition 3.0.1.** *Flutter* ist ein von Google entwickeltes Framework um nativ kompilierbare Anwendungen anhand einer einzigen Codebasis zu schreiben.

**Definition 3.0.2.** *Dart* ist eine von Google entwickelte Computersprache. Ihr ursprüngliches Ziel war es JavaScript zu ersetzen und so zur neuen Sprache des Webs zu werden. Dennoch liegt ihr Fokus im Moment eher darauf, in JavaScript Code umgewandelt zu werden, sowie auf der Entwicklung von Multiplattformprogrammen. [33]

[1] https://facebook.github.io/react-native/

**Definition 3.0.3.** Eine *Deklarative Benutzeroberfläche* beruht auf einer unveränderlichen View-Konfiguration. In Flutter werden für die Konfiguration der Oberfläche Widgets benutzt. Eine Manipulation der Benutzeroberfläche wird durch eine Neuinstanziierung der entsprechenden Widgets eingeleitet. [26]

**Definition 3.0.4.** Im Kontext des Flutter-Framework ist ein *Widget* ein Objekt mit Eigenschaften, das zur Darstellung einer deklarativen Benutzeroberfläche benutzt wird.

Um Flutter zu verstehen, ist eine fundierte Kenntnis des Widget-Baums der intern aufgebaut wird und die Handhabung der User-Interface-Elemente (UI-Elemente) essenziell. Daher wird dieser Theorieteil gesondert in den Abschnitten 3.3.1 bis 3.3.3 erarbeitet. Aus Definition 3.0.3 wird ersichtlich, dass jedes Widget ein mehr oder weniger kleiner Bestandteil der Gesamtbeschreibung der finalen Anwendungsoberfläche ist. Es wird nicht wie bei dem bekannten *Model-View-Controller-Patter* (MVC-Pattern) ein großes View-Objekt erstellt, welches dann sukzessive durch Methoden verändert wird. Das Listing 3.1 zeigt ein kleines Beispiel nach dem klassischen MVC-Pattern. Im Gegensatz dazu, wird Flutter-Code wie in dem Beispiel des Listings 3.2 konstruiert.

Listing 3.1: Imperativer Stil

```
1 //imperativer Stil
2 b.setColor(red)
3 b.clearChildren()
4 ViewC c3 = new ViewC(...)
5 b.add(c3)
```

Listing 3.2: Deklarativer Stil

```
1 //deklarativer Stil
2 return ViewB(
3         color: red,
4         child: ViewC(...),
5 )
```

Widget *ViewB* enthält das verschachtelte Widget *ViewC* als Kind im Konstrukt des gesamten Widget-Baums. Ebenso wird die Farbe rot direkt als Eigenschaft deklarativ im Widget *ViewB* festgelegt, ohne dass dafür eine Extramethode nötig wäre.

## 3.1 Abgrenzung zu anderen Frameworks

Mit der Benutzung des Flutter-Frameworks gehen mehrere Vorteile einher. Wie zuvor erwähnt, liegt der besondere Fokus darauf, eine einzige Codebasis für mehrer Plattformen aufzubauen. Dies spart im optimalen Fall Zeit und Geld. Merkmale wie *hot reload* erlauben es dem Entwickler schnell und einfach Oberflächen zu bauen, neue Merkmale hinzuzufügen und direkt erscheinen zu lassen, oder aber auch Bugs schneller zu finden.

**Definition 3.1.1.** Beim *hot reload* werden die Änderungen im Quellcode an die Dart Virtual Machine (VM) direkt übertragen. Nachdem die VM die betroffenen Klassen mit den neuen Variablen und Methoden aktualisiert, wird der Widget-Baum von Flutter automatisch neu aufgebaut, sodass die Änderungen *sofort* in der Benutzeroberfläche zu sehen sind. Dabei werden nur Widgets ausgeführt, die von Code-Änderungen betroffen sind. Emulatoren oder echte Android bzw. iOS Hardwaregeräte können so die neue Code-Version sofort nachladen und verlieren nicht ihren Zustand. [25]

Flutter ist seit Dezember 2018 stabil als Download auf dem Markt der plattformübergreifenden Software verfügbar.[2] Dadurch ist die Programmierschnittstelle (API) gespickt mit *modernen* Widgets die ein großes Augenmerk auf Animationen, *Motion* und *Styling* wie dem *Theme Widget* legen. Googles Material Design und iOS' Cupertino Design werden konsequent durchgesetzt.
Native Performance hat bei Flutter hohe Priorität. Alle plattformkritischen Unterschiede wie Scrollen, Seitennavigation, Icons oder Schriften werden nativ sowohl auf Android als auch auf iOS voll unterstützt. Die Performance wird mittels eines Schichtenmodells erreicht, welches in Abbildung 3.1 zu sehen ist. Jede Schicht baut dabei auf der jeweils unteren auf, wobei die hören Schichten wesentlich häufiger benutzt werden als die unteren. Beispielsweise setzt sich die Material-Schicht aus einfacheren Grundwidgets zusammen und die Widget-Schicht selber wird wiederum von der *Rendering*-Schicht zusammen gebaut usw. Durch diese Herangehensweise ist es sehr einfach die schon bestehenden Flutter Widgets zu verwenden, oder auch maßgeschneiderte Widgets nach dem selben Prinzip effizient zu bauen. Die aggressive Zusammensetzung der Widgets ist ein einfaches Konzept und

[2] https://en.wikipedia.org/wiki/Flutter_(software)#History

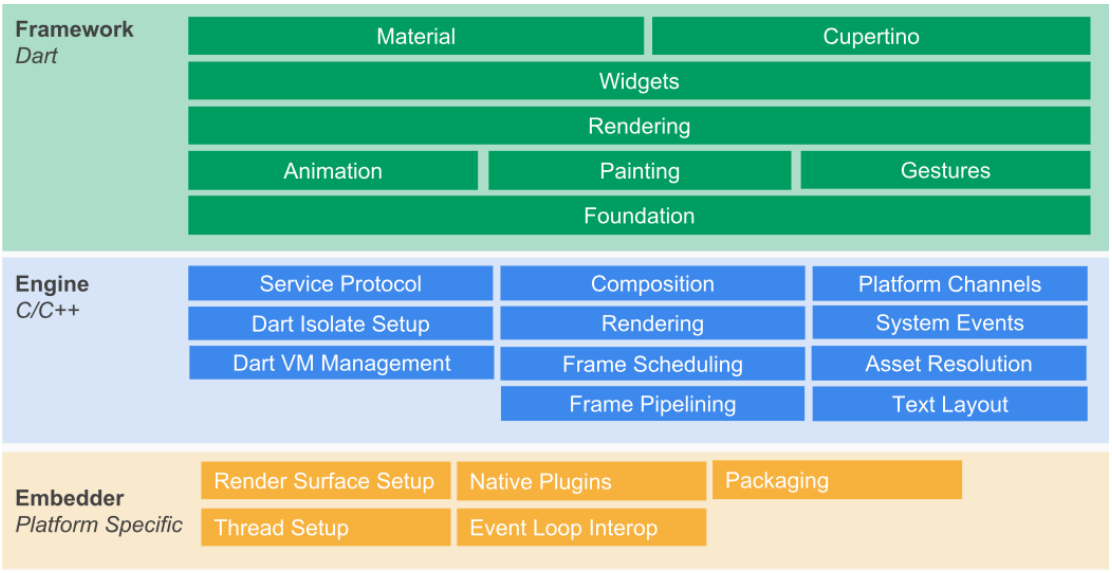



Abbildung 3.1: Schichtenübersicht über das Flutter Framework [4]

wird konsequent durchgesetzt. Dart ist eine streng typisierte Programmiersprache wodurch letztendlich der Dartcode in eine C/C++ Library kompiliert werden kann. Diese kann in der Folge von den jeweiligen Plattformen leicht weiterverarbeitet werden. Eine JavaScript Bridge wie es bei React Native der Fall ist, wird hier nicht gebraucht. Diese Prinzip beschleunigt die Kompilierung bei Flutter. [19]

## 3.2 Widget

Flutter Anwendungen setzen sich aus zwei Arten von Widgets zusammen. Zum einen gibt es die *Stateless-Widgets* und zum anderen existieren die *Stateful-Widgets*. Beide Widget-Arten sind *immutable*, also unveränderlich. In Code ausgedrückt bedeutet dies, dass alle Eigenschaften die ein Widget tragen kann, als *final* markiert werden können. Integrierte Entwicklungsumgebungen (IDE) wie *Visual Studio Code*, die das Flutter-Framework unterstützen bieten *Plug-Ins* an, die sofort farblich anmerken falls das Setzen des Schlüsselworts *final* in einer Widget-Klasse vergessen wird.

**Definition 3.2.1.** *Integrierte Entwicklungsumgebung* Eine Integrierte Entwicklungsumgebung bündelt die wichtigsten Werkzeuge zur Erzeugung von Software in ei-

nem Programm. Typischerweise gehören Editor, Compiler, Interpreter, Linker, Debugger und Versionsverwaltungen dazu. [10]

Ein weiteres Merkmal von Flutter ist, dass jedes Widget eine `build()`-Methode erhält. Um dem Leser dieser Arbeit einen ersten Eindruck vom Flutter-Framework zu ermöglichen, werden in den nächsten Abschnitten Teilquellcodes betrachtet. Die Benutzeroberfläche die von dem gesamten Quellcode erzeugt wird ist in Abbildung 3.2 zu sehen und erzeugt im wesentlichen ein kleines Programm, das zählt wie häufig der sich unten rechts befindende Button gedrückt wurde.

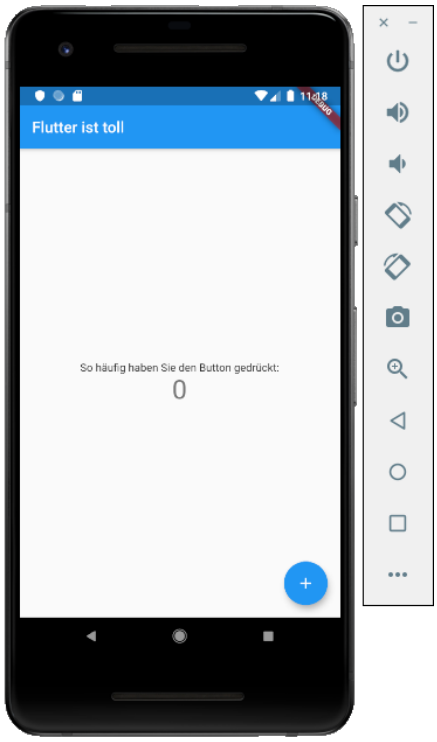



Abbildung 3.2: UI mit Stateless- und StatefulWidget

### 3.2.1 Stateless Widget

Wie in vielen anderen Programmiersprachen, wird auch bei Flutter ein Programm mit einer `main()`-Methode gestartet.

```
void main() => runApp(MyApp());
```

Der `main()`-Methode wird ein weiterer Ausdruck mittels der *Fat-Arrow-Notation* übergeben, in der hingegen das *StatelessWidget* MyApp instanziiert wird.

**Definition 3.2.2.** Die *Fat-Arrow-Notation*, oder auch *Short-Hand-Syntax* wird benutzt um Methoden mit jeweils nur einer Anweisung im Methodenrumpf kürzer zu schreiben. Der Aufbau dieser Notation erfolgt nach dem Muster `WiedergabeTyp funktionsName(Parameter...) => Ausdruck;`. Die geschweiften Klammern entfallen.

Listing 3.3: StatelessWidget

```
1  class MyApp extends StatelessWidget {
2    @override
3    Widget build(BuildContext context) {
4      return MaterialApp(
5        title: 'Flutter',
6        theme: ThemeData(
7          primarySwatch: Colors.blue,
8        ),
9        home: MyHomePage(title: 'Flutter ist toll'),
10     );
11   }
12 }
```

Die Klasse MyApp im Listing 3.3 ist ein klassisches StatelessWidget, da es keinen State enthält und unveränderlich ist. Hier in diesem Fall besitzt das Widget keine weiteren Eigenschaften, jedoch wird in seiner `build()`-Methode ein weiteres Widget aufgerufen, dass über einen Konstruktor instanziiert wird. Manche Widgets besitzen viele Eigenschaften, von denen aber nicht immer alle zu jedem Zeitpunkt benötigt werden. Dart stellt daher eine geschickte Möglichkeit zur Verfügung, immer genau die Argumente zu verwenden, die man jeweils für die Anwendung eines spezifischen Widgets braucht.

**Definition 3.2.3.** *Named arguments* können in beliebiger Reihenfolge aufgerufen werden, somit muss sich der Entwickler nicht darum kümmern, sie genau in der von dem Widget vorgegebenen Reihenfolge aufzurufen, bzw. kann sie, sofern sie nicht im Konstruktor mit der *@required*-Annotation markiert sind, nach freien Belieben verwenden. Named arguments werden im Konstruktor mit geschweiften Klammern als solche markiert.

In dem MaterialApp-Widget sind `title`, `theme` und `home` named arguments. Dem `home`-Argument wird das StatefulWidget *MyHomePage* aus Listing 3.4 übergeben.

### 3.2.2 StatefulWidget und State

Wie in Abschnitt 3.2 erwähnt sind auch StatefulWidgets *immutable*, also unveränderlich.

Listing 3.4: StatefulWidget

```
1 class MyHomePage extends StatefulWidget {
2   MyHomePage({Key key, this.title}) : super(key: key);
3   final String title;
4   @override
5   _MyHomePageState createState() => _MyHomePageState();
6 }
```

Das String-Attribut `title` in Zeile 3 im Listing 3.4 ist mit `final` markiert, ansonsten befindet sich noch ein Konstruktor und die notwendige `createState()`-Methode in der Klasse. Diese Methode erzeugt das zum Widget zugehörige *State-Objekt*.

**Definition 3.2.4.** Ein *State* ist eine Klasse in Flutter die Variablen enthält, die von dem Framework zur Laufzeit verändert werden können. Auf diese Weise können verschiedene Zustände von Widgets auf dem Display gerendert werden. State und Stateful-Widget hängen über eine Referenz zusammen. [29]

In Listing 3.5 wird ersichtlich, dass ein State vom Typ der StatefulWidget-Klasse erzeugt wird und State und StatefulWidget maßgeblich über die Typisierung zusammenhängen.

Listing 3.5: State

```
1  class _MyHomePageState extends State<MyHomePage> {
2    int _counter = 0;
3    void _incrementCounter() {
4      setState(() {
5        _counter = _counter + 1;
6      });
7    }
8    @override
9    Widget build(BuildContext context) {
10     return Scaffold(
11       appBar: AppBar(
12         title: Text(widget.title),
13       ),
14       body: Center(
15         child: Column(
16           mainAxisAlignment: MainAxisAlignment.center,
17           children: <Widget>[
18             Text(
19               'So haeufig haben Sie den Button gedrueckt:',
20             ),
21             Text(
22               '$_counter',
23               style: Theme.of(context).textTheme.display1,
24             ),
25           ],
26         ),
27       ),
28       floatingActionButton: FloatingActionButton(
29         onPressed: _incrementCounter,
30         tooltip: 'Increment',
31         child: Icon(Icons.add),
32       ),
33     );
34   }
35 }
```

In der State-Klasse können nun Variablen erstellt werden die nicht als `final` deklariert werden müssen, so wie es bei `_counter = 0` in Zeile 2 vorzufinden ist. Andernfalls könnte sie nicht in der `_incrementCounter()`-Methode hochgezählt werden. Als Gemeinsamkeit zum StatelessWidget folgt die `build()`-Methode, in der weitere Widgets deklariert werden. Das besondere an der State-Klasse ist die `setState()`-Methode in Zeile 4 die innerhalb des Methodenrumpfs der `_incrementCounter()`-Methode aufgerufen wird. Diese erlaubt es Flutter neue Zustände von Widgets zu registrieren und wie im Fall des Code aus Listing 3.5 den Zähler auch auf dem Smartphone-Bildschirm zu aktualisieren.

## 3.3 Widget-, Element- und Render-Baum

Aus technischer Sicht werden zur Erstellung von Flutter-Apps drei Bäume aufgebaut. Diese werden in den folgenden Abschnitten genauer erklärt. Außerdem wird gezeigt, was die Aufgabe des jeweiligen Baums ist und wie alle drei miteinander zusammenhängen.

### 3.3.1 Widget-Baum

Ein *Widget-Baum* besteht aus den in Abschnitt 3.2.1 und 3.2.2 vorgestellten Widgets. Sie beschreiben die Benutzeroberfläche und legen die Konfiguation des Programms fest. Das Wurzelelement des Widget-Baums wird mit der Methode `runApp()` festgelegt. Daraus folgt, dass man aus technischer Sicht ein Flutter-Programm als Widget-Baum bezeichnen kann. [27]

Flutter stützt sich zum Zeitpunkt der Programmierung auf die deklarative Zusammensetzung von Widgets. Diese Widgets werden intern als ein Baum aufgebaut, und bilden eine Blaupause für das Programm. Ein Widget bildet die Wurzel. In der Beispielapplikation aus Abbildung 3.2 war dies das Stateless-Widget *MyApp*. In Abbildung 3.3 wird nun Abschnitt ① betrachtet. Dieser fiktive und zu Demonstrationszwecken klein gewähte Baum beinhaltet als Wurzelelement einen Container-Widget das ein Column-Widget hält. Column-Widgets haben die Eigenschaft, dass sie mehrere Kinder halten können, weil sie als *named arguments* eine Liste von Widgets erhalten. Wie der Name andeutet, werden die Kinder später vertikal auf dem Bildschirm angeordnet. In dem Fall hier hält das Column-Widget ein beliebiges Stateless- und Stateful-Widget. Dieser Zusammensetzung von Widgets bildet nun die Konfiguration für den Widget-Baum den das Framework intern aufbauen kann. [28]

### 3.3.2 Element-Baum

Als nächstes wird Abschnitt ② in Abbildung 3.3 betrachtet. Für jedes Widget im Widget-Baum wird ein äquivalentes Element im Element-Baum angelegt. Mit ande-

ren Worten heißt das, dass während Flutter den Widget-Baum abläuft, durch die Methode `createElement()` auch der Element-Baum aufgebaut wird.

Der *Element-Baum* enthält die logische Struktur einer Benutzeroberfläche. Er ist notwendig, weil die Widgets an sich unveränderlich sind, und selber nicht wissen, wer ihr Vorgänger bzw. Nachfolger ist. Der Element-Baum speichert ebenso die State-Objekte, die mit den Stateful-Widgets über eine Referenz verbunden sind. [3]

Für die Abbildung 3.3 bedeutet dies, dass sukzessive das Containter-Element, dann als Kind das Column-Element und schließlich die beiden Stateless- und Stateful-Elemente als Kinder des Column-Elemente aufgebaut werden. Wie oben erwähnt erhält das Stateful-Element noch eine Referenz auf das State-Objekt. Ebenso verweisen alle Elemente auf ihr jeweiliges Widget. Während die Widgets beispielsweise in dem Code-Listing 3.5 schnell zu erkennen sind, muss dem Leser bewusst sein, dass die Elemente in dem jeweiligen *BuildContext* versteckt sind.

```
@override
Widget build (BuildContext context) {
   return ...;
}
```

Aus diesem Grund erhält der Element-Baum für jedes Widget ein Element.

### 3.3.3 Render-Baum

Aus Abbildung 3.3 Bereich ③ geht hervor, dass letztendlich ein dritter Baum aufgebaut wird.

Der *Render-Baum* ist ein maschinennahes Layout und Zeichensystem, das auf Render-Objekten basiert. Der Render-Baum wird mithilfe von Widgets aufgebaut, weshalb üblicherweise ein Flutter-Entwickler nicht mit dem Render-Baum interagiert. [13]

Dadurch dass die Methode `createRenderObject()` auf die jeweiligen Elemente aufgerufen wird, entsteht die Baumstruktur des Render-Baums ③. In dem vorliegendem Fall werden so die einzelnen Render-Objekte aufgebaut. Sie enthalten die notwendige Logik um das jeweilige Widget auf dem Gerätebildschirm darzustellen.

Eine Instanziierung dieser Objekte ist aufwendig, daher versucht das Framework sie so lange wie möglich im Speicher zu behalten.

### 3.3.4 Zusammenspiel der drei Bäume

In den Abschnitten 3.3.1 bis 3.3.3 werden die drei Bäume die Flutter zur Darstellung eines Programms braucht separat eingeführt. Dennoch arbeiten sie eng zusammen und dieses Zusammenspiel ist wesentlich für Flutter.
Jedes mal wenn sich der Widget-Baum ändert, vergleicht der Element-Baum den aktuellen Widget-Baum mit den Render-Objekten. Wenn nun der Typ des jeweiligen Widgets der gleiche ist wie vorher, dann erstellt Flutter für dieses kein neues Render-Objekt. Dieser Vorgang ist positiv, weil die Render-Objekte *teuer* zu instanziieren sind. Widgets hingegen sind günstig in ihrer Instanziierung und eigenen sich aus diesem Grund zur Beschreibung der aktuellen Konfiguration der Anwendung.

Angenommen der Container in der Abbildung 3.3 besitzt zunächst die Farbe gelb und wird dann in grün geändert. Dann wird in der Folge der gesamte Widget-Baum neu aufgebaut, weil die `build()`-Methode des Containers aufgerufen wird und Widgets unveränderlich sind. In anderen Worten heißt das, dass der Container als neues Widget instanziiert wird, anstatt nur seine Farbeigenschaft zu aktualisieren. Als nächstes vergleicht Flutter mithilfe der Elemente das erste Widget mit dem ersten Render-Objekt, danach das zweite Widget mit dem zweiten Render-Objekt und so weiter. Man erkennt, wieso die Elemente sowohl eine Referenz auf die Widgets, als auch auf die Render-Objekte benötigen. Flutter prüft ob der alte *Widgettyp* mit dem neuen übereinstimmt. Ist das nicht der Fall, werden alle drei Objekte, d.h. Widget, Element und Render-Objekt aus den jeweiligen Bäumen entfernt und es werden neue Instanzen eingefügt. Wenn der *Typ* des Widgets gleich bleibt, dann aktualisiert das Framework nur die neue Konfiguration des Widgets in den Render-Objekten und spart sich so die teure Instanziierung. Dieses Vorgehen findet statt, bis der gesamte Widget-Baum abgelaufen ist.

Das Widget wurde neu instanziiert, bei dem Render-Objekt, also das was tatsächlich auf dem Bildschirm zu sehen ist, aktualisiert das Framework lediglich die Farbeigenschaft und zeichnet es erneut.

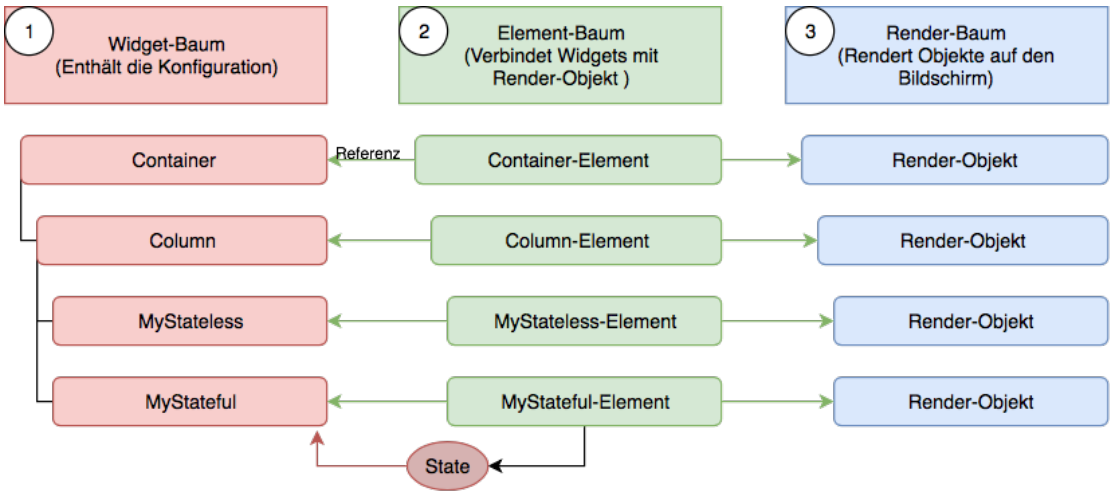



Abbildung 3.3: Widget-Baum, Element-Baum und Render-Baum

Würde das Container-Widget zum Beispiel in ein Card-Widget[3] geändert werden, dann würde das Element an der ersten Stelle im Baum feststellen, dass der alte Typ Container nicht zum Typ Card passt. Alle drei Instanzen in den Bäumen würden durch eine Neuinstanziierung ausgetauscht. Bei dieser Operation gibt es für Flutter also nicht den Ausweg das Render-Objekt im Speicher zu halten und es lediglich anders zu zeichnen, sondern das Framework muss es erstmal neu erstellen. Den Vergleich zwischen den Objekten löst Flutter sehr effizient, weshalb es sich lohnt diese Trinität der Bäume aufzubauen. [22]

## 3.4 Widget Lebenszyklus

Die Programmierung einer Flutter-App funktioniert über Widgets. Darüberhinaus ist der Ablauf eines Flutterprogramms aus Abschnitt 3.3 bekannt. In diesem Abschnitt wird der Lebenszyklus der beiden Widgetarten beleuchtet. Abbildung 3.4 zeigt den Sachverhalt zusammenfassend in einem Schaubild. Man erkennt, dass Stateless-Widgets einen einfachen Lebenszyklus besitzen, weil sie einmal von der Konstruktor-Funktion instanziiert werden und dann über die `build()`-Methode in die aus Abschnitt 3.3 bekannten Bäume eingefügt werden.

[3] https://api.flutter.dev/flutter/material/Card-class.html

Stateful-Widgets werden ebenso über einen Konstruktor instanziiert. Danach unterscheidet sich jedoch der Verlauf des Lebenzyklus von einem Stateless-Widget, denn es wird die `createState()`- Methode aufgerufen die das *State*-Objekt an der jeweiligen Stelle im Baum initialisiert.[4] Innerhalb des State-Objekt wird nun automatisch die `initState()`- Methode aufgerufen, die das State-Objekt initialisiert. Diese Methode wird nur *genau einmal* aufgerufen und kann mittels der *@override*-annotation überschrieben werden, falls

- bestimmte Daten initialisiert werden müssen die zu dem Widget gehören,
- Eigenschaften initialisiert werden müssen die zu dem Vater-Widget gehören,
- Streams, ChangeNotifiers oder andere Objekte die Daten in dem Widget ändern könnten, angemeldet werden müssen.

Anschließend wird die `build()`- Methode von Flutter ausgeführt, die es dem Framework bzw. dem Entwickler erlaubt die `setState`- Methode aufzurufen, falls das Widget oder das Vater-Widget seine Eigenschaften ändert. Die `didUpdateWidget()`- Methode wird aufgerufen, wenn sich das Vater-Widget ändert und muss im Zuge dessen das aktuelle Widget neu bauen, weil es andere Daten benötigt. Das funktioniert nur wenn der Widget-Typ sich nicht verändert hat. An dieser Methode erkennt man auch, dass das State-Objekt langlebig ist, denn hier werden Daten initialisiert die andernfalls in der `initState()`- Methode initialisiert würden. Danach erfolgt ein *rebuild* des Widgets, weshalb die `build()`- Methode automatisch von Flutter aufgerufen wird. Sie muss aus diesem Grund nicht in der `didUpdateWidget()`- Methode aufgerufen werden. Wird ein State-Objekt nicht mehr benötigt, wird `dispose()` aufgerufen. Dies könnte beispielsweise der Fall sein, falls eine Animation zum Ende kommt und das Widget seinen Zustand nicht mehr verändert.

## 3.5 Keys

Im Listing 3.4 in Zeile 2 wird ein Konstrukt benutzt, dass noch nicht weiter behandelt wurde, in Flutter jedoch ein wichtiges Sprachelement darstellt.

```
MyHomePage({Key key, this.title}) : super(key: key);
```

[4] https://api.flutter.dev/flutter/widgets/StatefulWidget/createState.html

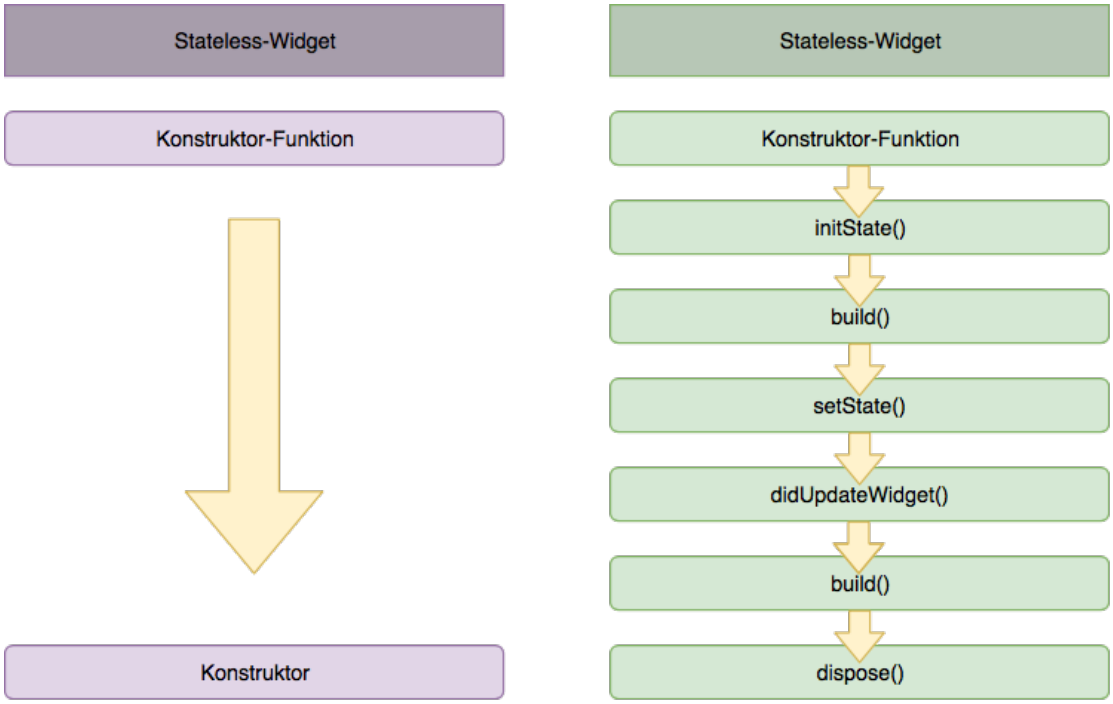



Abbildung 3.4: Widget Lebenszyklus

Das Argument `title` wird im Stateless-Widget benutzt um in der App-Bar den Titel „Flutter ist toll“ zu setzen. Das `key` - Argument wird hingegen nicht benutzt, da es in der Anwendung nicht benötigt wird.

**Definition 3.5.1.** Ein *key* ist ein eindeutiger Identifikator eines Flutter-Widgets. Sobald ein Widget einen key zugeordnet bekommt, werden nicht mehr nur Widget-Typ zwischen Element-Baum und Widget-Baum verglichen, sondern auch der key.

Auch wenn man keys generell bei allen Widgets verwenden kann ist dies bei Stateless-Widgets in der Regel nicht notwendig, weil diese keinen Zustand besitzen. Keys sind immer dann wichtig, wenn Zustände im Widget-Baum zwischen gleichartigen Widgets beim jeweiligen Widget erhalten bleiben sollen. Ein kleines Programm aus Abblidlung 3.5 soll zur Illustration dienen. Es ist an das Kachelproblem von Google-Mitarbeiterin Emily Fortuna angelehnt.[5] In diesem werden zwei unterschiedlich gefärbte Kacheln miteinander vertauscht.

[5] https://gist.github.com/efortuna

### 3.5.1 Collection gleicher Widgets ohne Key

Im Listing 3.6 sehen wir ein erstes Stateful-Widget, dessen State eine Liste mit zwei Kacheln und die `build()`-Methode beinhaltet. Außerdem führt die Methode `swapTiles()` den notwendigen Code aus, um die beiden Kacheln bei einem Klick auf den Floating-Action-Button zu vertauschen.

Listing 3.6: PositionedTiles

```
1  class PositionedTiles extends StatefulWidget {
2    @override
3    State<StatefulWidget> createState() => PositionedTilesState();
4  }
5  class PositionedTilesState extends State<PositionedTiles> {
6    List<Widget> tiles = [
7      StatefulColorTile(1),
8      StatefulColorTile(2),
9    ];
10   @override
11   Widget build(BuildContext context) {
12     return Scaffold(
13       body: Row(children: tiles),
14       floatingActionButton: FloatingActionButton(
15           child: Icon(Icons.sentiment_very_satisfied), onPressed: swapTiles),
16     );
17   }
18   swapTiles() {
19     setState(() {
20       tiles.insert(1, tiles.removeAt(0));
21     });
22   }
23 }
```

In dem Listing 3.7 wird das Stateful-Widget, das für die Kacheln verantwortlich ist, implementiert.

Listing 3.7: StatefulColorTile

```
1  class StatefulColorTile extends StatefulWidget {
2    final int number;
3    StatefulColorTile(this.number);
4    @override
5    _StatefulColorTileState createState() => _StatefulColorTileState();
6  }
7  class _StatefulColorTileState extends State<StatefulColorTile> {
8    Color tileColor;
9    @override
10   initState() {
11     super.initState();
12     tileColor = UniqueColorGenerator.getColor();
13   }
14   @override
15   Widget build(BuildContext context) {
16     return Container(
17       color: tileColor,
18       child: Padding(
19         padding: EdgeInsets.all(70.0),
20         child: Text(
21           'Box Nr. ${widget.number}',
22           style: TextStyle(
23             fontSize: 16,
24           ),
25         ),
26       ),
27     );
28   }
29 }
```

Der Konstruktor instanziiert die Kachel mit der jeweiligen Nummer, die dann auf der Kachel ausgegeben wird. Von Bedeutung ist hier noch die `getColor()`-Methode die in eine Extraklasse ausgelagert ist. Die Methode wird statisch über `UniqueColorGenerator` aufgerufen und erzeugt eine zufällige Farbe, die dann der Kachel zugewiesen wird. In keinem Teil dieses Codes wird während der Laufzeit ein Key verwendet, was sich problematisch auf das Programm auswirkt.
Ein Blick in Abbildung 3.5a lässt erkennen, dass die Anwendung nicht richtig funktioniert. Obwohl Flutter die Beschriftung der Kacheln getauscht hat, ist die Einfärbung der Kacheln gleich. Dieses Verhalten wirkt zunächst anormal, da die Methode `swapTiles()` auf das gesamte `StatefulColorTile()` der Liste `tiles` aus dem Listing 3.6 in Zeile 6 angewendet wird.
Um dieses Phänomen zu klären, muss man sich noch einmal Flutters interne Baumstruktur aus Abschnitt 3.3 vergegenwärtigen. Neben dem Widget-Baum, baut Flutter für jedes Widget ein korrespondierendes Element auf. Die Elemente vergleichen

nun bei einem *Rebuild* ob die Widget-Baum-Struktur noch die gleiche ist. Es wird also lediglich überprüft, ob der vorherige *Widget-Typ* mit dem neuen übereinstimmt. Aufgrund der Tatsache, dass States, wie in Abbildung 3.3 zu sehen ist, mit separaten Referenzen gespeichert werden, baut Flutter die Referenzen zwischen Widget und Element auf, ohne aber die Referenzen der jeweiligen States zu aktualisieren. Das Element mit dem Text „Box Nr. 1 “ zeigt daher nun auf den State des Elemente mit der grauen Einfärbung und das Element mit dem Text-Widget „Box Nr. 2“ auf den State mit dem roten Color-Attribut.

### 3.5.2 Collection gleicher Widgets mit Key

Um das obige Problem zu beheben, stellt Flutter die Möglichkeit des Keys zur Verfügung. Wie oben in der Definition beschrieben, ist es ein eindeutiger Identifikator für ein Widget. Deshalb wird, sobald Keys in Verwendung sind, bei einem Rebuild nicht nur auf Gleichheit zwischen Widget-Typen im Widget-Baum geachtet, sondern zusätzlich auf Gleichheit der Keys geprüft. In der Abbildung 3.5b wird die Funktionalität der Kachel-App durch einen solchen Key gewährleistet. Die rote Box Nr. 1 tauscht hier tatsächlich den Platz mit der grünen Box Nr. 2. Dem Konstruktor aus Listing 3.7 in Zeile 3 wird ein *UniqueKey* mitgegeben, sodass die Zeile wie folgt aussieht:

```
StatefulColorTile(this.number) : super(key: UniqueKey());
```

Die Notation `: super(...)` ist dabei nur ein Verweis auf den Superklassenkonstruktor der Stateful-Widget-Klasse. Bei der Ausführung der Methode `swapTiles()` baut Flutter den Widget-Baum erneut auf. Die Referenzen der Elemente auf die Widgets und States werden jetzt aber nicht nur abhängig von den Widget-Typen gesetzt, sondern auch in Abhängigkeit von dem `UniqueKey`. Dadurch wird der State mitgenommen und die nicht gewollte Referenzierung aus Abbildung 3.5a wird verhindert.

### 3.5.3 Verschiedene Key-Arten

Das Framework stellt Keys für bestimmte Anwendungen zur Verfügung. [5]

**Unique-Key** Ein Unique-Key ist ein eindeutiger Identifikator für ein Widget.

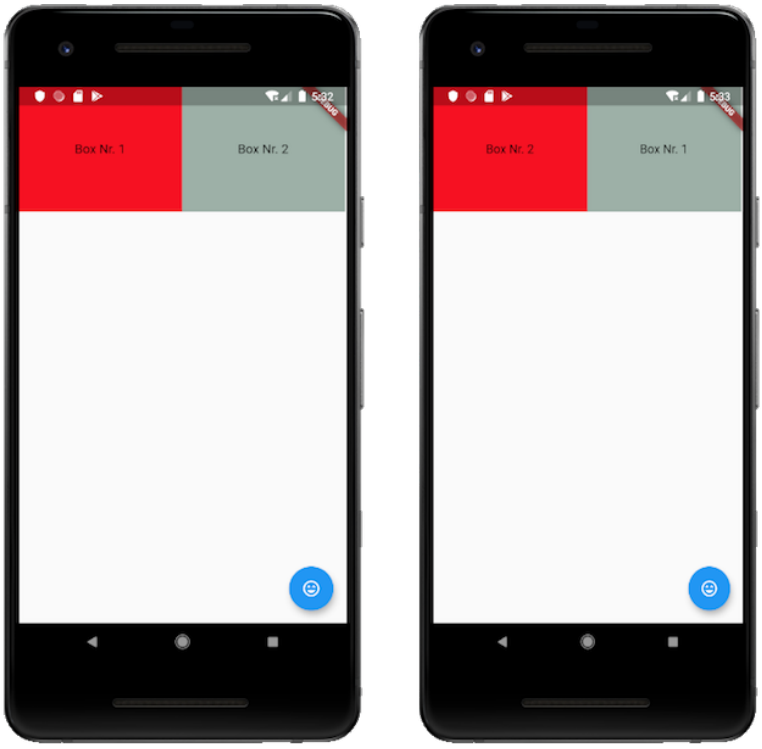
**Value-Key** Der Value-Key wird anhand eines Attributs des Widgets berechnet. Das Attribut wird vom Programmierer im Quellcode festgelegt.

**Object-Key** Der Object-Key wird aus der Kombination mehrer Attribute einer Klasse berechnet.

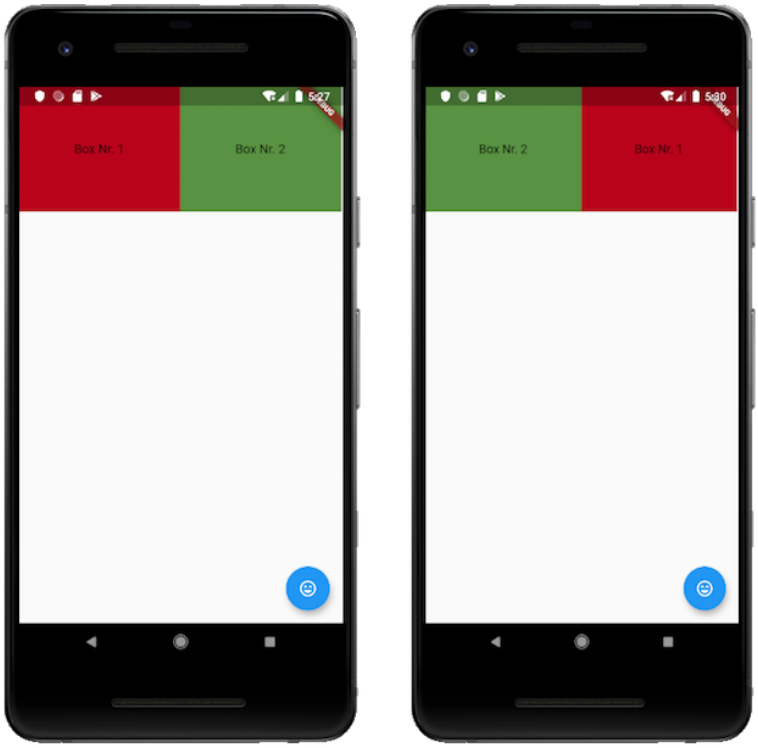
**Page-Storage-Key** Der Page-Storage-Key wird berechnet um die Scroll-Position des Benutzers zu speichern, sodass beim Zurückkehren auf eine vorherige Seite die vorherige Position sofort zugänglich ist.

**Global-Key** Ein Global-Key ist ein Identifikator der über die gesamte Anwendung eindeutig ist.

Wesentliche Bestandteile Flutters – von Widgets bis zu keys – sind in diesem Kapitel erklärt, sodass dem Leser dieser Arbeit ein Verständnis für das Framework vorliegt.




(a) Key nicht implementiert - links Button nicht gedrückt, rechts Button gedrückt.




(b) Key implementiert - links Button nicht gedrückt, rechts Button gedrückt.

Abbildung 3.5: Kachel-App mit und ohne key

# 4 Usability

Usability ist ein englischer Begriff und wird im Deutschen mit Gebrauchstauglichkeit übersetzt. Hierbei wird ein Gegenstand anhand von objektiven Kriterien nach dessen Gebrauchstauglichkeit bewertet. Wenn das zuvor festgelegte Ziel vom Anwender effizient und effektiv erreicht wurde, dann bekommt der Gegenstand der auf Gebrauchstauglichkeit getestet wird, ein positives Feedback vom Nutzer zugeschrieben. Usability wird laut DIN EN ISO 9241-11 folgendermaßen definiert:

**Definition 4.0.1.** „*Usability* ist das Ausmaß, in dem ein Produkt durch bestimmte Nutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen.“ [20]

Durch diese Definition wird deutlich, dass die Gebrauchstauglichkeit eines Produkts durch die *bestimmten* Nutzer, den *bestimmten* Nutzungskontext und die *bestimmten* Ziele klar eingeschränkt ist. Das bedeutet, dass ein und das selbe Produkt für die eine Zielgruppe gebrauchstauglich sein kann, für eine andere Gruppe jedoch keinen, oder nur einen eingeschränkten Nutzen hat.
Begriffe wie Effizienz und Effektivität werden häufig im Alltag benutzt, dennoch muss auch hier eindeutig geklärt sein, was unter den beiden Begriffen zu verstehen ist.

**Definition 4.0.2.** *Effizienz* ist das Verhältnis zwischen Ergebnis und Aufwand. Sie misst inwiefern eine Maßnahme zielgerichtet ist, ein vorgegebenes Ziel auf eine bestimmte Art und Weise zu erreichen. [21]

**Definition 4.0.3.** *Effektivität* ist das Verhältnis zwischen Ergebnis und Ziel. Hierbei handelt es sich darum zu beurteilen, inwieweit eine Maßnahme geeignet ist ein bestimmtes Ziel zu erreichen. Aussagen über das *Wie* ein Ziel erreicht wird, spielen hier keine Rolle. [21]

Steve Krug beschreibt Usability in seinem Buch [9] wie folgt:

> „Schließlich meint Usability einfach nur, dass man darauf achten soll, dass etwas richtig funktioniert: Eine Person mit durchschnittlichen (oder auch unterdurchschnittlichen) Fähigkeiten und Erfahrungen soll das Ding – sei es eine Website, ein Kampfjet oder eine Drehtür – in der beabsichtigten Weise benutzen können, ohne hoffnungslos frustriert zu werden.“ [9]

Auch wenn diese Überlegung bezüglich der Usability eher allgemein gehalten wurden, gilt dies selbstverständlich auch für Apps auf mobilen Endgeräten.

## 4.1 Usability im Kontext von Software

Oben genannte Definitionen treffen natürlich auch auf Software – sei es eine App oder Website – zu. Dennoch kann man durch die speziellen Eigenschaft von Software weitere Kriterien bezüglich Usability nennen, wie sie Krug in seinem Buch [9] herausstellt.

- Anwendungen sind keine Bücher. Die Benutzer werden selten eine Webseite oder Anwendung komplett durchlesen, sondern sind vielmehr darauf fokussiert den Inhalt zu scannen. Dem ist so, da sie auf der Suche nach einer bestimmten Sachen sind oder weil sie wenig Zeit haben. Damit der Benutzer sich schnell zurechtfindet, ist der Einsatz von großen, prägnanten Überschriften sinnvoll. Diese können beim Überfliegen schnell auf den gesuchten Inhalt hinweisen. Für längere Sachverhalte eigenen sich Aufzählungslisten. Schlüsselwörter sollten wenn möglich hervorgehoben werden.
- Genauso sollten Abschnitten einer Anwendung die klar zusammengehören auch gleich ausschauen. Dazu gehört ein einheitlicher Style oder auch eine gemeinsame Überschrift der Abschnitte.
- Konventionen und akzeptierte Designmuster sollten möglichst eingehalten werden, da die Benutzer mit ihnen vertraut sind. Neue Ansätze sollten nur dann eingesetzt werden wenn sie tatsächlichen Mehrwert bringen.

- Instruktionen sollten nicht vorhanden sein, da die Anwendung selbsterklärend an sich sein sollte. Sind sie dennoch notwendig, so sollten sie möglichst knapp gehalten werden.
- Die meisten Menschen interessieren sich nicht dafür wie eine Anwendung funktioniert, sondern sie sind daran interessiert, *dass* sie funktioniert. Es kann auch vorkommen, dass beispielsweise mit einer App letztendlich anders interagiert wird als dies zuvor von dem Programmierer bzw. Designer gedacht war.
- Ein Großteil der Aktivität die Menschen in Anwendungen erledigen, ist nach dem nächsten anklickbaren Objekt zu suchen. Anklickbare Schaltflächen sollten daher klar gekennzeichnet sein.
- Entwickler sind selber Anwendungsbenutzer und haben daher ihre eigenen Vorlieben. Diese sollten sie jedoch in Bezug auf die Usability der Anwendung die sie bauen vernachlässigen. Es geht vielmehr darum, was am geeignetsten für die Anwendung ist und nicht was dem Entwickler bzw. Designer am liebsten ist.
- Schließlich sollte der Anwender einige Fragen im Bezug auf die genutzte Software leicht beantworten können. Wo befindet sich der Anwender gerade? Wo wird die Anwendung begonnen? Wo werden bestimmte Merkmale gefunden? Was sind die wichtigsten Elemente der Seite? Wieso wird etwas so genannt, wie es in der Anwendung steht?

## 4.2 Zusammenhang zwischen Usability und User Experience

Häufig wird User Experience (UX) und Usability gemeinsam verwendet, ohne dass die Abgrenzung zwischen den beiden Begriffen deutlich ist. Was unter Usability verstanden wird ist oben definiert. User Experience wird im Deutschen mit Nutzererlebnis übersetztet und geht mehr auf das subjektive Empfinden der Benutzer ein, wohingegen Usability auf den reinen Nutzen bzw. die Effizienz während der Benutzung des Produkts eingeht. Das heißt auch, dass bei UX die Erwartungshaltung vor

und nach der Benutzung zum Beispiel einer Software miteinfließen lässt. Laut der DIN ISO 9241-210 wird User Experience folgendermaßen definiert.

**Definition 4.2.1.** „[*User Experience* sind die] Wahrnehmungen und Reaktionen einer Person, die aus der tatsächlichen und/oder der erwarteten Benutzung eines Produkts, eines Systems oder einer Dienstleistung resultieren. [...] Dies umfasst alle Emotionen, Vorstellungen, Vorlieben, Wahrnehmungen, physiologischen und psychologischen Reaktionen, Verhaltensweisen und Leistungen, die sich vor, während und nach der Nutzung [des Produkts] ergeben.“ [31]

Abbildung 4.1 verdeutlicht den Zusammenhang von Usability und User Experience noch einmal in Form eines vorwärtsgerichteten Diagramms deutlich.

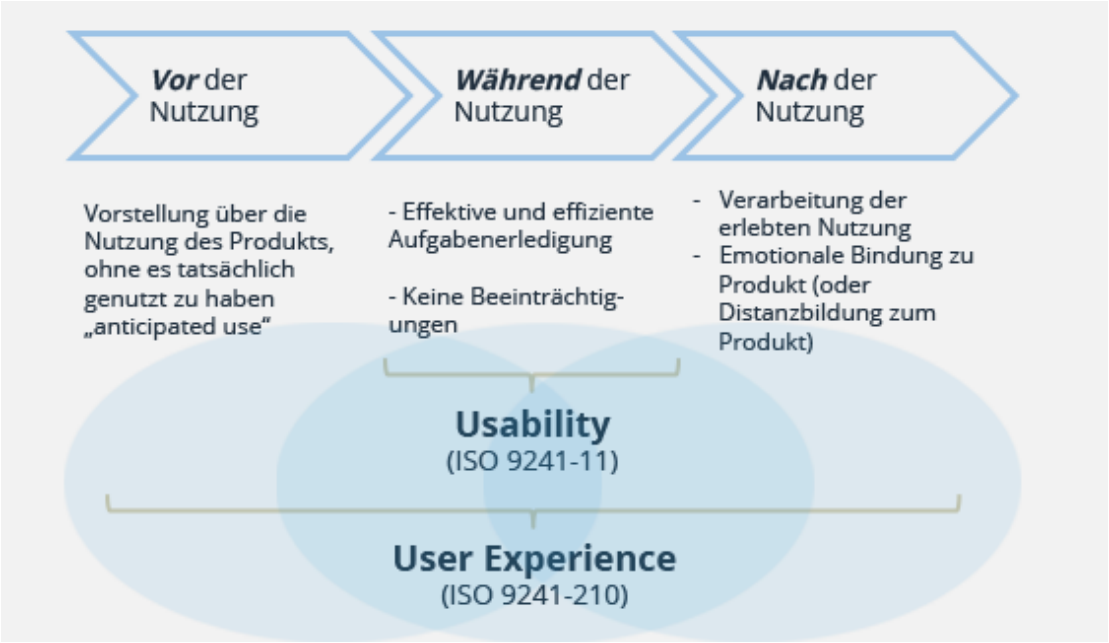



Abbildung 4.1: Usability im Kontext von UX. Übernommen aus [31].

Aus dem Diagramm wird klar, dass es ohne Usability keine UX gibt. Vorstellungen über ein Anwendung kommen bei dem Anwender unweigerlich auf. Der Entwickler hat bei der Anwendung dann die Aufgabe, diese in Code umzusetzen. Schafft er das, sodass die App möglichst effektiv und effizient in ihrer Anwendung ist, erfährt der Nutzer ein erfolgreiches Erlebnis und baut eine positive, emotionale Bindung zu der Anwendung auf. Andernfalls erfolgt eine Distanzbildung zum Produkt.

## 4.3 Relevanz von User Experience

Steve Krug stellt in [9] auf Seite 18 die Frage wieso es wichtig sei die Usability-Erwartungen einer Anwendung zu treffen, beziehungsweise zu übertreffen. Die Annahme, dass die Konkurrenz im Internet immer nur ein Klick entfernt sei und die *User* woanders hingingen wenn sie frustriert mit der Anwendung seien, stimmt laut Krug nur zu einem gewissen Grad. Er schreibt, dass es erstaunlich sei, wie viel Durchhaltevermögen manche Menschen bezüglich schlecht programmierter Websites haben und dass sie dazu tendieren sich selber die Schuld am Misserfolg zu geben. Hinzukämen noch die folgenden Aspekte:

- Die Nutzer kennen keine weiteren Alternativen.
- Es stellte schon eine Hürde dar, die Anwendung überhaupt zu finden.
- Beim Wechsel der App/ Site müsste der *User* von vorne anfangen.
- Das Phänomen, dass man jetzt schon Zeit investiert habe und jetzt nicht so schnell aufgeben möchte.
- Unwissenheit darüber ob die Konkurrenz nicht genauso frustrierend sei.

Krug ist in erster Linie der Meinung, dass man die Nutzung einer Anwendung ergonomisch gestalten solle. Er begründet, dass die Technologie im Dienste des Menschen stünde und nicht andersherum. Aus diesem Grund sollten Anwender nicht gezwungen werden über die Benutzung einer Site nachzudenken. Sie sollen die Informationen die sie suchen *mühelos* bekommen, ohne dass ihnen Energie, Enthusiasmus und Zeit geraubt wird.
Jakob Nielsen weicht in dieser Frage von Krug ab denn er schreibt in [12]:

> „If a website is difficult to use, people leave. If the homepage fails to clearly state what a company offers and what users can do on the site, people leave. If users get lost on a website, they leave. If a website's information is hard to read or doesn't answer users' key questions, they leave. Note a pattern here?“ [12]

Nielsen deutet damit darauf hin, dass User Experience eine notwendige Bedingung für das Überleben einer Anwendung ist.

## 4.4 Faktorenpyramide für ein positives User Experience

Um ein durchweg positives Nutzererlebnis für den Benutzer zu erzeugen, ist die Einhaltung gewisser Faktoren notwendig. Die Pyramide in Abbildung 4.2 zeigt die notwendigen Schichten, die eingehalten werden müssen. Es gilt, dass die oberen Schichten auf der Unteren aufbauen.

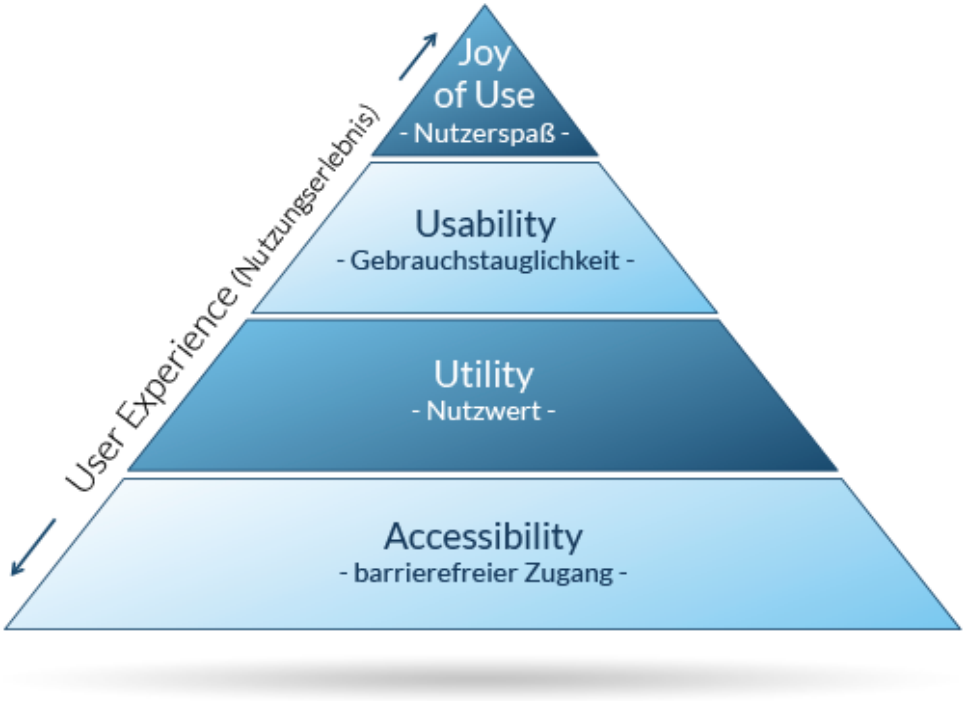



Abbildung 4.2: Faktorenpyramide eines positiven Nutzererlebnis. Übernommen aus [35].

### 4.4.1 Faktor 1: Accessibility

Um eine Anwendung überhaupt nutzen zu können, muss sie zugänglich sein. Diese Voraussetzung muss zunächst geschaffen sein. Laut der Web Accessibility Initiative (WAI)[1] heißt das insbesondere, dass

[1] https://www.w3.org/WAI/fundamentals/accessibility-intro/#important

> Menschen mit Einschränkungen das Web benutzen können. Exakter ausgedrückt bedeutet Web Accessibility, dass Menschen mit Einschränkungen das Web wahrnehmen, verstehen, navigieren und mit ihm interagieren können. [14]

Laut einer Studie der World Health Organisation (WHO) leiden etwa 15 % der Weltbevölkerung unter einer Einschränkung.[2] Besonders ältere Menschen und Personen mit körperlichen Einschränkungen, wie Farbenblindheit, oder Taubheit sind auf Accessibility angewiesen. In dem Artikel „The evaluation of accessibility, usability and user experience“ weisen Petrie und Bevan darauf hin, dass es kostengünstiger sei von Anfang an auf den barrierefreien Zugang zu achten, als später die fehlenden Features im Entwicklungsprozess nachzuholen. [14]

### 4.4.2 Faktor 2: Utility

Utility sagt aus, ob ein *Nutzwert* in der Anwendung vorhanden ist. Ganz konkret bedeutet das ob die Anwendung das liefert, was der Nutzer anfordert. Hier zählt weniger wie einfach die Anwendung zu bedienen ist. Es wird nur darauf geachtet, dass die reine Funktionalität gegeben ist. Utility ist eine noch nicht benutzerorientierte Schicht. [12]

### 4.4.3 Faktor 3: Usability

Was Usability bedeutet und wie es in den Kontext des User Experience einzuordnen ist, wurde in Abschnitt 4.1 und 4.2 erläutert. In Bezug auf die Faktorenpyramide bedeutet das, dass Usability ohne Utility und Accessiblity nicht existiert. Aus dem Kontext der Pyramide geht hervor, dass Usability beschreibt, wie einfach und angenehm die Benutzung einer Anwendung ist. Voraussetzung ist die Zugänglichkeit und Funktionalität der Anwendungsmerkmale.

[2] https://www.interaction-design.org/literature/topics/accessibility

### 4.4.4 Faktor 4: Joy of Use

Die Spitze der Faktorenpyramide bildet die Schicht *Joy of Use*. Damit ist nicht mehr nur die Gebrauchstauglichkeit der Anwendung gemeint, sondern vielmehr die *freiwillige* Benutzung, weil eine emotionale und engere Verbindung aufgebaut wird. Es entsteht Freude bei der Benutzung. Damit dies geschieht, muss die Anwendung selbstverständlich zugänglich sein, einen Nutzwert haben und tauglich sein, weshalb die tieferliegenden Schichten ausschlaggebend sind. In ihrem Artikel „Egineering Joy“ schreiben Hassenzahl, Beu und Burmester, dass ein fundamentaler Unterschied darin besteht, ob eine Arbeit effizient oder genüsslich verrichtet wird. Joy of Use sei die logische Konsequenz von Usability, da Freude aus humanistischer Sichtweise essentiell für das Leben sei. Software solle aus diesem Grund stärker unter dem Aspekt der Nutzererfahrung anstelle der Gebrauchstauglichkeit entwickelt werden. [6]

## 4.5 Usability in Flutter/ Crossplattformprogrammen

Flutter verspricht auf seiner Homepage einen Fokus auf User-Experience zu legen. Der Vorteil von plattformübergreifenden Frameworks ist, dass man sowohl für iOS, als auch für Android Anwendungen entwickeln kann. Beide Plattformen haben unterschiedliche Richtlinien bezüglich des User Experience Design (UX Design). Vaishali Sonik schreibt in [23], dass Anwender einen großen Unterschied zwischen Apples iOS und Googles Android bemerken. Der Grundstein für die unterschiedlichen Designs wird in den Human Interface Guidelines[3] von Apple und mit den Material Design Guidelines[4] von Google gelegt.
Ein genauerer Blick in die Dokumentation offenbart selbst bei unscheinbaren UI-Komponenten große Unterschiede. In Abbildung 4.3 sieht man die beiden Kalender-Date-Picker der jeweiligen Plattformen. Das Einstellen des richtigen Datums wird bei iOS über ein Rädchen simuliert, dass mit dem Daumen hoch- und runtergeschoben wird. Bei Android (Oreo) hingegen wählt man das Datum aus der Übersicht des gesamten Monats. Um eine gleichbleibende User Experience auf bei-

[3] https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/
[4] https://www.material.io/design/

den Betriebssystemen zu gewährleisten, legt Flutter über den Widget-Katalog[5] die unterschiedlichen UI-Komponenten fest. Google definiert das Material Design. Die Widgets für die Material-Komponenten sind in dem Widget-Katalog daher sehr ausgeprägt. Selbstverständlich hat ein Programmierer die Möglichkeit, eine Material-App in den AppStore hochzuladen. Apple-Nutzer sind jedoch die Komponenten im iOS-typischen Design gewöhnt.

Ein *Cupertino-Widget* ist ein Widget, dass typische iOS-Design-Komponenten für die aktuelle iOS-Version in Flutter darstellt. Damit zum Beispiel der Date-Picker mit Flutter genauso wie auf dem iPhone in Abbildung 4.3 aussieht, stellt das Framework in dem Cupertino-Widget-Katalog den Cuptertino-Date-Picker zur Verfügung. Viele weiter UI-Komponenten wie Buttons, Dialog- oder Textfelder sind ebenfalls in einer Cupertino-Variante vertreten. So soll trotz der Tatsache, dass Google sowohl Flutter als auch das Material Design herausgibt sichergestellt werden, dass iPhone Nutzer eine gleichbleibende User Experience auf ihrer Plattform erfahren.

[5] https://flutter.dev/docs/development/ui/widgets

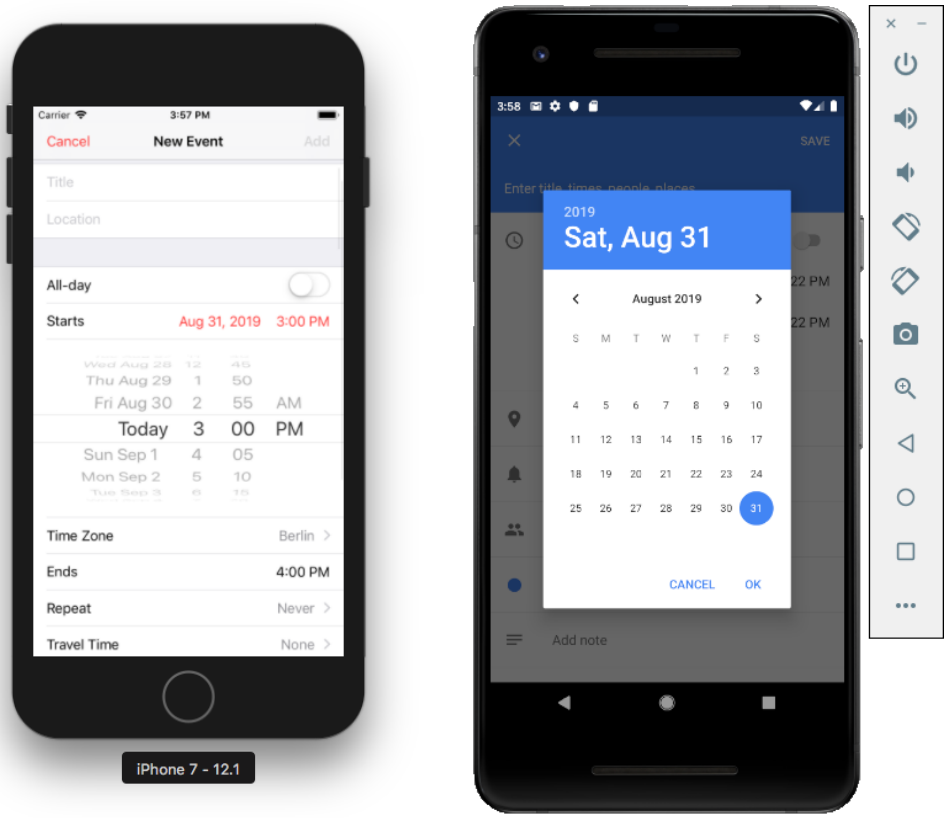



Abbildung 4.3: Vergleich iOS-Picker und Android-Picker

# 5 Entwurf

Im Kapitel 2 wurde das TrackYourTinnitus Projekt der Universität Ulm vorgestellt. Es wurde darauf hingewiesen, dass bei der aktuellen Version des Projekts erhebliche Verbesserungsmöglichkeiten in Bezug auf die Datenerfassung bei mobilen Endgeräten besteht. In den Kapiteln 3 und 4 wurden die theoretischen Kenntnisse vorgestellt, auf die im vorliegenden Entwurfskapitel und letztendlich bei der Implementierung eingegangen werden.
Der Entwurf eines jedweden Produkts, auch der eines Programms, ist ein kontinuierlicher Prozess, in dem Ideen schrittweise umgesetzt und verbessert werden. Dieser Entwurfsfluss wird auch in diesem Entwurfskapitel abgebildet. Aus diesem Grund werden zunächst die initialen Ideen zur Entwicklung progressiver Eingabekonzepte zur Datenerfassung vorgestellt, um dann daraus die Entwürfe erstellen zu können. Am Anfang sind hierbei Mockups entstanden, die sich dann über eine Zwischenversion zur Endversion des Konzeptes etwickeln.

## 5.1 Allgemeine Konzeptionierung

Bei der Benutzung der aktuellen TrackYourTinnitus App fällt auf, dass die Aufforderung die Fragebögen auszufüllen, unabhängig von Ort und Zeit passiert. Das bedeutet, dass der Benutzer in der Lage sein muss, den Fragebogen schnell und effizient zu beantworten. Dieser Aspekt soll sich insbesondere bei den Gesten der Benutzung der App und den angezeigten Informationen auf dem Bildschirm zur jeweiligen Frage wiederfinden. Der Informationsfluss soll klar vorwärtsgerichtet sein, sodass der User das Gefühl bekommt zügig voranzukommen. Der Fragebogen der aktuellen Version der TrackYourTinnitus App ist ein großer scrollbarer Fragebogen der unübersichtlich ist. Damit die einzelnen Fragen mit mit einem Blick erfassbar

sind, sollen sie jeweils auf eine separate Seite gespeichert werden. Auf dieser Seite kann dann die Aktion zur Dateneingabe stattfindet.

Das Leiden an einem Tinnitus ist eine unangenehme Lebenssituation. Ein besseres Verständnis über die Krankheit soll in Zukunft mithilfe der Erhebung der Daten möglich sein. Im Moment ist es aber wahrscheinlich, dass der Patient zunächst ein persönliches Missvergnügen mit der TrackYourTinnitus App verbindet, weil sie ihm an seine täglichen Leiden erinnert. Aus dem Kapitel 4 geht hervor, dass es von wesentlicher Bedeutung ist, eine ausreichende Benutzerfreundlichkeit bei der Benutzung der App zu erreichen. Zusätzlich ist ein positives Maß an Benutzererlebnis erforderlich, um den User an die Anwendung zu binden. Die Schwierigkeit bei diesem Aspekt ist die Subjektivität die jeder Anwender bezüglich eines Erlebnisses fühlt. Dennoch soll aus diesem Grund versucht werden, dem Benutzer das Gefühl zu geben, nicht einfach nur einen bedeutungslosen Fragebogen auszufüllen, sondern stattdessen kleine Aktionen auszuführen. Beispiele hierfür könnten das Spielen eines Musikinstruments, Sporttreiben oder das Lösen kleiner Rätsel sein.

Die Grundlage eines Eingabekonzeptes zur Datenerfassung bildet die allgemeine Struktur und Navigation der Anwendung. Die aktuelle Version der TrackYourTinnitus App ist im Hinblick auf die Navigation veraltet und wenig intuitiv. Das Drop-Down-Menü am oberen Rand versperrt den Blick auf die Informationen dahinter, sodass ein rascher Überblick über die App nicht möglich ist. Weiterhin ist nicht klar wo ein klar definierter Ort wie Home oder Start ist. Die verschiedenen Arten der Einstellungen die in dem Drop-Down-Menü erscheinen, können an einem zentralen Ort gesammelt werden, weil es letztendlich alles Einstellungen sind. Die Doppelfunktion des Drop-Down-Menüs als Überschrift und als Navigationsmöglichkeit innerhalb der App, kann heutzutage aufgrund der größer gewordenen Displays aufgebrochen werden. Bei immer größer werdenden Smartphone-Bildschirmen muss nicht zwanghaft Platz eingespart werden. Stattdessen sollte bei der Konzeptionierung darauf geachtet werden, dass man weiterhin alles gut mit dem *Daumen* erreichen kann, weil er ein wesentlicher Bestandteil der Dateneingabe im Einhandmodus darstellt. Für Besitzer von Mobiltelefonen der Größe eines iPhones Xs Max ist es nur erschwert möglich die Hauptnavigation der TrackYourTinnitus App zu erreichen, oder die oberen Fragen der Studie zu beantworten *ohne* das Handy umzufassen.[1] Das Konzept soll daher für die Navigation auf App-Level eine Tab-Bar am

[1] Eigenerfahrungen des Autors dieser Bachelorarbeit.

unteren Rand des Bildschirms erhalten, damit die Hauptnavigation wieder problemlos mit dem Daumen gesteuert werden kann. Viele moderne Apps weisen dieses Navigationsmuster auf. Unter anderem wird diese Art der Navigation ausdrücklich in den Human Interface Guidelines von Apple empfohlen.[2]


Abbildung 5.1: Struktur der Navigation

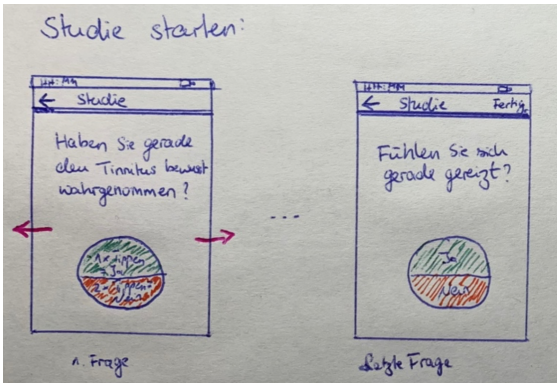



Abbildung 5.2: Extrahierte Fragen

Die Realisierung eines Konzeptes ist ein kreativer und insbesondere ein iterativer Prozess und fängt klein an. Daher war der allererste Entwurf noch auf Papier. Vor allem wird in Abbildung 5.1 die beschriebene Grundnavigation dargestellt, wohingegen die eigentlichen Fragen die über ein Button „Studie starten“ zu erreichen sein sollen, noch recht vage in Abbildung 5.2 dargestellt sind.

[2]https://developer.apple.com/design/human-interface-guidelines/ios/bars/tab-bars/

### 5.1.1 Auswahl der Gesten

Mobile Endgeräte werden in der Regel mit dem Daumen bedient. Textfelder und Formulare können auch mit dem Daumen bedient werden, dennoch sind sie im Hinblick auf Schnelligkeit bei der Dateneingabe auf mobilen Endgeräten zu vernachlässigen. Daher soll das Augenmerk bei der Datenerfassung auf Gesten mit dem Daumen liegen.

In [30] wird erklärt, wie Flutter Gesten umsetzt. Die untere Schicht wird durch Zeiger bzw. Pointers repräsentiert und generiert Rohdaten über die erste Berührung des Benutzers mit dem Bildschirm, bis dieser seinen Finger wieder hebt. Auch wenn man die Pointer-Events auf Widget-Ebene anwenden kann, ist es empfehlenswert direkt die Gesten aus dem Framework zu benutzen.
Die wesentlichen Gesten, ohne die jeweiligen Subgesten, die Flutter zur Verfügung stellt, sind:

- Tap
- Double tap
- Long press
- Horizontales Ziehen
- Vertikales Ziehen
- Pan: Ein Mischung aus horizontalem und vertikalem Ziehen

Wenn die Fragen aus dem Fragebogen wie oben beschrieben in eigene Screens unterteilt werden sollen, dann ist es sinnvoll, eine Geste für die Navigation innerhalb des Fragebogens zu reservieren. Jede Frage ähnelt einer Karte die weitergeschoben werden soll, daher kommt hier eine typische Swipe-Bewegung in Frage. Durch horizontales Wischen auf dem Bildschirm soll die nächste Frage erscheinen. Diese Geste ist einfach, intuitiv und auch leicht im Einhandmodus ausführbar.

Um die jeweilige Frage zu beantworten soll eine andere Geste reserviert werden. Ein Zusammenziehen der Finger oder auch Pinchen genannt ist hier eher nachteilig, da dies in Bezug auf die Anwendung eher umständlich ist. Der Fluss des Durchblätterns der einzelnen Fragen wird damit gestört. Tappen wäre eine Möglichkeit die auch im Einhandmodus verfügbar ist, dennoch wird dadurch die Interaktion des

Benutzers mit der Anwendung eingeschränkt. Wie zuvor beschrieben soll er durch Aktionen wie das Spielen eines Musikinstruments in die App verwickelt werden, wodurch eine längerfristige Bewegung auf dem Bildschirm geeigneter ist. Zudem ist es ersichtlich, dass es sich nach wie vor um das Einstellen oder Regeln eines Wertes auf einer Skala handelt. Somit stellt eine Wischgeste nach oben idealerweise die Werte hoch und andersherum, für den negativen Fall, ein Wischen nach unten die Wert runter. Zur Veranschaulichung wird im Laufe des Fragebogens beispielsweise die Frage gestellt: „Wie ist Ihre aktuelle Stimmungslage?“ Wenn der User hoch wischt, also auch von der physischen Bewegung einen „Daumen hoch“ gibt, dann könnte das ebenfalls sinnbildlich auf dem Bildschirm passieren, um zu signalisieren, dass seine Stimmungslage hervorragend ist. Wenn er einen „Daumen runter“ gibt, ist dies ein Zeichen dafür, dass seine aktuelle Stimmungslage nicht so gut ist es und die gleiche Geste soll auf dem Bildschirm dargestellt werden. Aufgrund der Intuition und Versinnbildlichung dieser Geste soll sie primär bei der Dateneingabe verwendet werden.

### 5.1.2 Synchronisation der Eingabe mit der Skala

Um Daten eingeben zu können muss der Nutzer wissen, welche Quantität seine Eingabe mit sich bringt. Daten sind nur im Kontext einer Skala semantisch zu verarbeiten. Dabei gibt es einfache Messgrößen wie zum Beispiel die Temperatur. Hier gestaltet sich die Erfassung der Daten als wenig problematisch, weil eine implizite Skala vorgegeben wird. Der gemessene Wert könnte zum Beispiel ins Verhältnis zum kältesten und heißesten Wert auf der Erde gesetzt werden und schon weiß man in welchem Bereich sich der gemessene Wert bewegt.
Bei *emotionalen* Fragen, die das Wohlergehen betreffen wie es im TrackYourTinnitus Projekt der Fall ist, gestaltet sich die Skalierung schwieriger. Wenn zwei Menschen auf die Frage „Wie ist Ihre aktuelle Stimmungslage“ mit „Gut“ antworten, bedeutet das nicht zwangsläufig genau dasselbe. Gut ist hierbei keine vergleichbare Maßeinheit. Letztendlich ist diese Frage nur schwer oder überhaupt nicht zu beantworten. Dennoch sollen bei diesem Eingabekonzept die Skalen so gewählt werden, dass sie eine *selbsterklärende* Funktion erfüllen bei dem der Patient seinen emotionalen Zustand auf dem Bildschirm wiederfindet. Darüberhinaus sollen die Skalen einprägsam sein. Die Patienten werden den Fragebogen viele Male ausfüllen

müssen und wenn sie jedes mal aufs Neue wieder über die Tatsache nachdenken müssen, ob der eingegebene Wert zu ihrer Tinnituswahrnehmung passt, ist das langfristig belastend. Zu den jeweiligen Fragen soll daher ausdrücklich ein Objekt gewählt werden, das allein mit seinem intrinsischen Charakter schon mit dem Wahrnehmungstyp in Verbindung gebracht wird. Beispielsweise soll auf die Frage „Wie laut ist der Tinnitus momentan?“ ein Musikinstrument verwendet werden, dass gut zu hören ist.

### 5.1.3 Unbefangenheit der Skala

Bei dem TrackYourTinnitus Projekt handelt es sich um emotional-subjektive Daten die erfasst werden. Aus diesem Grund sollen die Befragten so wenig wie möglich durch voreingestellte Werte der Skala beeinflusst werden. Wenn die Skala beim Durchwischen der Fragen einen voreingestellten Wert zeigt, kann sich der Nutzer von diesem leiten lassen Er wählt dann einen Wert in der Nähe, oder lässt die Vorgabe unverändert. Dieses Verhalten soll bei der Datenerfassung vermieden werden. Darüber hinaus ist es Ziel, dass der Nutzer aktiv an der App teilnimmt, was die Eingabe eines persönlichen Wertes voraussetzt. Für den Entwurf bedeutet das, dass die Skala am Anfang ausgegraut ist und der Zeiger keinen Wert anzeigt.

## 5.2 Konzeptionierung des Fragebogens

Wie in Abschnitt 2 vorgestellt, wird der Benutzer während des Ausfüllens des Fragebogen durch acht Fragen geführt. Die erste Frage zielt auf die allgemeine Wahrnehmung ab. Die darauffolgenden Fragen versuchen den Tinnitus mit einer bestimmten emotionalen Eigenschaft in Verbindung zu bringen. Ganz explizit sind die Fragen:

- Haben Sie gerade den Tinnitus bewusst wahrgenommen?
- Wie laut ist der Tinnitus momentan?
- Wie belastend empfinden Sie den Tinnitus im Moment?
- Wie ist Ihre aktuelle Stimmungslage?
- Wie aufgeregt fühlen Sie sich gerade?

- Wie sehr haben Sie sich auf das konzentriert, was Sie gerade tun?
- Fühlen Sie sich gerade gereizt?

Um ein passendes Eingabekonzept zu entwerfen, müssen die Fragen zunächst einzeln analysiert werden. Dabei fällt auf, dass die erste und die letzte Frage den gleichen Fragetyp teilen. Es sind Entscheidungsfragen mit jeweils „Ja“ oder „Nein“ als Antwort. Sie gehören somit zu den geschlossenen Fragen.

Die zweite bis siebte Frage zielen auf eine quantitative Antwort des Befragten ab. Hier ist demnach eine kompliziertere Skala notwendig.

### 5.2.1 Datenerfassung zu Entscheidungsfragen

Für die erste und letzte Frage kommen als Datentyp nur zwei Werte in Frage. Es soll für beide Antworten ein starkes, visuelles Symbol ausgewählt werden. Wie bereits erwähnt, soll die Skala zunächst keinen Wert anzeigen, sodass der Nutzer nicht voreingenommen antwortet.

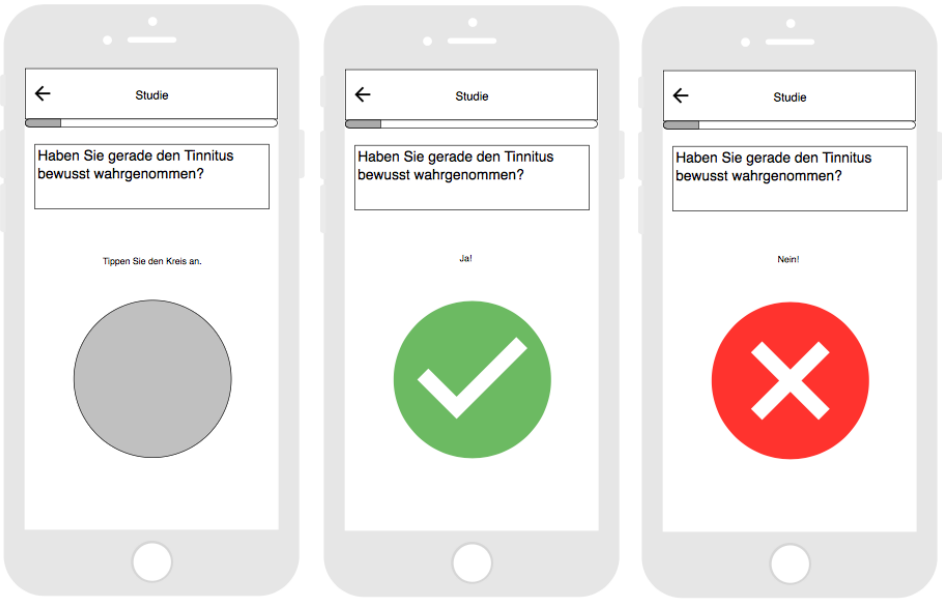



Abbildung 5.3: Haben Sie den Tinnitus bewusst wahrgenommen? – Ausgegraut, positiv, negativ.

Die Eingabe soll jeweils mit einem Swipe nach oben für „Ja“ und mit einem Swipe nach unten für „Nein“ beantwortet werden. Wischt der User nach oben erscheint

ein grüner Kreis mit einem Häkchen. Ein Häkchen, oder im Englischen auch Check steht im Allgemeinen für Zustimmung. Wischt der Patient nach unten soll ein roter Kreis erscheinen mit einem X. Das X steht allgemein für Abbruch bzw. negative Zustimmung. Die Farben und die Form sollen dabei an ein Ampelsystem erinnern. In vielen Ländern dieser Erde stehen diese Farben folglich für „*Ja*, ich darf fahren“ bzw. „*Nein*, ich darf nicht fahren“. Sollte es dennoch zu Unstimmigkeiten mit den Farben bezüglich der Zustimmungsrichtung kommen, soll die Möglichkeit der Abänderung bei der Implementierung beachtet werden. Die Fortschrittsanzeige zeigt hier noch an, dass der User ganz am Anfang des Fragebogens steht.

Für die letzte Frage, soll die gleiche Struktur verwendet werden. Im Unterschied zur initialen Frage gilt hier aber, dass der User die Möglichkeit bekommt, seine Eingabe abzuschicken- Daher soll auf dieser Seite ein Button erscheinen, der die Ergebnisse speichert. Die Fortschrittsanzeige signalisiert hier, dass keine weiteren Fragen mehr kommen.

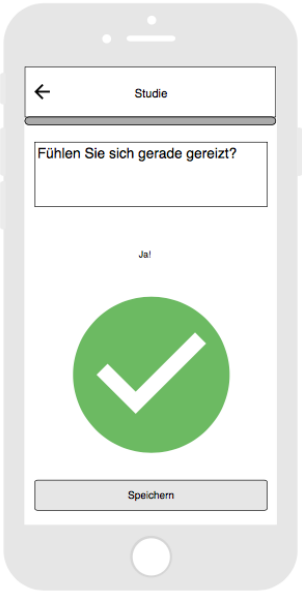



Abbildung 5.4: Fühlen Sie sich gerade gereizt? – Positiv.

### 5.2.2 Datenerfassung zu quantitativen Fragen

Die quantitativen Fragen stellen eine höhere Anforderung an die Skalen. Gleichermaßen gestaltet sich das Mapping der Eingabe auf die Skala schwieriger. Zu keinem Zeitpunkt der Eingabe darf unklar sein, welche Quantität der User gerade eingibt. Für die zweite Frage der Lautstärke, soll als einprägsames Symbol eine Trompete verwendet werden. Mit der genannten Geste aus Abschnitt 5.1.1 soll sie „gespielt werden“. Damit der Benutzer nicht von voreingestellten Werten fehlgeleitet wird, soll sie am Anfang keine Note zeigen. Mit kontinuierlichem Hochwischen erscheinen mehr und mehr Noten auf dem Bildschirm, bis letztendlich die Maximallautstärke in Form von einer maximalen Anzahl an Noten angezeigt wird.

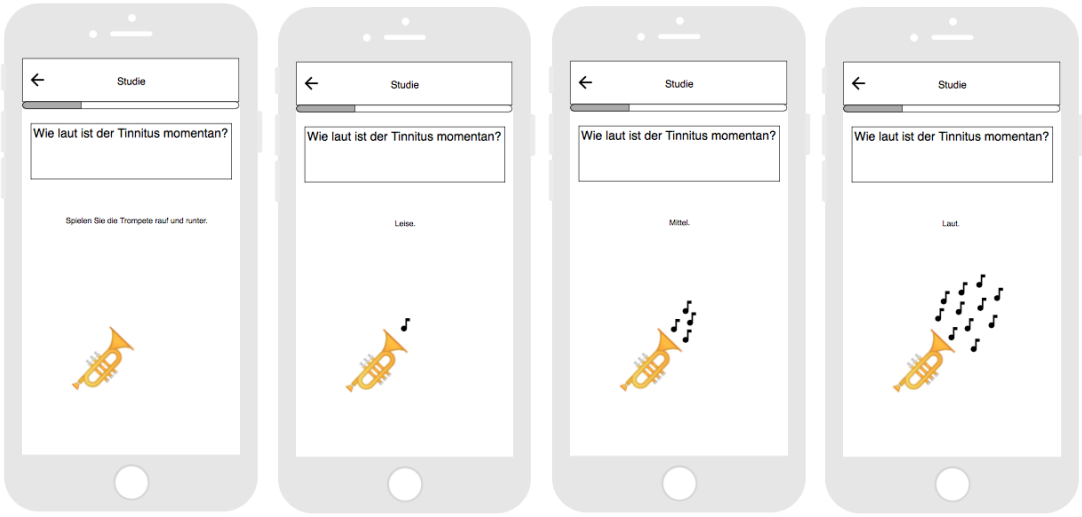



Abbildung 5.5: Wie laut ist der Tinnitus momentan? – Nicht hörbar, leise, mittel, laut.

Die dritte Frage verlangt von dem Eingebenden die Festlegung der momentanen Belastung durch den Tinnitus. Während bei der Lautstärke Musik affine Menschen angesprochen werden, soll hier versucht werden sportlich interessierte Personen zur Eingabe zu motivieren. Belastung ist im sportlichen Bereich stark mit Gewicht konnotiert. Eine Hantel ist somit ein ideales Symbol, das für die Dateneingabe verwendet werden kann. Je größer sie ist, desto schwerer ist sie und umso belastender ist der Tinnitus für die betroffene Person. Die Hantel muss mit einer Hebegeste vergrößerbar und in umgekehrte Richtung verkleinerbar sein. So soll sichergestellt werden, dass die Eingabe und der Bildschirmoutput übereinstimmen. Die Unbefangenheit der Skala soll initial durch ein X erreicht werden, das lediglich den Ort des

Sportgeräts darstellt.

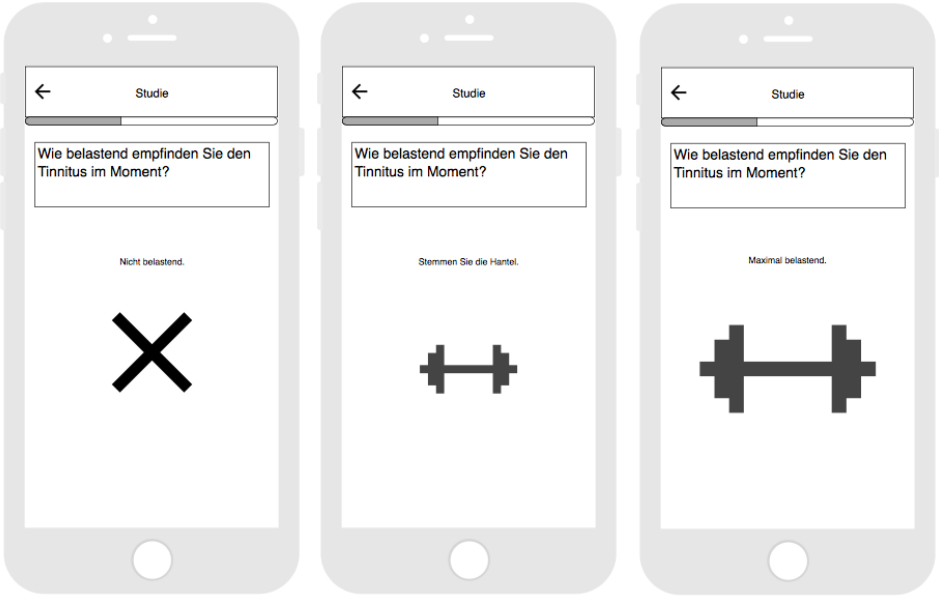



Abbildung 5.6: Wie belastend empfinden Sie den Tinnitus im Moment? – Nicht belastend, Ausgangslage, maximal belastend.

Die vierte Eigenschaft die in dem TrackYourTinnitus Fragebogen abgefragt wird, ist die aktuelle Stimmungslage des Patienten. Hier gab es zwei Überlegungen zur Umsetzung. Die eine Überlegung war die aktuelle Stimmungslage durch ein Emoji darzustellen, der abhängig von der Wischgeste seine Gesichtszüge verändert. Die andere Überlegung, die wie in Abbildung 5.7 zu sehen ist und im Entwurf konzipiert wurde, ist die Möglichkeit die Stimmung mit einem „Daumen hoch“ bzw. „Daumen runter“ und Zuständen dazwischen darzustellen. Die Daumenskala legt Hoch- und Tiefpunkt sehr präzise fest. Wenn der Daumen sich in seinem maximal oberen Anschlag befindet, kann er nicht weiter gedreht werden, ohne den Höchstwert zu verlieren. Gleiches gilt für den Tiefpunkt. Daumenhoch und -tiefpunkt stehen somit in einem verständichen Zusammenhang mit dem Hoch- und Tiefpunkt der Stimmungslage. Mit der Wischgeste des physischen Daumens soll der Daumen auf dem Bildschirm gesteuert werden, sodass er ein Abbild der realen Welt ist. Die Eingabe mit dem Daumensymbol ist sehr einprägsam und daher gut geeignet. Die Unbefangenheit der Skala muss noch entworfen werden. Der Daumen stellt den Zeiger dar, daher darf er auf dem ersten Blick der Frage nicht erscheinen. Zudem soll zur Unterstützung der impliziten Skala noch eine Textausgabe über dem Symbol er-

scheinen, das die Stimmungslage des Daumens noch einmal in Worten ausgibt. So soll sichergestellt werden, dass auch zwischen den Extrema keine Unklarheit bezüglich des aktuellen Wertes herrscht.


Abbildung 5.7: Wie ist Ihre aktuelle Stimmungslage? – Nicht belastend, Ausgangslage, sehr schlecht, sehr gut

Die nächste Frage im Bogen zielt darauf ab zu erfahren, wie aufgeregt sich der Benutzer gerade fühlt. Als einschlägiges Symbol soll hier ein Herz verwendet werden. Je aufgeregter ein Mensch ist, desto schneller pocht sein Herz. Daher ist es hier vorstellbar die Herzfrequenz als in Frage kommende Skala zu verwenden. Dazu soll das Herz zu einem Schlagen animiert werden und und je weiter der User nach oben wischt, desto schneller soll es pochen. Der Patient assoziiert so, dass er aktuell erheblich aufgeregt ist. Im negativen Fall soll das Runterwischen die Herzfrequenz wieder runterregeln. Da die Vibration des Herz allein als Skala schwieriger zu erkennen ist, soll die aktuelle Herzfrequenz als bpm-Wert textuell ausgeben werden. Zudem kann noch ein quantitativer Hinweis im Stil von „Wenig aufgeregt“, „Aufgeregt“, „Sehr aufgeregt“ eingeblendet werden. Das bewirkt, dass selbst Menschen denen die bpm-Einheit fremd erscheint, sich sofort mit dem schnelleren Schlagen des Herz in Verbindung mit den Textlabels zurechtfinden und den richtigen Wert eingegeben können. Wenn der Patient zum ersten Mal auf den Bildschirm gelangt, soll die Animation noch nicht starten. Erst wenn wenn er das Herz berührt, um sein

persönliche Aufregung zu dokumentieren soll es ein optisches Feedback geben.

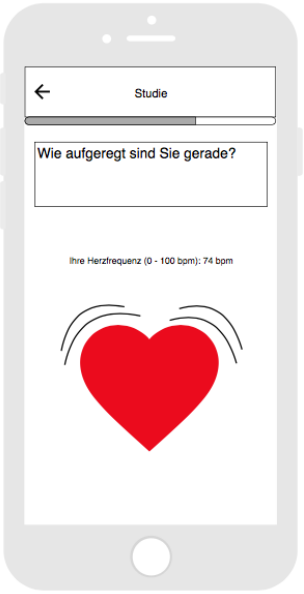



Abbildung 5.8: Wie aufgeregt sind Sie gerade? – Pochendes Herz.

Die sechste Frage im TrackYourTinnitus Projekt misst, inwiefern persönlicher Stress mit dem Tinnitus in Verbindung steht. Stress zu symbolisieren ist schwer, weil es eine individuelle Eigenschaft bei jeder Person ist. Charakteristisch ist jedoch, dass in Stresssituationen zu viel auf eine Person einwirkt und sie das Gefühl hat, nicht mehr Herr der Lage zu sein. Sinnbildlich soll für diesen emotionalen Zustand eine Person auf dem Smartphone-Bildschirm abgebildet werden, auf die Stressblitze „einprasseln“. Um dem User eine visuelle Skala zu ermöglichen, sollen sie sich verfärben. Je weiter der Patient nach oben streicht, desto mehr Blitze werden rot eingefärbt. Wischt der Patient auf dem Bildschirm nach unten, werden die Blitze wieder ausgegraut. Parallel dazu wird neben den Blitzen ein Label eingeblendet, dass den aktuellen Stresswert in Worten ausgeben soll. Um die Unbefangenheit der Dateneingabe zu gewährleisten, soll am Anfang kein Stressblitz farblich gekennzeichnet sein.

Als vorletzte Frage der Datenerfassung wird geprüft, wie stark sich der Anwender auf seine aktuelle Aufgabe konzentriert hat. Um das Maß der Konzentration verständlich zu visualisieren, ist ein Gehirn symbolisch sinnvoll. Es soll mittig auf dem Bildschirm platziert werden. Darüber sollen Matheaufgaben erscheinen, die abhän-

gig von dem Konzentrationslevel an Schwierigkeit gewinnen. Dabei ist darauf zu achten, dass die Aufgaben für den Durchschnittspatienten lösbar sein sollen, damit er aktiv in die Anwendung einbezogen wird. Das aktuelle Schwierigkeitslevel wird dabei farblich markiert und dient so als Skala für das Konzentrationsmaß. Der Aspekt der Unbefangenheit findet auch hier Anwendung, in dem die Skala am Anfang komplett ausgegraut ist und keine Matheaufgaben angezeigt werden.

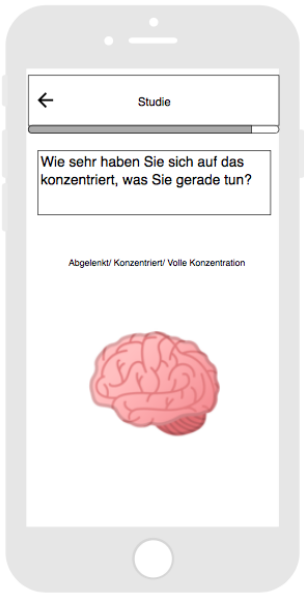



Abbildung 5.9: Wie sehr haben Sie sich auf das konzentriert, was Sie gerade tun? – Gehirnaktivität als Maß für die Konzentration.

## 5.3 Übergang von Mock-Ups zu Flutter

In diesem Abschnitt wird überprüft wie die Konzeptionierung in Flutter ausschaut. Das Konzept sieht eine neue Gliederung der App vor. Dabei ist in diesem Entwurf die Aufteilung zwischen Home, Ergebnisse und Einstellungen gelungen. Die Hauptnavigation befindet sich jetzt am unteren Rand des Smartphones, sodass das Umschalten zwischen den verschiedenen Reitern selbst bei sehr großen Geräten problemlos geschieht.
Als Kernstück der Anwendung ist die Tinnitus-Studie in dem Home-Tab zu finden.

Für den Fall das der Tab in Zukunft mit mehr Inhalt gefüllt wird, soll die Studie immer noch leicht auffindbar sein. Daher wird für den Button „Studie starten“ eine auffällige Farbe gewählt.

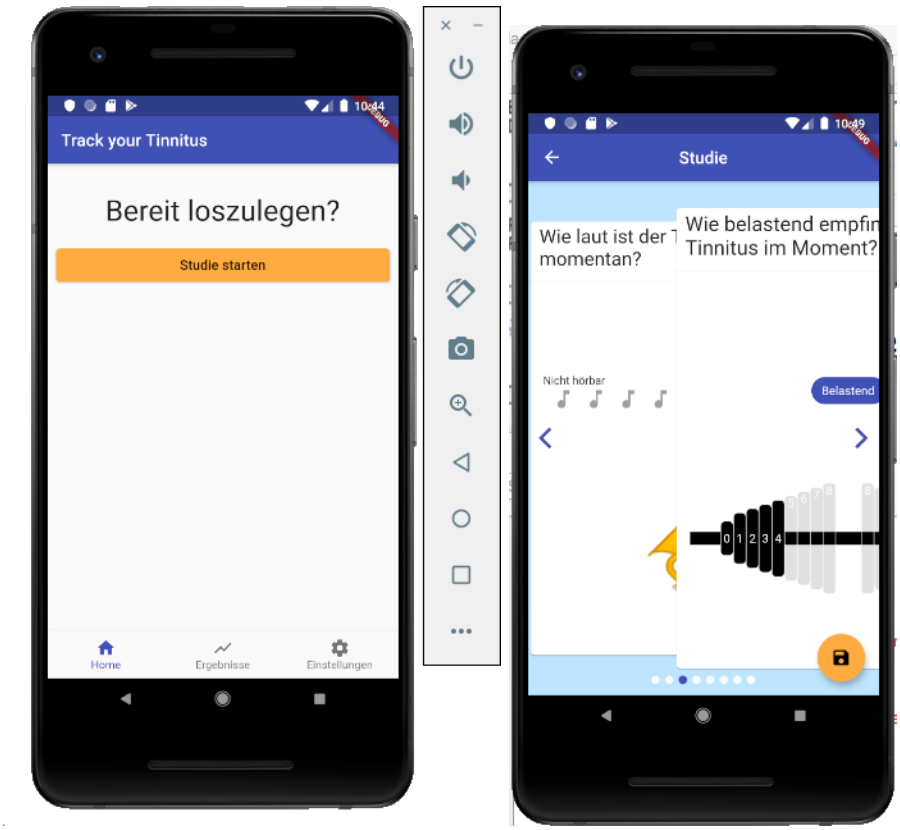



Abbildung 5.10: Übersicht Hauptnavigation und Transition zwischen den Fragen.

Das Bildschirmfoto rechts neben der Gesamtübersicht zeigt den Übergang zwischen den Fragen. Beispielhaft wird hier die Transition zwischen der Frage der Lautstärke und der Frage der Belastung des Tinnitus gezeigt. Die nächste Seite wird dabei stackartig auf die vorherige geschoben. Die Aktion ist durch zwei Kontrollelemente möglich. Zum einen kann der Benutzer der App zwischen den einzelnen Seiten rechts und links vor und zurück wischen. Zum anderen hat er die Möglichkeit mit den Vor- und Zurück-Icons am linken und rechten Rand die Seiten weiterzuklicken. Diese Steuerelemente helfen dem User von der ersten Nutzung an, da sie ihm klar verdeutlichen wie er weiterkommt.

Am rechten unteren Rand befindet sich eine typische Material-Komponente. Der

Floating-Action-Button[3] soll dem User die Möglichkeit geben seine Dateneingabe *am Ende* des Fragebogens abzuspeichern. Das heißt, dass nur auf der Seite der letzten Frage bei Drücken des Buttons eine Aktion geschehen soll.
Falls der Patient die Studie anfängt und abrupt feststellt, dass er keine Zeit hat die Fragen auszufüllen, kann er mit dem Zurückpfeil am oberen linken Rand der App zum Home-Menü navigieren. Auf diese Weise bekommt er das Gefühl, die App zu beherrschen und es wird ihm nicht aufgezwungen, jede einzelne Erfassung abspeichern zu müssen.

Eine Übersicht des Entwurfs des Fragebogens in Flutter ist in Abbildung 5.11 gegeben. Für die erste Frage erscheint das positive Symbol, dass der Tinnitus gehört wurde. Eine Skala ist hier nicht notwendig. Bei der zweiten Frage sind sieben von neun Noten schwarz eingefärbt. Auf diese Weise wird eine spielerische Skala erzeugt, denn je höher der Anwender die Trompete spielt, desto mehr Noten färben sich in der Skala ein.

Für die Belastung durch den Tinnitus gilt ein ähnliches Szenario. Dennoch gibt es hier einen Unterschied im Entwurf. Der Benutzer muss die Hantel für jede weitere Belastungsstufe heben. Es ist somit keine kontinuierliche Wischbewegung nach oben oder unten. Das Hantelobjekt dient gleichzeitig als Skala, da hier die Gewichtsscheiben nach und nach schwarz eingefärbt werden. Beim Loslassen der Hantel findet die Transition von schwarz nach grau bzw. vice versa statt. Zusätzlich zu den visuellen Hinweisen durch die Gewichtsscheiben, werden auf den schwarzen und grauen Scheiben numerische Werte für die Skala verwendet. Dies dient als Hilfestellung für den Patienten, damit er sich schnell bei der Eingabe zurecht findet. Relative Belastungseinschätzungen sind auf diese Weise möglich, da eine Aussage in Bezug auf Minimal- und Maximalwert getroffen werden kann. Schlussendlich umfasst der Entwurf ein Label, dass den numerischen Wert der Eingabe in einen textuellen Wert umwandelt. Die drei verschiedenen Ausgabetypen stellen die aktuelle Dateneingabe deutlich heraus.

Im nächsten Bildschirm zeigt ein verstellbarers Daumensymbol die Stimmungslage des Patienten an. Minimal und Maximalwerte sind durch die jeweiligen Endstellungen des Daumens gegeben. Auf diese Weise kann der Patient auch eine relative Abschätzung vornehmen, wenn er einen Wert dazwischen einstellen möchte. Um

[3] https://api.flutter.dev/flutter/material/FloatingActionButton-class.html

die Eindeutigkeit des Wertes deutlich zu machen, wird die Position des Daumens in Worte gefasst und über ein Label ausgegeben.

Für die Erfassung der Erregung ist eine Herzanimation zu sehen. Die Bewegung des Herz verschnellert sich je weiter der Anwender nach oben wischt und verlangsamt sich je stärker er nach unten wischt. Die Unterschiede in der Schnelligkeit des Herzschlags sind in der Anwendung deutlich zu erkennen. Dennoch ist auf den ersten Blick die Stufe der Dateneingabe nicht sofort ersichtlich. Damit dies behoben wird, umfasst der Entwurf ein Label in dem die aktuelle Herzfrequenz ausgegeben wird. Als Minimal- und Maximalwert wird der Patient auf dem Bildschirm darauf hingewiesen, welches die Endwerte des animierten Herz sind. So kann er schnell abschätzen in welchem Intervall er seine persönliche Aufregung durch hoch- oder runterwischen einstellen möchte.

Die Seite der Stresserfassung zeigt als zentrales Element den User selber. Hier gilt die selbe Wischgeste wie auf der vorherigen Seite. Das Maß an Stress nimmt zu je weiter der Benutzer nach oben streicht. Visuell wird ihm seine Eingabe durch sich rot färbende Stressblitze angezeigt. In der Abbildung 5.11 ist die Eingabe sechs von neun Stressblitzen markiert. Relative Aussagen über den persönlichen Stress sind somit möglich. Auch hier sind die Labels erkennbar, die das jeweilige Stresslevel in Worten zusammenfassen. Abhängig von der Stressbelastung werden der Reihe nach die verschiedenen Labels farblich durchmarkiert. Zusätzlich kumulieren sich darunter die Stressblitze, um die Eigenschaft der Relativität der Eingabe beizubehalten. .

Als vorletzte Eingabe wird von dem Patienten verlangt, dass er angibt, wie sehr er sich auf seine aktuelle Aufgabe konzentriert. Abbildung 5.11 zeigt für diesen Bildschirm eine Denkblase die von einem Gehirn aufsteigt. Die Schwierigkeit der Aufgaben in der Denkblase verändern sich dabei in Abhängigkeit von der Konzentrationsstufe. Das Gehirn bzw. der Anwender benötigt mehr Konzentration für die Aufgaben je weiter er noch oben wischt und vice versa. Das Konzentrationslevel der Dateneingabe wird blau markiert. Die Markierung wandert schrittweise durch die Skala durch. Vom Anwender kann implizit spielerisch eine Einschätzung der Konzentration in Abhängigkeit von der Aufgabe getätigt werden. Verfügt er nicht über die notwendige Zeit die Aufgaben zu analysieren, ist es ihm möglich die Eingabe auch anhand der Skala zu machen. Durch die Konzentrationsstufen sind Minimal- und Maximalwert gegeben, sodass relative Aussagen über die persönlich Konzen-

tration auf die aktuelle Aufgabe unproblematisch sind.

Letztendlich gelangt der User zum letzten Bildschirm bei dem er seine Reizung eingibt. Die Art der Eingabe ist von der ersten Frage im Bogen bekannt. In der Abbildung 5.11 ist exemplarisch die negative Antwort zu sehen, die durch ein Wischen nach unten hervorgerufen wurde. Der ausgegraute Pfeilkopf am rechten mittigen Rand signalisiert, dass keine weiteren Daten erfasst werden. Bei einem Klick auf den Floating-Action-Button, wird die Eingabe gespeichert.


(a) Erste bis vierte Frage.


(b) Fünfte bis achte Frage.

Abbildung 5.11: Übersicht über den ersten Entwurf des Fragebogens in Flutter.

## 5.4 Verfeinerung des Flutter-Entwurfs

Im finalen Entwurf wurden einige Elemente in der TrackYourTinnitus App verbessert. Der Flutter-Entwurf aus dem vorherigen Abschnitt beinhaltet grundsätzlich bereits alle wichtigen Inhalte. Daher wird in diesem Abschnitt nur noch auf die Optimierungen eingegangen. In Abbildung 5.12 werden die Bildschirme gezeigt, die nicht für die Dateneingabe relevant sind. In Abbildung 5.13 wird der verbesserte Entwurf für die eingaberelevanten Seiten dargestellt.

Auf dem Homebildschirm ist der Button zum Start der Studie nach unten verlegt worden, weil er auf diese Weise bequemer zu erreichen ist. Abgesehen von dem Projekt-Logo, das den oberen Bereich der App einnimmt, hat sich nichts weiter verändert.

Der Ergebnis-Tab bekommt die Ergebnisse aus der Studie und zeigt diese an. Dabei wird jede Eigenschaft die abgefragt wird mit einem Stichwort versehen und der Wert dahinter ausgegeben.
Der Einstellungen-Tab sammelt alle Einstellungen an einem zentralen Ort. Das Menü orientiert sich dabei an dem Einstellungen-Menü wie man es von Android oder iOS kennt.
In dem rechten Bild der Abbildung 5.12 ist eine neue Seite zu finden. Sie stellt den Abschluss der Dateneingabe dar und erscheint von nun an am Ende des Fragebogens. Durch die Einführung der neuen Abschlusseite wird dem User deutlich signalisiert, dass er am Ende des Fragebogens angekommen ist. Außerdem kann so der Floating-Action-Button, der sich auf all den anderen Seiten befand, eliminiert werden. Er war recht präsent am unteren rechten Bildschirmrand zu sehen, wobei er nur am Ende der Dateneingabe benötigt wurde.

Ein Vergleich zwischen Abbildung 5.11 und 5.13 lässt zunächst kaum Veränderungen erkennen. Wesentlich verbessert hat sich die Seite zu der persönlichen Aufregung des Patienten. Der Pulsschlag des animierten Herz befindet sich nun mittig auf dem Herz und gibt die die aktuelle bpm-Rate an. Die Skala über dem Herz lässt den Anwender schnell seinen aktuell eingegebenen Wert abschätzen. Die blaue Wertanzeige verschiebt sich zusammen mit der bpm-Rate. Zusätzlich ändert sich dazu der Text auf der Wertanzeige mit den bekannten quantitativen Schlagworten. Das Zusammenspiel der Wertausgaben ermöglicht dem Anwender seine persönliche Dateneingabe schnell einzuordnen.

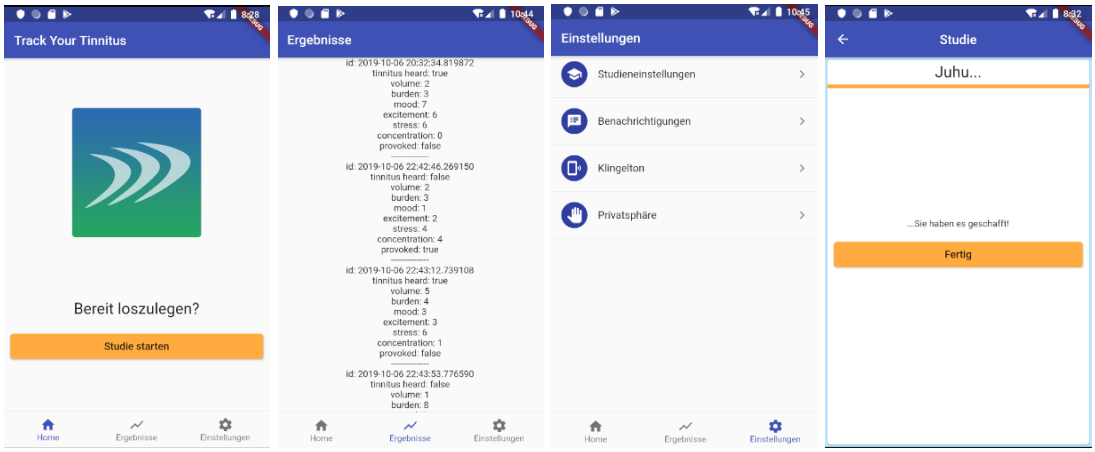


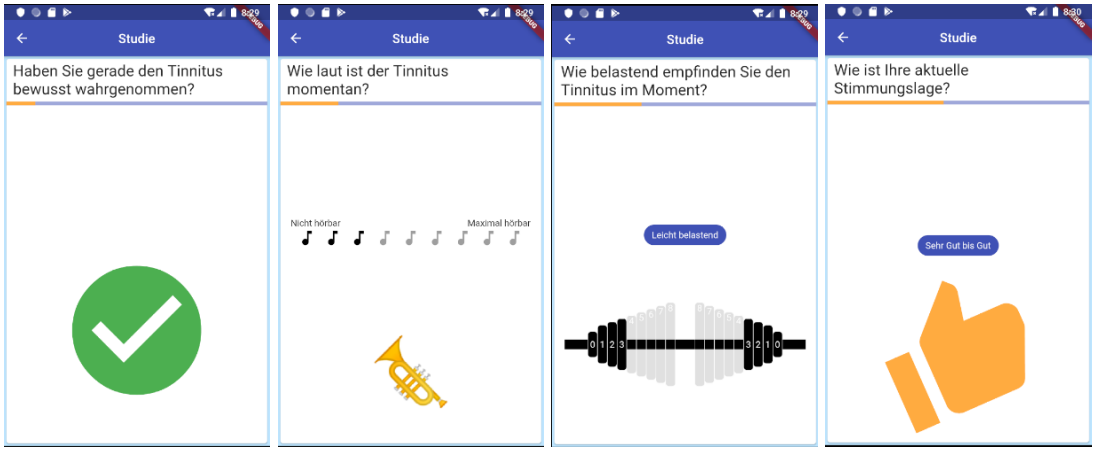

Abbildung 5.12: Übersicht Hauptnavigation und Transition zwischen den Fragen.

Die wesentlich größere Veränderung im verfeinerten Entwurf stellt die Umstellung der Navigation dar. Zuvor konnte mit einem Rechts- und Links-Swipe durch die Eingabe navigiert werden. Bei der Dateneingabe sollten die Skalen durch ein Hoch- und Runterwischen auf dem Bildschirm verändert werden. Diese beiden Bewegungen sind teilweise in Konflikt geraten. Flutter schreibt dazu auf der eigenen Homepage:

> The gesture arena is beneficial when there is only a horizontal (or vertical) drag recognizer. In that case, there will be only one recognizer in the arena and the horizontal drag will be recognized immediately, which means the first pixel of horizontal movement can be treated as a drag and the user will not need to wait for further gesture disambiguation. [30]

An Stelle des Rechts-Links-Swipe ist von nun an eine Doppeltipp durchzuführen um zur nächsten Eingabe zu gelangen. Diese Umstellung hat einen weiteren Vorteil. Der Anwender soll auf die Fragen intuitiv antworten. Während er sich auf einer der Seiten befindet, soll er die Möglichkeit haben, den Wert so häufig hoch und runter zu stellen, bis er das Gefühl hat seinen persönlichen emotionalen Wert gefunden zu haben. Geht er jedoch dann weiter, soll er nicht mehr zurückgehen können um den Wert erneut zu verstellen, weil dadurch die Intuition der Antwort verloren ginge. Aus diesem Grund wird bei dem finalen Entwurf keine Möglichkeit des Zurücknavigierens innerhalb der Dateneingabe gegeben. Die Vor- und Zurückpfeile die mittig an den Rändern der App platziert waren, wurden daher entfernt.

Die Fortschrittspunkte am unteren Rand wurden entfernt und dafür wurde eine Fortschrittsanzeige entworfen, die zwei Funktionen aufnimmt. Sie zeigt zum einen den Fortschritt innerhalb des Fragebogens an. Zum anderen dient sie als trennende Markierung zwischen Frage und Eingabe.




(a) Erste bis vierte Frage.


(b) Fünfte bis achte Frage.

Abbildung 5.13: Übersicht über den verbesserten Entwurf des Fragebogens in Flutter.

Damit der Durchlauf durch den Fragebogen optimal erfolgt, wird auf jeder Seite geprüft, ob eine Eingabe getätigt wurde. Möchte der User weitergehen, ohne einen Wert eingegeben zu haben, wird er darauf hingewiesen, wie in Abbildung 5.14 links zu sehen ist. Gleiches gilt für den Fall, dass er einen Fragebogendurchlauf begonnen hat und die Eingabe zwischendurch abbrechen möchte. Ein kompletter Datensatz ist für die nachfolgende Auswertung am sinnvollsten. Daher werden in

diesem Entwurf nur komplette Eingaben abgespeichert. Bricht der Patient frühzeitig ab, verliert er seine bisherige Dateneingabe. Der Hinweis auf der rechten Seite der Abbildung 5.14 macht dies deutlich.

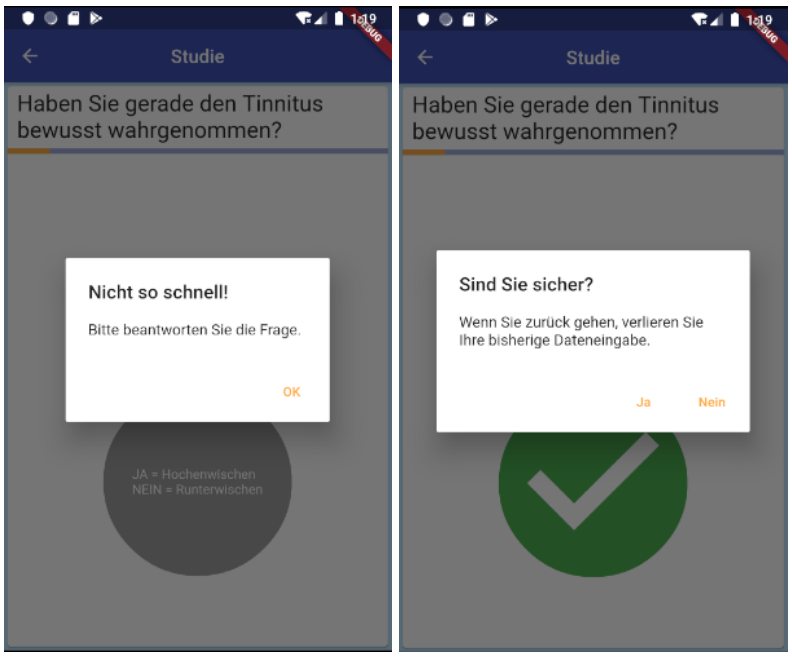



Abbildung 5.14: Eingabenüberprüfung und Hinweis über möglichen Datenverlust bei Abbruch.

# 6 Implementierung

Das Eingabekonzept zur Datenerfassung ist in Kapitel 5 ausführlich beschrieben worden. In diesem Kapitel folgt nun die Ausarbeitung der Implementierung. Genauer betrachtet werden in diesem Kapitel:

- die Paketstruktur der Implementierung,
- der Aufbau der Neugegliederten Grundnavigation,
- die Handhabung mit den in der App benutzten Gesten
- eine beispielhafte Implementierung der Skala
- die Steuerung von Animationen in Flutter
- und abschließend das Provider Package, das die Speicherung der Datenerfassung möglich macht.

## 6.1 Paketstruktur der Implementierung

Alle wichtigen Dateien, die die Implementierung betreffen befinden sich wie in Abbildung 6.1 zu sehen ist, in dem Ordner `lib`. Pakete sind in der Abbildung fett markiert, die anderen Blöcke stellen die dart-Dateien dar.
In dem lib-Ordner liegt die main-Datei, aus der die Anwendung gestartet wird, sowie drei wichtige Oberpakete. Screens enthält die Dateien, die Seitengrundstruktur der TrackYourTinnitus Anwendung aufbaut. Der Ordner gliedert sich weiter in den `study`-Ordner, der die `study`-dart-Datei enthält, den `settings`-Ordner der die Screens für die Einstellmöglichkeiten der App enthält und die `tabs_screen`-Datei, die die Bottom-Tab-Bar der Hauptnavigation aufbaut.
Der widgets-Ordner enthält die implementierten Dateien für die jeweilige Dateieingabe der acht Fragen und die Abschlussseite im Fragebogen. Die Struktur ist hier

verkürzt dargestellt, was durch die drei vertikalen Punkte deutlich gemacht wird. Für das verwendete Datenmodell wurde eine models-Ordner eingerichtet, der die einzelnen Fragebogen-Objekte verwaltet und die Gesamtdaten der Umfragen modelliert.

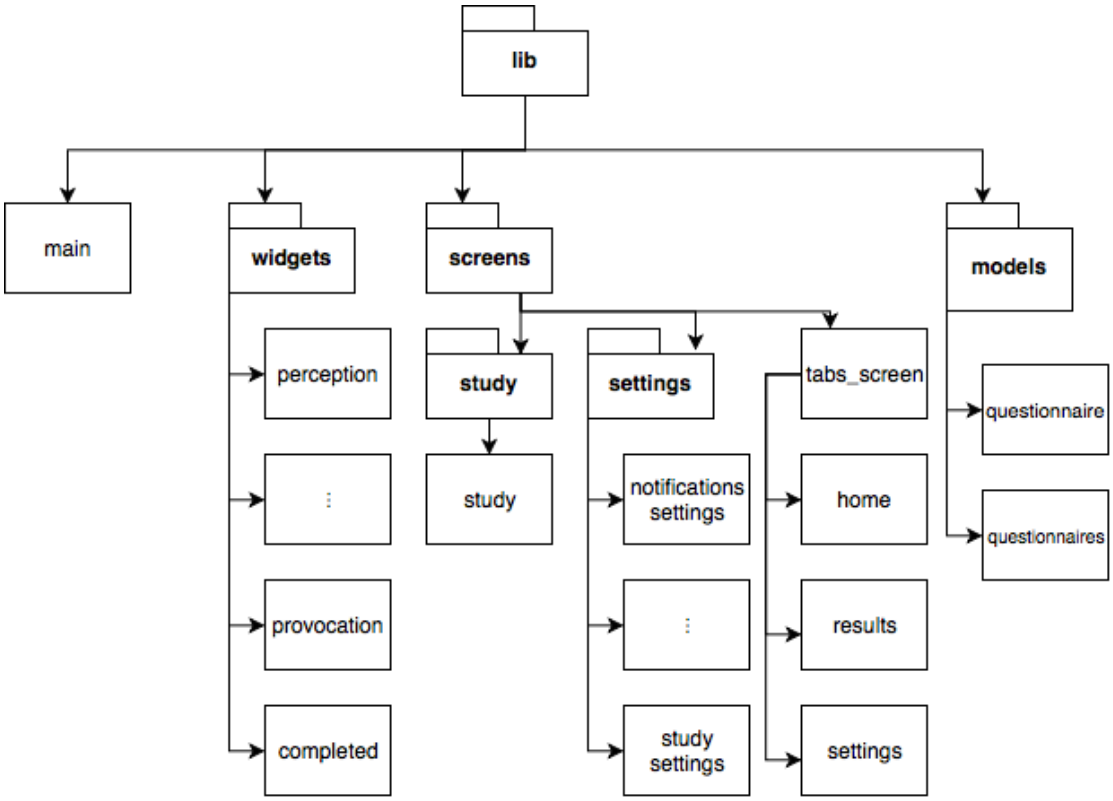



Abbildung 6.1: Allgemeine Paketstruktur der Implementierung

## 6.2 Navigation innerhalb der Anwendung

Ausgangspunkt der App bildet die main-Datei in der sich das `MaterialApp`-Widget befindet, das als `home`-Attribut das `TabsScreen`-Widget übergeben bekommt. In dem Stateful-Widget `TabsScreen` wird eine Liste den drei Tab-Seiten-Maps `_pages` implementiert. Eine Map enthält dabei den Titel der Seitennavigation und die eigentliche Seite. Die Bottom-Tab-Bar verändert bei einem Klick auf den jeweiligen Tab ihren Zustand. Dieser muss manuell vom Entwickler verwaltet werden.

Listing 6.1: Tabs der Hauptnavigation

```
1  final List<Map<String, dynamic>> _pages = [
2      {
3        'page': HomeScreen(),
4        'title': 'TrackYourTinnitus',
5      },
6      {
7        'page': ResultsScreen(),
8        'title': 'Ergebnisse',
9      },
10     {
11       'page': SettingsScreen(),
12       'title': 'Einstellungen',
13     },
14   ];
```

Dazu wird in dem State des Tabs-Screen eine Index-Variable und eine Methode angelegt, die sich um das Durchschalten der Tabs kümmern. Sobald der Index durch ein Klick auf dem Smartphone verändert wird, muss die `setState()`-Methode wie in Listing 6.2 zu sehen ist, aufgerufen werden. Der Widget-Baum wird so neue gesetzt und der veränderte Inhalt kann auf dem Bildschirm dargestellt werden.

Listing 6.2: Index Management des Tabs-Screen

```
1  var _selectedPageIndex = 0;
2    void _selectPage(index) {
3      setState(() {
4        _selectedPageIndex = index;
5      });
6    }
```

Der Tabs-Screen gibt gibt in seiner `build()`-Methode ein `Scaffold`-Widget wieder, das den gesamten Bildschirm des mobilen Endgeräts verwaltet. In dem `Scaffold` wird ein `AppBar`-Widget mit dem Titel aus der `_pages`-Liste implementiert. Ebenso erhält das `Scaffold` als `body` den jeweiligen Screen der Navigation. In Listing 6.3 ist zu erkennen, dass das `Scaffold` schon ein vorprogrammiertes Argument für die `bottomNavigationBar` enthält, dem ein namensgleiches Widget übergeben wird. In diesem registriert man bei der `onTap`-Methode die im State programmierte `_selectPage`-Methode. Auf gleiche Weise wird dem aktuellen Index die zuvor definierte Variable `_selectedPageIndex` übergeben, sodass das Widget intern erkennt welchen Screen es aus der `_pages`-Liste aufbauen muss. Als UI-Elemente für den Anwender wird in Zeile 12 dem Attribut `items` eine Liste von `BottomNavigationBarItem`-Widgets übergeben. Icon und Titel für die jeweilige

Tab-Seite sollten inhaltlich übereinstimmen, damit der Anwender nicht fehlgeleitet wird.

Listing 6.3: Scaffold des Tabs-Screen

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget._pages[_selectedPageIndex]['title']),
      ),
      body: widget._pages[_selectedPageIndex]['page'],
      bottomNavigationBar: BottomNavigationBar(
        onTap: _selectPage,
        currentIndex: _selectedPageIndex,
        [...]
        items: [
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            title: Text('Home'),
          ),
          [...]
          )
        ],
      ),
    );
  }
```

Mit dieser Implementierung steht die Grundnavigation auf App-Level, da die drei Stateless-Widgets `HomeScreen`, `ResultsScreen` und `SettingsScreen` aufgebaut werden. Der Ergenisse-Tab gibt in seiner `build()`-Methode das `ResultList()`-Widget wieder. In diesem befinden sich die Ergebnisse der durchgeführten Fragebögen. Im `SettingsScreen` ist ein `ListView`-Widget implementiert, dass die Ansicht der Einstellungen auf den Bildschirm rendert.

Ebenfalls wichtig für das Verständnis der Gesamtnavigation ist der Aufbau der Studiennavigation. Hier wird ein ähnliches Muster aufgebaut wie für den `TabsScreen`. Das Stateful-Widget `StudyScreen` erhält eine Liste mit Maps die jeweils als Werte eine `String` und `dynamic` erhalten. Der `String` ist die Variable für die Fragen im Fragebogen und das `dynamic` implementiert das jeweilige Widget für die Dateneingabe. Wie aus dem Listing 6.4 ersichtlich ist, gelangt man mit dem Tag „question“ an die Frage und mit „data“ an das Aktionswidget für die Datenerfassung. Die Seiten werden mit der Variable `questionIndex` weitergeschaltet, da in der `setState()`-Methode der Index bei Ausführung der Geste des Weiterkommens innerhalb des Fragebogens um eins inkrementiert wird. Beachtet werden muss hier, dass sobald

der Fragebogen durchlaufen ist der Index nicht weiter erhöht werden darf, weil sonst ein Exception geschmissen wird. Aus diesem Grund muss der Index vor Inkrementierung im Code mit `questionIndex < questionData.length - 1` überprüft werden.

Listing 6.4: Liste der Maps für die Studiennavigation

```
1  List<Map<String, dynamic>> questionData = [
2      {
3        'question': 'Haben Sie gerade den Tinnitus bewusst wahrgenommen?',
4        'data': Perception(),
5      },
6      {
7        'question': 'Wie laut ist der Tinnitus momentan?',
8        'data': Volume(),
9      },
10     [...]
11     {
12       'question': 'Fuehlen Sie sich gerade gereizt?',
13       'data': Provocation(),
14     },
15     {
16       'question': 'Juhu...',
17       'data': Completed(),
18     },
19   ];
```

### 6.2.1 Flutter-Navigator

Wenn die Seitennavigation im Code nicht über Maps und Indizes verwaltet wird folgt die Navigation einem weiteren wichtigen Muster. Beispielhaft wird die Implementierung der Einstellungen-Seite vorgestellt.
Der `SettingsScreen` der über dem `TabsScreen` aufgebaut wird, enthält die `ListView`, die die Ansicht der Einstellungen der App als `ListTile` implementiert. Ein `ListTile` sieht dabei ganz konkret wie in Listing 6.5 aus. Interessant ist hierbei der Navigator aus Zeile 8. Bei einem Klick auf das `ListTile` wird die `onTap()`-Methode ausgeführt, in dieser dann der Navigator den neuen Screen auf den App-Stack pusht. Im Listing 6.5 gelangt man mit dem Tap zu den Studieneinstellungen. Dies ist nur möglich, weil zuvor im `StudySettingsScreen` der `routeName` als `static const routeName = '/study-settings';` festgelegt wurde. So kann über die Klassenvariable `routeName` immer auf den Screen zugegriffen werden. Damit Futter die

routes korrekt verarbeiten kann, müssen diese auf Ebene der MaterialApp registriert werden. Aus diesem Grund besitzt das MaterialApp-Widget das Argument routes. Sobald die routes wie in Listing 6.6 registriert sind, können sie über ihren statischen Namen mit dem Flutter-internen Navigator gepusht werden. Genau dieses Verhalten wurde im Listing 6.5 in der Zeile 8 implementiert. Soll die Seite vom Stack geschoben werden, so kann die pop()-Methode des Navigators verwendet werden.

Listing 6.5: ListTile im Einstellungen-Bildschirm

```
1  ListTile(
2    leading: CircleAvatar(
3      child: Icon(Icons.school),
4    ),
5    title: Text('Studieneinstellungen'),
6    trailing: Icon(Icons.navigate_next),
7    onTap: () {
8     Navigator.of(context).pushNamed(StudySettingsScreen.routeName);
9    },
10 ),
```

Listing 6.6: Routes auf MaterialApp-Ebene

```
routes: {
  StudyScreen.routeName: (ctx) => StudyScreen(),
  MyHomePage.routeName: (ctx) => MyHomePage(),
  StudySettingsScreen.routeName: (ctx) => StudySettingsScreen(),
  PrivacySettingsScreen.routeName: (ctx) => PrivacySettingsScreen(),
  NotificationsSettingsScreen.routeName: (ctx) => NotificationsSettingsScreen(),
  RingtoneSettingsScreen.routeName: (ctx) => RingtoneSettingsScreen(),
},
```

## 6.3 Die Gesten der Anwendung

Gesten und Interaktionen werden mit dem Bildschirm in dieser Implementierung mit dem GestureDetector verwaltet. Wie in Abschnitt 5.4 erläutert, wurde jeweils eine Geste für das Durchschalten durch den Fragebogen entworfen und eine für die Dateneingabe.

Das Durchschalten des Fragebogens ist in der study_screen-Datei implementiert, da für jede Eingabe die gleiche Geste gilt. Die Geste soll auf dem gesamten Bildschirm der jeweiligen Frage verfügbar sein, daher wird der GestureDetector direkt

an den `body` des `Scaffold`s übergeben. Innerhalb des `GestureDetector` ist die Methode `onDoubleTap` implementiert.

Listing 6.7: GestureDetector zum Durchschalten des Fragebogens

```
1  @override
2  Widget build(BuildContext context) {
3      [...]
4      return Scaffold(
5      [...]
6        body: GestureDetector(
7          onDoubleTap: () {
8            if (questionIndex < questionData.length - 1)
9              setState(() {
10               if (questionIndex == 0 && questionnaire.tinnitusheard != null ||
11                   questionIndex == 1 && questionnaire.volume != null ||
12                   questionIndex == 2 && questionnaire.burden != null ||
13                   questionIndex == 3 && questionnaire.mood != null ||
14                   questionIndex == 4 && questionnaire.excitement != null ||
15                   questionIndex == 5 && questionnaire.stress != null ||
16                   questionIndex == 6 && questionnaire.concentration != null ||
17                   questionIndex == 7 && questionnaire.provoked != null) {
18                 questionIndex += 1;
19                 increaseStudyProgress();
20               } else {
21                 showDialog(
22                     context: context,
23                     builder: (ctx) => AlertDialog(
24                           title: Text('Nicht so schnell!'),
25                           content: Text('Bitte beantworten Sie die Frage.'),
26                           actions: <Widget>[
27                             FlatButton(
28                               child: Text('OK'),
29                               onPressed: () {
30                                 Navigator.of(ctx).pop();
31                               },
32                             )
33                           ],
34                         ));
35               }
36             });
37           _controller.forward().whenCompleteOrCancel(() {
38             _controller.reverse().orCancel;
39           });
40         },
41     [...]
42   );
43 }
```

In der Methode wird der aktuelle Zustand der Fragebogenseiten verwaltet. Daher befindet sich in Zeile 9 die `setState()`-Methode die den Widgetbaum zum Rebuild

zwingen kann, wenn durch die Eingabe des Benutzer Veränderungen stattfinden sollen. Dies geschieht nur wenn die Bedingung aus Zeile 8 zutrifft. Wie oben erklärt prüft sie ob das Ende des Fragebogens erreicht wurde. Ebenso wird durch die Bedingungen in den Zeilen 10 bis 17 überprüft, ob eine Dateneingabe vorgenommen wurde. Falls dies nicht der Fall ist, wird dem Anwender eine Alert-Dialog angezeigt, dass er erst weitergehen kann, sobald er die Frage auf dem Bildschirm beantwortet hat. Zum Schluss soll bei jedem Double-Tap eine Animation ausgeführt werden. Diese simuliert das Kippen der Fragekarte. Die Funktion die den Animation-Controller ausführt, muss daher in der `onDoubleTap`-Methode aufgerufen werden.

Für die Dateneingabe wird im Konzept eine weitere Geste reserviert. Durch vertikales Wischen soll die Dateneingabe minimiert bzw. maximiert werden. Um dies zu implementieren wird eine Methode benötigt, die ein konstantes, vertikales Drag-Update liefert. Für jedes Dateneingabe-Widget wird ein `GestureDetector` implementiert um die jeweiligen Feinheiten auf der Seite zu beachten. Somit wären in Zukunft auch eine Erweiterungen der Eingabe für unterschiedliche Eingabe-Widgets möglich, was bei dieser Implementierung jedoch explizit nicht getan wird. Der Fluss der Eingabe soll einfach gehalten werden. Die Implementierung dieser Geste wird exemplarisch an der Frage über die aktuelle Konzentration des Patienten analysiert.

Im `GestureDetector` bekommt die Methode `onVerticalDragUpdate` die `dragUpdateDetails` als Parameter übergeben. Auf diese Weise kann die Richtung und der Offset der Wischbewegung des Fingers auf dem Bildschirm erkannt werden. In dem Fall der TrackYourTinnitus Dateneingabe werden die y-Koordinaten des Bildschirms angefordert. Dies passiert in Zeile 7 des Listings 6.8 mit `dragUpdateDetails.delta.dy.round()` und zeitgleich werden sie zu Integern gerundet. Insgesamt wird der Offset in der `concentrationRate` aufsummiert. Da der Bezugspunkt für das Koordinatensystem links oben in der Ecke ist, muss man die y- Koordinaten subtrahieren, um positive Werte beim Hochwischen zu erhalten. Mit dieser Summe wird die Skala des Eingabe gebildet und die Wischgeste wird auf einen Wert gemappt. Bei der Implementierung stellen jeweils 60 Punkte ein Intervall auf der Skala dar, sodass der Maximalwert bei 540 liegt. Durch die Überprüfungen wird in Zeile 6, 8 und 10 wird sichergestellt, dass der Wert der Eingabe innerhalb der Skala bleibt. Der erfasste Wert der Dateneingabe wird dann in der `setState()`-Methode in Zeile 13 für die Aktivierung der durchlaufenden Skala gesetzt. Auf dem Bildschirm darf immer nur das aktuelle Intervall aufleuchten, daher wird die Variable

`activateMap`, die eine Liste mit booleschen Werten ist, nach dem 1-aus-n-Code gesetzt. Für die Skala mit neun Werten existieren folglich 8 weitere Abfragen im Code. Sobald der Finger vom Display gehoben wird, liefert `dragUpdateDetails` keinen neuen Daten und die Skala bleibt bei ihrem aktuellen Wert stehen.

Listing 6.8: Vertikale Drag-Bewegung am Beispiel der Frage über die aktuelle Konzentration

```
@override
Widget build(BuildContext context) {
  [...]
  return GestureDetector(
    onVerticalDragUpdate: (dragUpdateDetails) {
      if (concentrationRate > -1 && concentrationRate < 541) {
        concentrationRate -= dragUpdateDetails.delta.dy.round();
      } else if (concentrationRate < 0) {
        concentrationRate = 0;
      } else {
        concentrationRate = 540;
      }
      setState(() {
        if (concentrationRate < 61) {
          activateMap[0] = true;
          activateMap[1] = false;
          activateMap[2] = false;
          activateMap[3] = false;
          activateMap[4] = false;
          activateMap[5] = false;
          activateMap[6] = false;
          activateMap[7] = false;
          activateMap[8] = false;
          concentrationRateValue = 0;
       }
       [...]
       });
       [...]
    },
  ),
}
```

## 6.4 Implementierung der Skalen

Im vorherigen Abschnitt wurde bei der `onVerticalDragUpdate`-Methode das Thema der Skalen schon nebenläufig angeschnitten. Aufgrund der Wichtigkeit der Skalen, sollen sie in einem eigenen Abschnitt des Implementierungskapitels beschrie-

ben werden. Im Entwurf wurde klar, dass es im TrackYourTinnitus zwei Arten von Eingaben gibt. Auf der einen Seite gibt es Eingaben für die Entscheidungsfragen und auf der anderen die für die quantitativen Eingaben. Sie sind ein wenig komplexer, weil die Skala Minimal- und Maximalwert erkennen lassen muss, dazu soll sie sich kontinuierlich verändern und ein Erlebnis beim Nutzer hervorrufen. Beispielhaft wird die Implementierung für die Skala des persönlichen Stress des Patienten gezeigt. Die Skalen für die übrigen Eingabe-Widgets folgen in der Implementierung grundsätzlich dem gleichen Muster, können jedoch minimal variieren.

Die Skalen wurden jeweils mit neun Intervallen implementiert. Auf diese Weise gibt es einen Minimal-, Maximal- und Mittelwert. Hinzukommen noch drei feinere Abstufungen zwischen den Minimal- und Mittelwert und Maximal- und Mittelwert. Dem Patienten wird so genügend Freiraum gegeben, seine aktuellen Wert einzugeben, ohne ihn mit zu vielen Werten zu überfordern. Die Eingabe des Patienten interagiert mit der Bildschirmausgabe über eine boolesche Liste, die mit der Wischgeste nach dem aus Abschnitt 6.3 bekannten Muster `true` oder `false` gesetzt wird.

Listing 6.9: Verwaltung Stresspunkte bei der Erfassung des persönlichen Stress

```
1  onVerticalDragUpdate: (dragUpdateDetails) {
2    range -= dragUpdateDetails.delta.dy.round();
3    if (range > -1 && range < 541) {
4      for (var i = 0; i < showStressPoints.length - 1; i++) {
5        if (range >= ((Stress.one + i) * intervalStep)) {
6          setState(() {
7            showStressPoints[i + 1] = true;
8          });
9        } else if (range < ((Stress.one + i) * intervalStep)) {
10         setState(() {
11           showStressPoints[i + 1] = false;
12         });
13       }
14     }
15   } else if (range < 0) {
16     range = 0;
17   } else if (range > 540) {
18     range = 540;
19   }
20   [...]
21 }
```

Der Variablenname für die Liste im Listing 6.9 ist `showStressPoints` und wird abhängig vom Hoch- oder Runterwischen des Benutzers in der `for`-Schleife in Zeile 4 gesetzt. Zuvor wurde in der Methode `onVerticalDragStart` der erste Index und

eine boolesche Variable `tapped` auf `true` gesetzt. Eine `tapped`-Variable wird auf allen Eingabebildschirmen verwendet, damit die Skala das Kriterium der Unbefangenheit aus dem Entwurf implementiert. Andernfalls würde sie einen initialen Wert anzeigen, was unerwünschte Verzerrungen in der Antwort des Patienten hervorrufen könnte.

Die Skala zum Messen des aktuellen Stressniveaus besteht zum einen aus den im Halbkreis angeordneten Chips und zum anderen aus den Stressblitzen.

Listing 6.10: Chip für die textuelle Anzeige der Skala

```
Chip(
  label: Text(
    'Nicht gestresst',
    style: TextStyle(
      fontSize: 10,
      color: (range < 61) ? Colors.white : Colors.black,
    ),
  ),
  backgroundColor: (range < 61)
      ? Theme.of(context).primaryColor
      : Colors.black12,
),
```

Listing 6.11: Stressblitze für die kumulative Skala des persönlichen Stress

```
Positioned(
  bottom: 105,
  left: 100,
  child: Transform.rotate(
    angle: (math.pi * (90 / 360)),
    child: Icon(
      Icons.show_chart,
      size: 40,
      color: showStressPoints[0] ? Colors.red : Colors.grey,
    ),
  ),
),
```

Diese beiden UI-Elemente befinden sich in einem `Stack` und werden mit dem `Positioned` innerhalb des `Stack`s platziert. Um nun die Skala Durchzuschalten müssen binäre Ausdrücke in die Widgets eingebaut werden. Zu beachten ist, dass die Chips one-hot-kodiert sind, die Stressblitze hingegen kumulieren sich auf. Aus diesem Grund muss für die Chips, wie in Listing 6.10 gezeigt, die Variable `range` im binären Ausdruck verwendet werden. Die `range`-Variable ist die Summe der y-Koordinatenveränderung. Um die Skala der kummulierenden Stressblitze aus Lis-

ting 6.11 sukzessiv einzufärben, kann die Liste `showStressPoints` im binären Ausdruck verwendet werden.

## 6.5 Animationen

Um die Anwendung lebendiger wirken zu lassen befinden sich Animationen in der Implementierung. Die Herzanimation auf der Seite zur Eingabe der aktuellen Aufregung ist dabei sehr präsent. Sie wird daher als Beispiel herangezogen, wie in der Implementierung Animationen mit Flutter umgesetzt wurden.

Ohne die Animation wäre das Herz ein starres Icon, dass sich nicht bewegen könnte. Um eine herzschlagähnliche Bewegung zu animieren, ist es möglich das Herzicon in eine `ScaleTransition` zu verschachteln. Das `ScaleTransition`-Widget nimmt neben dem Kind-Widget unter anderem das Argument `scale` auf. `Scale` erwartet eine `Animation` vom `Typ` double. Die `Animation` und ein `AnimationController` müssen außerhalb der `build`-Methode im State instanziiert werden. Wie in Listing 6.12 zu sehen ist, muss die `initState()`-Methode überschrieben und die `Animation` und `AnimationController` in ihr initialisiert werden.

Listing 6.12: Initialisierung der Animation

```
1 @override
2   void initState() {
3     super.initState();
4     _controller = AnimationController.unbounded(
5       vsync: this,
6     );
7     _animation =
8         CurvedAnimation(parent: _controller, curve: Curves.elasticInOut);
9   }
```

Das Herz ist nicht konstant animiert, stattdessen kann der Patient seine aktuelle Aufregung einstellen, wodurch sich der Puls des animierten Herz verschnellert. Aus diesem Grund erhält der `AnimationController` noch die `unbound`-Methode. Sie legt zunächst keinen festen Wert für die Skalierung der Animation fest. Das Argument `vsync` ist wichtig, damit die Animation weiß, wann ein es einen Frame-Update innerhalb der App gibt und der `AnimationController` das Icon neu zeichnen kann. Damit `vsync` mit `this` einen Wert zugewiesen bekommen kann, muss

die State-Klasse des Stateful-Widgets das `TickerProviderStateMixin` beinhalten. Mxins sind in Flutter ähnlich wie Interfaces in Java. Mixins können implementierte Methoden besitzen.[1]

Die Animation wird mit einer `CurvedAnimation` initialisiert. Ihr wird der `AnimationController` übergeben und dazu noch angegeben welche `curve` die Animation durchführen soll. Es gibt viele `curves` die `Curves.elasticInOut` eignet sich hier gut. Eine Übersicht über die verschiedenen `curves` können in der Fußnote gefunden werden.[2] Sobald Animationen im Code verwendet werden müssen diese wie im Listing 6.13 zu sehen ist geschlossen werden. Dazu wird die `dispose()`-Methode des States überschrieben.

Listing 6.13: Dispose-Methode für den AnimationsController

```
1 @override
2   void dispose() {
3     \_controller.dispose();
4     super.dispose();
5   }
```

Die Animation soll abhängig von dem angegeben Wert schneller oder langsamer schlagen. Daher wird in der `onVerticalDragUpdate`-Methode neben den Intervallstufen für die sich über dem Herz befindende Schiebeanzeige auch der `_controller` geregelt. Wie der Name es sagt, kontrollieren `AnimationController` eine Animation. Dabei kann er sie einmalig vorwärts mit `forward()`, oder rückwärts mit `reverse()` abspielen. Soll die Animation jedoch die ganze Zeit laufen, muss die Methode `repeat()` auf den `AnimationController` angemeldet werden.
Im Listing 6.14 wird es im Wiederholungsmouds implementiert. Für die Werte der Animation gilt, je unaufgeregter der Patient ist, desto langsamer schlagt sein Herz. Das heißt, dass das `period`-Argument eine lange `Duration` zugewiesen bekommen muss. Die `min`- und `max`-Angaben sind für die Skalierung des Icons. In Zeile 3 wird das Herz im langsamsten Fall der Animation von einem Wert von 0.65 auf 1.0 während einer Periode von 1000 Millisekunden vergrößert. In der nächsten Stufe der Aufregung steigt die Vergrößerung um 0,01, die Periode nimmt aber um 100 Millisekunden ab. Dadurch wird der Herzschlag schneller animiert. Die `intervall`-Variable in Zeile 4 schiebt den Balken der Skala über dem Herz weiter und die

[1] https://en.wikipedia.org/wiki/Mixin
[2] https://api.flutter.dev/flutter/animation/Curves-class.html

`heartFrequence` wird auch abhängig von der Dateneingabe gesetzt. Im langsamsten Fall der Animation sind es 45 bpm.

Listing 6.14: Aktivierung eines unbound-AnimationController

```
setState(() {
  if (sum > -1 && sum < 61) {
    _controller.repeat(min: 0.65, max: 1, period: Duration(milliseconds: 1000));
    interval = 0;
    heartFrequence = 45;
    } else if (sum > 60 && sum < 121) {
    _controller.repeat(min: 0.64, max: 1, period: Duration(milliseconds: 900));
    [...]
  }
}
```

## 6.6 Provider Package und Questionnaire

Die Daten die auf den Seiten des Fragebogens eingegeben werden, sollen in der App verfügbar sein, sodass sie auf dem Ergebnis-Bildschirm genutzt werden können. Die Schwierigkeit bei der Implementierung besteht hierbei, dass in jedem der verschieden Eingabe-Widgets Daten entstehen und diese dann gebündelt an den Ergebnis-Tab geschickt werden müssen.

In dem Ordner models, definiert die Datei questionnaire.dart einen einzelnen Fragebogen.Der Konstruktor für einen `Questionnaire` enthält die in Listing 6.15 definierten *named arguments*.

Listing 6.15: Konstruktor eines Questionnaires

```
Questionnaire({
    this.id,
    this.tinnitusheard,
    this.volume,
    this.burden,
    this.mood,
    this.excitement,
    this.stress,
    this.concentration,
    this.provoked,
  });
```

In der Datei wird die `resetQuestionnaire`-Methode implementiert, damit von einem Ort aus auf sie zugegriffen werden kann. Eine weitere Datei, `questionnaires`

`.dart`, befindet sich in dem selben Ordner und enthält alle einzelnen Fragebögen. Implementiert wird dies mittels einer Liste vom Typ `Questionnaire`. Die Questionnaires-Klasse implementiert zudem die Methode `addQuestionnaire()`, die einen Fragebogen der Liste hinzufügt.

Mit dem Provider-Package kann man das Datenmodell App-weit verfügbar machen.[3] Dazu wird es noch vor dem `MaterialApp`-Widget in dem Widget-Baum eingefügt. Wie der Name sagt, stellt das Provider-Package Daten *zur Verfügung*. In der TrackYourTinnitus App soll sowohl auf `questionnaires.dart`, als auch auf `questionnaire.dart` zugegriffen werden. Daher wird wie in Listing 6.16 Zeile 4 zu sehen ist, ein MultiProvider implementiert, der die beiden Datenmodelle als `Change- NotifierProvider` als Liste übergeben bekommt.

Listing 6.16: MultiProvider

```
1  @override
2  Widget build(BuildContext context) {
3    return MultiProvider(
4      providers: [
5        ChangeNotifierProvider.value(
6          value: Questionnaire(),
7        ),
8        ChangeNotifierProvider.value(
9          value: Questionnaires(),
10       )
11     ],
12     child: MaterialApp(
13     [...]
14   ),
15 ),
```

Im Verlauf des TrackYourTinnitus Fragebogens werden nun die Daten gesammelt und in einen `questionnaire` gespeichert. Durch den folgende Code wird nun der Kommunikationskanal zwischen den jeweiligen Seiten geöffnet:

```
final questionnaire = Provider.of<Questionnaire>(
  context,
  listen: false,
);
```

Durch die Typ-Angabe weiß der `Provider`, dass es sich um ein `Questionnaire`-Objekt handelt und nicht um ein `Questionnaires`. Mit der Variable `questionnaire`

[3] https://pub.dev/packages/provider

kann dann auf jedes einzelnes Attribut innhalb der `Questionnaire`-Klasse zugegriffen werden.

Um beispielsweise die Lautstärke des Tinnitus auf der zweiten Seite des Fragebogens auszulesen, wird die Liste der schwarzmarkierten Noten ausgelesen, denn sie stellt hier die Skala dar.

```
questionnaire.volume =
            blackNotes.lastIndexWhere((value) => value == true);
```

Dart stellt viele Methoden zur Verarbeitung von Listen zur Verfügung. Unter anderem existiert eine Methode die den letzten Index eines Merkmals wiedergibt. Die Liste `blackNotes` speichert boolische Werte über den Zustand, d.h. ob die Noten an der i-ten Stelle schwarz oder grau sind. Daher eignet sich die Methode `lastIndexWhere()` optimal, um den Wert als Integer auszulesen und im `volume`-Attribut der Variable `questionnaire` zu speichern. Bei weiterem Durchschalten durch den Fragebogen werden immer mehr Daten in den `questionnaire` gespeichert.

Auf dem letzten Screen des Fragebogens werden beide `Provider` geladen, sodass ein `questionnaire`- und ein `questionnaires`-Objekt existiert. Wenn der User dann auf den Button „Fertig“, drückt löst das die `onPressed()`-Methode aus Listing 6.17 aus.

Listing 6.17: onPressed-Methode des letzten Bildschirms des Fragebogens

```
1  onPressed: () {
2    questionnaires.addQuestionnaire(
3      Questionnaire(
4        id: DateTime.now().toString(),
5        volume: questionnaire.volume,
6        burden: questionnaire.burden,
7        concentration: questionnaire.concentration,
8        provoked: questionnaire.provoked,
9        excitement: questionnaire.excitement,
10       mood: questionnaire.mood,
11       tinnitusheard: questionnaire.tinnitusheard,
12       stress: questionnaire.stress,
13     ),
14   );
15   questionnaire.resetQuestionnaire();
16   Navigator.of(context).pop();
17 },
```

Dem `questionnaires`-Objekt wird mithilfe der Implementierung der Methode `addQuestionnaire()` in den Zeilen 5 bis 12 die Attribute der einzelnen Fragebogendurchlaufs hinzugefügt und gespeichert. Für die den nächsten Fragebogendurchlauf, wird das `questionnaire`-Objekt in Zeile 15 zurückgesetzt. Schließlich wird mit dem Navigator zum Home-Tab zurückgekehrt.

Dadurch dass die Methode `addQuestionnaire()` die Methode `notifylisteners()` implementiert, bemerkt der Ergebnis-Tab, die neuen Daten und kann sie direkt in dem implementierten Widget `ResultList` über das `questionnaires`-Objekt auslesen und darstellen.

# 7 Fazit

Ziel dieser Bachelorarbeit war es, ein progressives Eingabekonzepte zur Datenerfassung auf mobilen Endgeräten zu entwickeln. Die Entwicklung des Konzepts geschah anhand des universitären TrackYourTinnitus Projekts.

## 7.1 Zusammenfassung

Den Einstieg in die Thematik der Arbeit bildete das Kapitel 2 mit dem Überblick über das bestehende TrackYourTinnitus Projekt. Dort wird erläutert, dass die Eingabe zur Zeit aus bekannten UI-Elementen wie einem Slider, oder Radio-Button besteht. Diese Elemente wurden bei dem Entwurf zur Datenerfassung eliminiert, da Progressivität voraussetzt, bestehende Muster zu überdenken. Selbstverständlich wurden die Minimalbedingungen, die für die Dateneingabe des Benutzers notwendig sind, übernommen. Genauer gesagt bedeutet das:

1. Jede Eingabe muss mit einer Skala einhergehen, die den Minimal- und Maximalwert der Skala erkennen lässt. Dadurch wird die eigene Dateneingabe relativiert.
2. Die Skala muss zudem ihre Unbefangenheit sicherstellen, da es sich um eine subjektive Datenerfassung handelt. Es darf kein voreingestellter Wert existieren, um den Patienten nicht zu beeinflussen.

Aufgrund der Tatsache, dass die Eingabe der Datenerfassung im Mittelpunkt steht, wird die TrackYourTinnitus App, so wie sie im AppStore zum Download zur Verfügung steht, gänzlich neu gegliedert. Die Neugliederung der Anwendung stellt die Vorbedingung zur Entwicklung eines progressiven Eingabekonzepts. Die Studie zur Tinnitusüberwachung befindet sich jetzt auf einem eigenen Home-Bildschirm, der dem Anwender das Gefühl eines klaren Startpunktes innerhalb der App gibt. Der

Home-Bildschirm gehört zur neuen Hauptnavigation der App, die als Bottom-Tab-Bar implementiert ist. Neben dem Home-Tab ist ein Ergebnis- und Einstellungen-Tab am unteren Rand der Applikation zu finden. Weiterhin wurde beachtet, dass Smartphone-Bildschirme in der Entwicklung größer wurden. Da das Haupteingabemedium bei der Tinitus-Anwendung der Daumen ist, wurde ein Umdenken in der App-Navigation notwendig. So konnte nun sichergestellt werden, dass der Anwender trotz größeren Bildschirmen mit einem Daumen ausreichend agieren kann.

Gleiches Umdenken galt für die Daten des Fragebogens. Das Konzept umfasst nun eine kartenähnliche Stack-Anordnung der einzelnen Fragen. Das bietet zum einen den Vorteil, dass die Eingabe übersichtlich strukturiert ist und der Anwender nicht mit der Gesamtheit aller Fragen konfrontiert wird. Nach Abschluss seiner Eingabe geht der Anwender bequem mithilfe eines Double-Taps weiter und widmet sich dann mit seiner *vollen Aufmerksamkeit* der nächsten Erhebung seines emotionalen Zustands.

Zum anderen hat der kartenähnliche Stack-Aufbau zur Folge, dass man für die eigentliche Eingabe sehr viel mehr Platzt auf dem Smartphonebildschirm hat, ohne dafür Scrollen zu müssen. Wesentliche Prinzipien des *Usability Engineering*, wie sie im Kapitel 4 erklärt werden, können daher Beachtung finden. Der User wird nun auf eine spielerische Art und Weise durch die Eingabe geführt. Die Tatsache, dass er an einem Tinnitus leidet, soll ihm nicht den Spaß bei der Benutzung der App nehmen. Sie soll dem Benutzer durch die generierten Daten langfristig helfen, statt Frustration hervorzurufen. Die verschiedenen Aktionen, wie zum Beispiel Sport treiben, ein Instrument spielen oder Matheaufgaben lösen, sollen ein gewissen *Joy of Use* hervorrufen, um eine tiefere Verwicklung des Patienten mit der App zu erreichen. Eine einheitlich konsistente Eingabebewegung, das Hoch- und Runterwischen, sorgt dafür dass der Benutzer nicht überfordert wird. Die Steuerung der Anwendung bleibt auf diese Weise einfach und effizient. Denkbare Dateneingaben von unterwegs, wobei man bei mobilen Endgeräten davon ausgehen muss, sind so problemlos möglich. Double-Tap, Wischen und Neugliederung des Fragebogens machen die Tinnitus-App jetzt wieder vollständig im typischen Einhandmodus, bei dem der Daumen die Haupteingabe verrichtet, möglich. Das Kapitel 5 über den Entwurf ebnet den Weg von einer reinen Fragebogen-App hin zu einer progressiven spielerischen Anwendung.

Die benutzte Computersprache ist Dart und das Framework Flutter. Beides stammt

von Google und funktioniert reibungslos zusammen. Der Programmierstil mit Flutter bedarf einer gewissen Umgewöhnung, denn es wird nicht prozedural, sondern deklarativ programmiert. Flutter basiert auf Widgets, die sozusagen die kleinste eingenständige UI-Komponente einer gesamten Benutzeroberfläche darstellen. Durch aggressive Zusammensetzung der Widgets entsteht der Widget-Baum, der die Blaupause für die Render-Objekts darstellt. Diese übernehmen letztendlich das Zeichnen auf dem Gerätebildschirm. Das Konzept der Stateful- und Stateless-Widgets muss gut verstanden werden, um Projekte in Flutter zu implementieren. Besonders wichtig ist dabei die `setState()`-Methode, denn mit ihr kann man tatsächlich den Zustand der Benutzeroberfläche auf dem Bildschirm der Applikation verändern. Das Kapitel 3 über die Grundlagen in Flutter stellt daher eine elementare Voraussetzung zum Verständnis dieser Arbeit dar.

## 7.2 Ausblick

Ein progressives Eingabekonzept zur Datenerfassung auf mobilen Endgeräten ist im Zuge dieser Bachelorarbeit entstanden. Darüberhinaus wurde das Konzept mithilfe von Flutter entwickelt. Flutter ist eine progressive Technologie und neu auf dem Markt der plattformübergreifenden Softwarelösungen. Nach der Einarbeitung in das Framework, ist es sehr zielstrebig in seiner Benutzung. Der weitere große Vorteil ist die Plattformunabhängigkeit. Für die Implementierung wurde zunächst ein Android-Emulator und später ein Smartphone mit Android als Betriebssystem verwendet. Der Android-Emulator wurde einem iOS-Simulator bevorzugt, weil der Emulator wesentlich flüssiger auf dem Entwicklercomputer lief. Dennoch wurde die App kurz auf einem iOS-Simulator getestet. Bis auf den genannten Nachteil der Zähflüssigkeit, wurde keine große Veränderung zum Android-Emulator festgestellt. Aufgrund der Zugänglichkeit wurde das Android-Smartphone gegenüber dem iPhone ausgewählt. Somit ist die Anwendung für Android Geräte gut getestet. Ein ausgiebiges Testen der Anwendung für iOS-Hardwaregeräte erscheint für die weitere Etablierung sinnvoll. Da der Test am iOS-Simualtor erfolgreich verlief, sollte das Testen auf einem echten Gerät keine großartigen Probleme mit sich bringen.

Die App ist lokal funktionstüchtig, allerdings ist bisher noch kein Backend angebunden. Dieser Schritt fehlt noch, um die Fragebogendaten auf einem Server zu

speichern und der Tinnitus Research Initiative zur Verfügung zu stellen. Durch die bestehende Infrastruktur des Track your Tinnitus Projektes in Kombination mit den internen Vorbereitungen dieser vorliegenden Implementierung, erscheint die Anbindung an ein Backend unproblematisch. Die Fragebogendaten werden innerhalb der Neuimplementierung in einem Fragebogen-Objekt gesammelt und gespeichert. Bei der Speicherung im Abschlussbildschirm des Fragebogens werden die Ergebnisse anschließend an den Ergebnis-Tab innerhalb der App gesendet. Dort wäre die Einbindung eines http-Request denkbar.

Die Ergebnisse eines Fragebogens werden bisher nur rudimentär abgespeichert und nicht anschaulich aufbereitet. Diese Arbeit widmet sich der Datenerfassung und nicht dem Output von Daten. Für den User wäre es jedoch hilfreich, wenn er seine eingegebenen Daten im Anschluss mit Graphen, oder ähnlichen visuellen Hilfsmitteln analysieren könnte.
Auch hier gilt, dass die Vorarbeit dank des implementierten Provider-Package gemacht ist. Alle Daten sind in den Ergebnissen verfügbar und können aus dem Datenmodell `questionnaires` ausgelesen werden.

Für die Entwickler einer App ist ihre eigene Implementierung und deren Benutzung verständlich. Das mag für den eigentlichen Endanwender nicht gelten. Die Progressivität des Konzepts bedeutet für den Nutzer eine Umgewöhnung von der bekannten Funktionsweise und dem bisherigen Design. Ist es bei der AppStore-Version ersichtlich wie man zur nächsten Frage gelangt – sie steht unter der vorherigen – muss nun eine Geste angewendet werden, die nicht sofort zu erraten ist.
Nach kurzer Zeit haben die wenigen Testpersonen die das Privileg hatten die App auszuprobieren herausgefunden, wie sie Eingaben tätigen und zum nächsten Screen gelangen. Um dem Benutzer diese Zeit zu sparen wäre ein Bildschirm zur Erläuterung der Gesten beim erstmaligen Öffnen der App sinnvoll. Diese Herangehensweise kennt man von anderen Apps.

Schlussendlich sollte die App und die Art der Eingabekonzepte am Tinnitus-Patienten getestet werden. Für den Entwickler ist es von Interesse, welche neuentwickelte Eingabe am meisten Anklang findet und das größte Nutzererlebnis hervorruft. Bewährt sich das Zusammenspiel der Gesten? Gibt es Veränderungen bezüglich der Antworten auf die selben Fragen im Fragebogen, wenn man die neue und die alte Version heranzieht? Dies sind nur ein paar Fragen des Entwicklers, die sich in der realen Nutzung der Anwendung erschließen lassen.

# Literatur


[1] Digital Commerce 360. *Mobile Shopping trends in 2018 for Amazon*. 2019. URL: https://www.digitalcommerce360.com/2019/03/25/analyzing-amazons-mobile-users-in-2018/.

[2] JMango 360. *Mobile App versus Mobile Website Statistics*. 2019. URL: https://jmango360.com/wiki-pages-trends/mobile-app-vs-mobile-website-statistics/.

[3] Patrice Chalin, Hickson Ian und Zakhour Shams. *Inside Flutter*. 2019. URL: https://flutter.dev/docs/resources/inside-flutter.

[4] Patrice Chalin u. a. *Technical Overview*. 2019. URL: https://flutter.dev/docs/resources/technical-overview#layer-cakes-are-delicious.

[5] Emily Fortuna. *Keys! What Are They Good For?* 2019. URL: https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d.

[6] Marc Hassenzahl, Andreas Beu und Michael Burmester. „Engineering Joy“. In: *IEEE Software* (2001).

[7] Jochen Herrmann. „Konzeption und technische Realisierung eines mobilen Frameworks zur Unterstützung tinnitusgeschädigter Patienten“. Magisterarb. Universität Ulm, 2014.

[8] Tinnitus Research Initiative. *Track Your Tinnitus*. 2018. URL: https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/trackyourtinnitus/.

[9] Steve Krug. *Don't Make Me Think*. Second. Heidelberg: Redline GmbH, 2006.

[10] Stefan Luber. *Was ist eine IDE?* 2017. URL: https://www.dev-insider.de/was-ist-eine-ide-a-600703/.

[11] Thomas de Moor. *React Native vs Flutter: Which One Is Better?* 2019. URL: https://x-team.com/blog/react-native-vs-flutter/.

[12] Jakob Nielsen. *Usability 101: Introduction to Usability*. Website. https://www.nngroup.com/articles/usability-101-introduction-to-usability/. 2012.

[13] Chalin Patrice, Lin James D. und Zakhour Shams. *Rendering in Flutter*. 2019. URL: https://flutter.dev/docs/resources/rendering#render_boxdart.

[14] Helen Petrie und Nigel Bevan. „The Evaluation of Accessibility, Usability and User Experience“. In: *The Universal Access Handbook* (2009).

[15] Thomas Probst u. a. „Outpatient Tinnitus Clinic, Self-Help Web Platform, or Mobile Application to Recruit Tinnitus Study Samples?“ In: *Frontiers in Aging Neuroscience* 9 (2017), S. 113–113. URL: http://dbis.eprints.uni-ulm.de/1533/.

[16] Rüdiger Pryss u. a. „Mobile Crowdsensing Services for Tinnitus Assessment and Patient Feedback“. In: *6th IEEE International Conference on AI & Mobile Services (IEEE AIMS 2017)*. IEEE Computer Society Press, 2017. URL: http://dbis.eprints.uni-ulm.de/1521/.

[17] Rüdiger Pryss u. a. „Mobile Crowdsensing for the Juxtaposition of Realtime Assessments and Retrospective Reporting for Neuropsychiatric Symptoms“. In: *30th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2017)*. IEEE Computer Society Press, 2017. URL: http://dbis.eprints.uni-ulm.de/1517/.

[18] Rüdiger Pryss u. a. „Prospective Crowdsensing versus Retrospective Ratings of Tinnitus Variability and Tinnitus – Stress Associations Based on The TrackYourTinnitus Mobile Platform“. In: *International Journal of Data Science and Analytics* (2018). URL: http://dbis.eprints.uni-ulm.de/1654/.

[19] Adhithi Ravichandran. *React Native or Flutter — What Should I Pick To Build My Mobile App?* 2019. URL: https://medium.com/@adhithiravi/react-native-vs-flutter-what-are-the-differences-b6dc892f0d34.

[20] Daniel Saffer. *Usability Engineering nach DIN EN ISO 9241-11*. 2018. URL: https://medtech-ingenieur.de/usability-engineering-nach-din-en-iso-9241-11/.

[21] Axel Schröder. *Effizienz und Effektivität – was ist was? Definitionen & Tipps*. 2019. URL: https://axel-schroeder.de/effektivitaet-und-effizienz-was-ist-was-definitionen-tipps/.

[22] Frederik Schweiger. *The Layer Cake*. 2018. URL: https://medium.com/flutter-community/the-layer-cake-widgets-elements-renderobjects-7644c3142401.

[23] Vaishali Sonik. *iOS vs Android User Experience*. 2019. URL: https://www.apptunix.com/blog/ios-vs-android-user-experience/.

[24] Kyle Taylor und Laura Silver. *Smartphone Ownership Is Growing Rapidly Around the World, but Not Always Equally*. Techn. Ber. Pew Research Center, 2019.

[25] Flutter Team. *Hot Reload*. 2019. URL: https://flutter.dev/docs/development/tools/hot-reload.

[26] Flutter Team. *Introduction to Declarative UI*. 2019. URL: https://flutter.dev/docs/get-started/flutter-for/declarative.

[27] Flutter Team. *Introduction to widgets*. 2019. URL: https://flutter.dev/docs/development/ui/widgets-intro.

[28] Flutter Team. *Layouts in Flutter*. 2019. URL: https://flutter.dev/docs/development/ui/layout.

[29] Flutter Team. *State Management*. 2019. URL: https://flutter.dev/docs/development/data-and-backend/state-mgmt.

[30] Flutter Team. *Taps, Drags, and Other Gestures*. 2019. URL: https://flutter.dev/docs/development/ui/advanced/gestures.

[31] Fokus UX. *Usability & User Experience (UX)*. 2019. URL: https://fokus-ux.de/usability-user-experience.

[32] Wikipedia. *App Store (iOS)*. 2019. URL: https://de.wikipedia.org/wiki/App_Store_(iOS).

[33] Wikipedia. *Dart (langage)*. 2019. URL: https://fr.wikipedia.org/wiki/Dart_(langage).

[34] Yücel Yelken. *Mobiles Internet in Schwellen- und Entwicklungsländern*. 2013. URL: https://www.marktforschung.de/dossiers/themendossiers/mobile-research/dossier/mobiles-internet-in-schwellen-und-entwicklungslaendern/.

[35] https://fokus-ux.de/ux design. *UX-Design (User Experience Design)*. 2019. URL: https://fokus-ux.de/ux-design.

Name: Jost Jonas Matrikelnummer: 925652

**Erklärung**

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

Ulm, den 15.10.2019 J. Jonas

Jost Jonas