Technische-Hochschule Nürnberg

Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Studiengang: Master Elektronische und Mechatronische Systeme

Abschlussarbeit von

Richard Stern

Evaluierung eines Cross-Plattform-SDKs für mobile Anwendungen im Kontext von Multitouch-Sensoren und prototypische Implementierung eines app-basierten Frontends

Sommersemester 2019

Abgabedatum: 4. Juni 2019

Betreuer: Prof. Dr. Oliver Hofmann
Prof. Dr. Ralph Lano
Christian Braun (KURZ Digital Solutions)
Dipl.-Ing. Sean Durkin (PolyIC)

Schlagwörter: Flutter, Cross-Plattform, App-Entwicklung

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Stern Vorname: Richard Matrikel-Nr.: 2404898

Fakultät: Elektro-, Feinwerk-, Informationstechnik Studiengang: Elektronische und Mechatronische Systeme

Semester: Sommersemester 2019

Titel der Abschlussarbeit:

Evaluierung eines Cross-Plattform-SDKs für mobile Anwendungen im Kontext von Multitouch-Sensoren und prototypische Implementierung eines app-basierten Frontends

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☒ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,

☐ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Ort, Datum, Unterschrift Studierende/Studierender

Abstract

Das Thema dieser Arbeit ist die Entwicklung einer App mit dem Cross-Plattform Software Development Kit Flutter und die Einbindung in firmeninterne Entwicklungs-Prozesse bei KURZ Digital Solutions. Die zu entwickelnde App zeigt Berührungen auf einem Touch-Sensor darstellen, der um Bluetooth erweitert wird. Bei Flutter wird sowohl die Benutzeroberfläche als auch die Funktionalität in der Sprache Dart beschrieben. Flutter baut die Benutzeroberfläche aus Widgets auf, die lediglich den State der App visualisieren. Zur Verwaltung des States werden verschiedene Systeme vorgestellt und die Implementierung des für Flutter entwickelten BLoC-Patterns gezeigt, das den Datenfluss in der App mit Streams realisiert und den State in Geschäftslogiken ausgliedert. Mögliche Restrukturierungen werden gezeigt, die besonders in der Cross-Plattform-Entwicklung positive Effekte auf die Wartbarkeit und Testbarkeit haben können. Vom Konzept der App bis zum Test lässt sich Flutter gut in bestehende Abläufe der App-Entwicklung einbinden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

1 Einleitung

Die Entwicklung von Apps ist zeitaufwändig, erfordert Vorwissen und ist teuer. Auf dem Markt der mobilen Betriebssysteme herrscht ein Duopol aus Android und iOS [7]. Um eine App für annähernd den gesamten Markt anbieten zu können, müssen dementsprechend zwei Apps separat entwickelt werden. Diese entstehen meist durch unterschiedliche Entwicklungs-Teams oder Firmen. Die Anforderungen, Designentscheidungen und Änderungen müssen mehreren Personen mitgeteilt werden und in den entsprechenden Code eingepflegt werden. Die Dokumentation erfolgt deshalb auch oft doppelt. Die Idee, nur noch eine Code-Basis, statt zwei, mit einem Entwicklungsteam pflegen zu müssen, ist hierbei naheliegend. Betriebssystemtypische Gegebenheiten sollten leicht implementierbar sein und die Performance dem entsprechen, was ein Nutzer der jeweiligen Plattform, gewohnt ist.

Der Schwerpunkt dieser Arbeit liegt auf der Cross-Plattform-Entwicklung einer App mit dem Software Development Kit (SDK) *Flutter*. Mit Flutter kann eine Applikation in nur einer Programmiersprache, nämlich Dart, geschrieben werden, die auf verschiedenen Plattformen lauffähig ist. Diese App wird für die Firma PolyIC GmbH & Co. KG entwickelt und soll eine Windows-Software ersetzen. Die Entwicklung einer Flutter-App erfolgt bei der Firma KURZ DIGITAL Solutions GmbH & Co. KG. Betrachtet werden die Einbindung in bestehenden Entwicklungsabläufe, sowie verschiedenen Möglichkeiten, eine Flutter-App zu konstruieren.

Die Programmiersprache Dart, in der Flutter geschrieben ist, ist eine deklarative Programmiersprache, bei der die UI (engl. Benutzeroberfläche) einer App im Code selbst beschrieben wird [8]. Eine *just in time* (JIT) Kompilierung soll es ermöglichen, bei der Softwareentwicklung schneller iterieren zu können. Flutter soll durch eine optimierte Render-Engine eine hohe Performance aufweisen. Der Google Developer Blog berichtet darüber hinaus von einem stark ansteigenden Interesse an Flutter [5]. Dies und die Unterstützung von Google sind die motivierenden Gründe Flutter im praxisnahen Kontext zu betrachten.

1.1 Problemumfeld

Die Firma PolyIC entwickelt gedruckte Elektronik auf Basis von auf Polyethylenterephthalat-Folien (PET) gedruckten Metal-Mesh-Strukturen. Die Firma KURZ DIGITAL Solutions GmbH & Co. KG entwickelt größtenteils Plattform-Lösungen im Bereich Mobile und Web.

Im Rahmen eines Forschungsprojektes ist bei der Firma PolyIC GmbH & Co. KG ein multitouch-fähiger, kapazitiver Touch-Sensor entstanden. Zur Demonstration der Funktionsweise des Sensors (Kapitel 2.2) wurde eine Windows-Software entwickelt, welche erkannte Berührungen auf dem Sensor auswertet und grafisch darstellt. Der Sensor ist in einen physikalischen Demonstrator eingebaut. Der Demonstrator wird zur Präsentation des Sensors bei Messen, oder (potentiellen) Kunden eingesetzt. Dabei sollen die technischen Möglichkeiten und Einsatzgebiete des Sensors gezeigt werden. Anschließend wird mit dem Kunden ein optimiertes Produkt geplant und hergestellt.

Der bestehende Demonstrator wird per USB-Leitung an einem Windows PC angeschlossen, auf welchem die Software und Treiber installiert sind. Bei Außeneinsätzen ist die Benutzung von Laptops (gewisser Marken) firmenpolitisch eingeschränkt, weswegen die Software auf verschiedenen Computern aufgespielt werden müsste. Eine Software-Variante, die mit dem Smartphone kompatibel ist, bietet daher eine höhere Flexibilität und Usability, da ein Demonstrator (mit einem zusätzlichen Akku) ohne externe Leitungen zur Strom- oder Datenversorgung möglich ist. Die Windows-Software soll nicht weiter entwickelt werden und durch eine App abgelöst werden. Eine gleichzeitige Entwicklung für die beiden Smartphone-Betriebssysteme ermöglicht es, zusätzlich zu den vermehrt in der KURZ-Unternehmensgruppe vorkommenden iOS-Geräten, die Kompatibilität der App für Android zu erweitern.

1.2 Zielsetzung

Im Rahmen dieser Arbeit soll das Cross-Plattform SDK Flutter zur App-Entwicklung eingesetzt werden. Am zuvor genannten Anwendungsbeispiel soll analysiert werden, ob und in welchen Situationen sich eine Cross-Plattform-Entwicklung in einem professionellen Umfeld eignet. Kann problemlos auf die Smartphone-Hardware, insbesondere Bluetooth zugegriffen werden und lässt sich Flutter in den bestehenden Workflow der App-Entwicklung von KURZ DIGITAL Solutions GmbH einpflegen?

1.3 Vorgehensweise

Zuerst werden die Eignungskriterien für das SDK und die Anwendung festgelegt. Diese umfassen sowohl die Wünsche des Benutzers der App sowie die Entwicklungsabläufe während der Programmierung. Tests, Fehlerbehebung, Erweiterbarkeit und die Integration in firmeninterne Arbeitsabläufe sind ebenfalls von Relevanz. Anschließend soll für den exemplarischen Anwendungsfall, für die Firma PolyIC, die App konzipiert und entwickelt werden. Dabei muss auf plattformspezifische Gegebenheiten und Usability Erfordernisse geachtet werden [9, 10]. Hierbei sollen die Schnittstellen zur Anbindung der Smartphone-Hardware geprüft werden. Mit den anfangs erarbeiteten Kriterien zur Überprüfung der Eignung von Flutter soll nun geprüft werden, wie gut sich Flutter in ein professionellen Umfeld

einfügen lässt. Betrachtet wird auch, welche Konzepte von Flutter den App-Entwicklungsablauf im Vergleich zu iOS und Android verändern oder was beachtet werden muss.

Diese Arbeit beschäftigt sich zunächst in Kapitel 2 mit den Hard- und Software Komponenten, die zu Beginn der Arbeit bereits vorhanden waren. Dies sind zum einen der Sensor mit daran angeschlossenen Touch- und Mikrocontroller und eine PC-Software, die mit der Hardware kommuniziert und Toucheingaben auf dem Sensor visuell darstellt. Im Kapitel 3 wird die vom SDK Flutter verwendete Programmiersprache Dart vorgestellt, gefolgt von Flutter selbst und Bluetooth zur Datenübermittlung. Anschließend wird in Kapitel 4 das Konzept der App vorgestellt und in Kapitel 5 auf die Implementierung von Bluetooth bei Hard- und Software, sowie der Programmierung der App und deren Testmöglichkeiten eingegangen.

2 Kapitel 2
Ausgangssituation

In diesem Kapitel wird darauf eingegangen, welche Hard- und Software Bestandteile dieser Arbeit zugrunde liegen. Zuerst wird auf die aktuelle Architektur in 2.1 eingegangen. Daraufhin wird der Sensor in Kapitel 2.2 vorgestellt, gefolgt vom Touch-Controller und Mikrocontroller in Kapitel 2.3 und der PC-Software, die zur Visualisierung benutzt wurde, in Kapitel 2.4.

2.1 Ursprüngliche Architektur des Demonstrators

Der Hardware-Aufbau besteht aus einem Touch-Sensor, der zu Auswertung an mehrere Platinen angeschlossen ist. Diese Elektronik besteht aus einem Touch-Controller und einem Mikrocontroller. Der Touch-Controller ist mit einer Adapterplatine mit dem Sensor verbunden und wertet die Signale von diesem aus. Dabei wird die Berührung eines Fingers auf dem Sensors als *Finger Down*, eine Bewegung des Fingers auf dem Sensor als *Finger Move* und ein Anheben des Fingers als *Finger Up* definiert. Diese Informationen werden vom Mikrocontroller zusätzlich zu der X- und Y-Koordinate per *UART* an das *FTDI-Modul* geschickt. Das FTDI-Modul wandelt die Daten vom Mikrocontroller, um diese per USB zu übertragen. Der angeschlossene Computer wertet die Daten mit der Windows-Software aus und zeigt die Berührungen an. Die einzelnen Komponenten werden in Abbildung 2.1 dargestellt und nachfolgend vorgestellt.

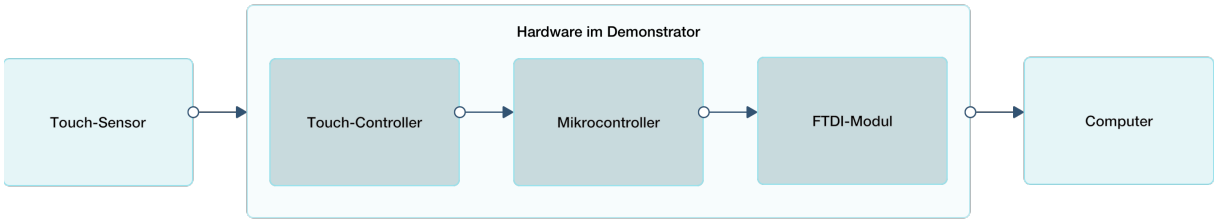

Abb. 2.1: Ursprüngliche Architektur des Demonstrators

In Abbildung 2.2 ist der Demonstrator dargestellt. Für den Nutzer ist lediglich der Touch-Sensor sichtbar. Die Optik und Haptik ähnelt dem Interieur eines Automobils, um einen gängige Anwendungsort optisch zu imitieren.

Abb. 2.2: Der Touch-Sensor in einem Demonstrator verbaut. Quelle: PolyIC [1].

2.2 Touch-Sensor von PolyIC

Ein Touch-Sensor erkennt Berührungen auf der Oberfläche und ermöglicht die Erkennung der Koordinaten der Berührungen. Der verwendete Sensor wird von PolyIC entwickelt und in Partnerschaft mit KURZ hergestellt. Zum Sensor selbst existieren zum jetzigen Zeitpunkt keine offiziellen Datenblätter, sondern lediglich Werbematerial. Der Sensor ist vom Typ *PolyTC* und besteht aus dünnen Metallschichten (Metal-Mesh) und einem Trägermaterial (Kunststofffolie aus PET) [11].

Die Firma PolyIC vermarktet unter dem Produktnamen *PolyTC* transparente Folien, mit der Oberflächen mit kapazitiver Berührungsempflichkeit ausgestattet werden können. Die Folien können mit einer hohen Lichtdurchlässigkeit von über 85 % hergestellt werden, um diese auch auf einem Display einsetzen zu können. PolyTC kann kostengünstig hergestellt werden. Durch eine große Flexibilität kann der Sensor leicht auf gekrümmten Kunststoffteilen angebracht werden, um möglichst die gesamte Fläche berührungsempfindlich zu machen. Die Empfindlichkeit der hergestellten Sensoren ist mit anderen kommerziell verfügbaren Touchscreen Sensoren vergleichbar. Der Sensor besitzt diamantförmige Sensorfelder.

Die Herstellung des Sensors und die weitere Verarbeitung kann entweder von PolyIC, KURZ, oder anderen Firmen der KURZ-Gruppe übernommen werden. Der Sensor kann im *Functional Foil Bonding* (FFB) Verfahren mit hohem Druck und Hitze auf Kunststoff aufgebracht werden, oder durch *In Mold Labeling* (IML) in einer Spritzgussform mit Kunststoff hinterspritzt werden. In Abbildung 2.3 ist der Sensor ohne Kunststoff-Dekorteil dargestellt.

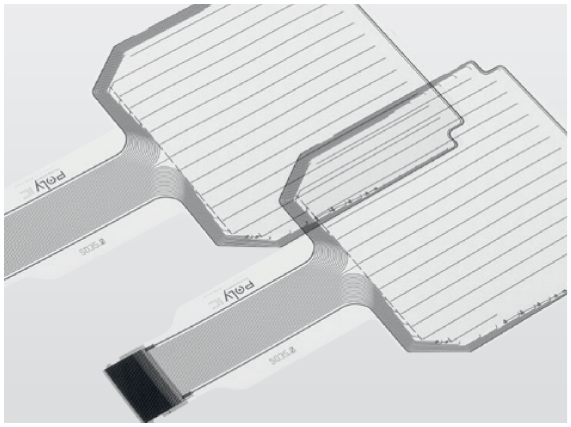

Abb. 2.3: Touch-Sensor von PolyIC ohne ein Dekorteil.

2.3 Touch-Controller und Mikrocontroller

Der Sensor ist über einen Adapter mit einem Touch-Controller verbunden, der die Auswertung der Berührungen auf dem Sensor übernimmt. Der Touch-Controller unterstützt *Multitouch*, also die Erkennung von mehreren Fingern auf dem Sensor. Die Koordinaten der Berührungen können den jeweiligen Fingern zugeordnet werden. Durch diese logische Zuordnung kann die Bewegung von Fingern verfolgt werden. Der Touch-Controller ermöglicht darüber hinaus die Gestenerkennung. Bewegungsmuster der Finger auf dem Sensor können erkannt werden, wie z.B. eine Wischgeste, die meist in X- oder Y-Richtung ausgeführt wird, um z.B. durch eine Liste oder ein Menü zu navigieren. Diese beiden Erkennungstypen (Gestenerkennung und Multitouch) können kombiniert werden, um Gesten mit mehreren Fingern zu erkennen (z.B. eine Geste, bei der mehrere Finger in Y-Richtung auf dem Sensor bewegt werden). Der Mikrocontroller leitet die Messwerte an den Computer weiter.

2.4 PC-Software zum Demonstrator

Die PC-Software wird benötigt, um einen Windows Computer mit dem Demonstrator zu verbinden und dem Benutzer eine Benutzeroberfläche anzuzeigen. Zentrales Element der Software ist ein Abbild des Sensors. An den Stellen, an denen Berührungen detektiert werden sind farbliche Kreise abgebildet. Jedem Finger, der von der Hardware detektiert wird, ist in der Software eine Farbe zugewiesen, die sich nicht ändert, solange der Finger nicht abgehoben wird. Am rechten Rand des Sensors wird ein Schieberegler dargestellt, wenn in diesem Bereich eine Fingerberührung erkannt wird. Der Schieberegler ändert einen Zahlenwert, der daneben dargestellt wird. Auf der linken Seite des Sensors befinden sich drei Schaltflächen, welche bei Betätigung in der Software farblich hervorgehoben werden.

Kapitel 3

3 Grundlagen

In diesem Kapitel werden Grundlagen beschrieben, die relevant für das Verständnis dieser Arbeit sind. Dabei leitet die Skriptsprache Dart (Kapitel 3.1) das Grundlagen-Kapitel ein, gefolgt vom SDK Flutter, das Dart verwendet (Kapitel 3.2). Zum Schluss wird der Funkstandard Bluetooth und die energiesparende Version Bluetooth Low Energy (Kapitel 3.3) gezeigt.

3.1 Die Programmiersprache Dart

Dart ist eine klassenbasierte, Single Inheritance und rein objektorientierte Sprache und wurde von Google entwickelt [12]. Die erste Version erschien im Oktober 2011. Die Veröffentlichung der stabilen Version 1.0 erfolgte im November 2013. Die Standardisierung von Dart erfolgte durch die Ecma International (Technical Committee TC52) in der ersten Version im Jahr 2014 [13]. Der aktuelle Stand der Entwicklung ist die stabile Version 2.3 (Mai 2019).

Mit Dart können komplexe Web Applikationen und Apps entwickelt werden [14]. Dart Code kann mit dem Dart-to-JavaScript Transcompiler *Dart2js* zu JavaScript kompiliert werden. Ist der Desktop das Entwicklungsziel, so empfiehlt Google das Framework *AngularDart* einzusetzen – für mobile Anwendungen Flutter [8]. Der Laufzeit-Typ eines jeden Objekts ist vom Typ der Klasse `Type` und kann mit einem Getter abgerufen werden. Dieser Getter `runtimeType` ist in der Klasse **`Object`** deklariert. Jede Dart Klasse ist eine Subklasse von **`Object`**.

Dart kann, muss jedoch nicht, statisch typisiert programmiert werden [15]. Offiziell werden Dart Plugins für Android Studio, Visual Studio Code und IntelliJ IDEA (und andere JetBrains IDEs) unterstützt [16]. Das Dart SDK ermöglicht es, Dart Code direkt in der Kommandozeile auszuführen. Dies funktioniert, da die Dart Virtual Machine (VM) ein Bestandteil des Dart SDKs ist. Der Aufruf einer Dart Datei erfolgt hierbei mit dem Befehl `dart`, gefolgt von der Datei.

In der Entwicklung kann Dart Code JIT kompiliert und in einer Dart-VM ausgeführt werden. Die JIT-Kompilierung kann Zeit einsparen, da Änderungen am Source Code schneller übersetzt werden. Bei der Entwicklung einer Flutter-App mit Dart wird diese Eigenschaft intensiv genutzt und mit der Bezeichnung *Hot Reload*, beziehungsweise *Hot Restart* von Google versehen, die im Kapitel 3.2.4 beschrieben ist. Bei der JIT-Kompilierung besitzt die Anwendung jedoch eine verringerte Performance. Um diese steigern zu können, kann anstatt der JIT-Kompilierung die Anwendung auch *ahead of time* (AOT) kompiliert werden.

Sobald die Entwicklung einer mit Dart geschriebenen Anwendung abgeschlossen ist und die Software produktiv eingesetzt werden soll, wird der Code AOT kompiliert. Viele Debugging-Optionen stehen in diesem Modus nicht mehr bereit, weshalb dieser Modus zur Entwicklung nur bedingt sinnvoll ist. Dart Code kann in Maschinencode (32 oder 64 Bit ARM Code) umgewandelt werden. Eine mit Flutter entwickelte App wird mit AOT kompiliertem Dart Code in die App Stores von Android (Google Play Store) und iOS (Apple AppStore) eingestellt.

3.1.1 Im SDK enthaltene Tools

Eine Dart Applikation basiert auf *packages*, die dieses mit Funktionalität erweitern. Zur Verwaltung von Dart *packages* kann der *Package Manager Pub* benutzt werden [17, 18]. *Pub* ist für das Dependency Management zuständig. Es können Dart *libraries* oder *packages* installiert, aktualisiert oder verteilt werden. Der Aufruf erfolgt per IDE oder Befehl. *Packages* können sich an verschiedenen Orten, wie beispielsweise auf *GitHub*, befinden.

Dabei muss *Pub* ein Name und eine Versionsnummer übergeben werden, um auf einen bestimmten, nicht veränderbaren, Code zu verweisen. Wird eine Abhängigkeit aktualisiert, so wird die Versionsnummer aktualisiert. Dart *packages* können gemeinsame Abhängigkeit haben, jedoch mit verschiedenen geforderten Versionen. Daher bietet Dart die Möglichkeit an, einen Bereich zu definieren, in dem sich die Version befinden kann. Der Bereichs kann beispielsweise mit `'>=1.3.0 <1.5.0'` zwischen 1.3.0 bis ausschließlich 1.5.0 festgelegt werden. Ein Bereich der Kompatibilität kann auch mit der Caret-Syntax angegeben werden. Dabei wird das Symbol `^`, gefolgt von der Versionsnummer angegeben. Jede Version, bis zu einem größeren Versionssprung ist kompatibel. Beispielsweise bei `^1.3` ist jede Version von 1.3 bis ausschließlich 2.0 kompatibel. *Pub* kontrolliert die verfügbaren Versionsnummern und wählt die höchste, kompatible Version aus. Werden zwei verschiedene, miteinander nicht kompatible Versionen eines *packages* erwartet, so wird ein Fehler ausgegeben. Um einen *version* lock zu vermeiden kann eine Bereichsangabe verwendet werden.

Ein Static Analyzer ist eine Methode, Code zu debuggen bevor das Programm gestartet wird. Diese Art von Analyse wird im gesamten Entwicklungsprozess angewendet. Mithilfe des Static Analyzer `dartanalyzer` werden Fehler und Warnungen angezeigt. Warnungen sind hierbei Hinweise darauf, dass der Code eventuell nicht funktioniert, das Programm jedoch trotzdem ausgeführt werden kann. Bei Fehlern wird in *compile-time* und *run-time* Fehler unterschieden. Bei einem Fehler zur Kompilierzeit wird der Code nicht ausgeführt - bei einem Laufzeit-Fehler wird bei Dart eine Ausnahmebehandlung (engl. *exception*) ausgelöst, während der Code weiterhin ausgeführt wird.

Ist der Code unformatiert, oder weist falsche Einrückungen auf, ist dieser schwer lesbar, denn Dart verwendet lexikalische Gültigkeitsbereiche. Die Einrückung der Klammern verdeutlicht diese Bereiche optisch. In Listing 3.1 kann innerhalb der geschweiften Klammern (der Code, der nach rechts eingerückt ist) auf die Variable `out` zugegriffen werden. Wenn auf die Variable **`in`** außerhalb dieser Klammern zugegriffen wird, wird ein Fehler generiert.

```
main(){
  var out = 'outside';
  {
    var in = 'inside';
    print(out); // Print: 'outside'
  }
  in = "tryingToWrite"; // ERROR!
}
```

Listing 3.1: Darstellung von lexikalischen Gültigkeitsbereichen bei Dart.

Der Code Formatierer mit `dartfmt` formatiert formatiert den Code gemäß den Empfehlungen aus dem *Dart Style Sheet* [19]. Es ist nicht zwingend notwendig, dass ein Entwickler sich an diesen Empfehlungen orientiert, es wird von Google jedoch empfehlenswert. Der Code Formatierer soll die Optik und damit die Lesbarkeit des Codes für den Programmierer verbessern. Die Funktionalität des Codes wird nicht beeinflusst. Unterstützt die IDE Dart, so kann der Code in der IDE per Knopfdruck, oder Shortcut, formatiert werden. In Listing 3.2 ist ein unformatierter Code dargestellt, der mit `dartfmt` formatiert wird. Das Resultat der Optimierung ist in Listing 3.3 zu sehen.

```
children: <Widget>[
  WidgetX, Row(children: <Widget>[
    WidgetY,WidgetZ,],),],
```

Listing 3.2: Code vor der Formatierung mit dartfmt.

```
children: <Widget>[
  WidgetX,
  Row(
    children: <Widget>[
      WidgetY,
      WidgetZ,
    ],
  ),
],
```

Listing 3.3: Code nach der Formatierung mit dartfmt.

3.1.2 Code Dokumentation

Ein Kommentar beginnt mit `//` und endet mit dem Ende der Zeile. Blockkommentare werden in Dart dazu verwendet werden, Code temporär zu kommentieren, beginnt mit `/*` und endet mit `*/`. Um Zusammenhänge zwischen Dateien zu erkennen, ist es oft sinnvoll einen Gesamtüberblick über viele Dateien zu erhalten, ohne den Code selbst sichten zu müssen. Dabei bietet Dart mit der Syntax `///` die Möglichkeit Funktionen, Methoden und Klassen zu beschreiben und daraus eine Dokumentation generieren zu lassen. Der nach `///` folgende Text wird beim Ausführen von `dartdoc` als Kommentar für die HTML-Dokumentation hergenommen. Der erste Satz, der auf `///` folgt, beschreibt die Logik der Funktion kompakt und möglichst vollständig. Ergänzende Informationen sollen, wenn notwendig, nach einer Leerzeile folgen. In Listing 3.4 ist eine Funktion so beschrieben, dass diese von `dartdoc` ausgewertet werden kann.

```
1 /// Deletes file at [location] in database.
2 ///
3 /// More info regarding this function is written
4 /// down here
5 void deleteFile(String location) {
6   // Function body
7 }
```

Listing 3.4: Dart Kommentare mit dartdoc zur Dokumentation des Codes hinzufügen.

3.1.3 Variablen

Eine Variable enthält eine Referenz auf ein Objekt. Der Name einer Variablen wird in Dart klein geschrieben. Dart unterstützt das Konzept der Typinferenz (Typableitung), bei dem der Typ einer Variablen, die mit **`var`** initialisiert wird, bei der ersten Zuweisung angenommen wird [12]. In Listing 3.5 ist die Initialisierung zweier Variablen dargestellt. Einmal geschieht dies unmittelbar (`surname`) und das andere Mal erst im späteren Programmablauf (`name`). Bis `name` initialisiert ist, ist der Wert dieser Variablen **`null`**. Beiden Variablen wird eine Referenz auf ein **`String`** Objekt zugewiesen und der Typ **`String`** angenommen. Die Benutzung von **`var`** bei lokalen Variablen soll die Lesbarkeit des Codes erhöhen, da die Aufmerksamkeit des Betrachters auf dem Namen und Wert der Variablen und nicht auf dem Typ liegen soll [20].

Typinferenz kann eine potentielle Fehlerquelle sein, besonders wenn der Typ nicht durch die erste Zuweisung richtig beschrieben werden kann. Ist beispielsweise der gewünschte Typ einer Variable **`num`**, aber die Variable eine **`var`**, so entscheidet die erste Zuweisung über den Typ der Variablen. Wird zuerst ein **`int`** zugewiesen und danach versucht ein **`double`** zuzuweisen, so wird ein Fehler generiert.

```
1 var name, surname = 'SURNAME';
2 name = 'NAME';
```

Listing 3.5: Initialisierungsmöglichkeiten von Variablen bei Dart.

Die erste Zuweisung entscheidet über den Typ (in diesem Fall **int**). Bei **var** ist zur Kompilier- und Laufzeit der Typ bekannt. Es ist daher egal, ob **var** s = *"text"* oder **String** s = *"text"* geschrieben wird.

Eine Variable kann den Typ während der Laufzeit ändern, wenn diese den Typ **dynamic** benutzt. Zur Kompilierzeit kann bei dem Typ **dynamic** nicht überprüft werden, ob Eigenschaften, Methoden, Operatoren, etc. des eingesetzten Typs vorhanden sind oder nicht. Nur zur Laufzeit kann dies überprüft und ein Fehler generiert werden.

Alle Dart Variablen, auch Zahlen, haben den Initialwert **null**, solange keine Zuweisung erfolgt ist. Soll die Variable unveränderlich sein, so kann **var** durch **const**, oder **final** ersetzt werden. Mit **const** wird ein Objekt bereits zur Kompilierzeit als unveränderlich definiert. Soll einem Objekt zur Laufzeit ein Wert zugewiesen werden, der zur Kompilierzeit noch nicht da ist, so wird **final** verwendet [21]. Mit **static** kann eine Variable einer Klasse und nicht nur der jeweiligen Instanz der Klasse zur Verfügung gestellt werden.

Ein **const** Objekt kann nur aus Daten und Objekten definiert werden, die zur Kompilierzeit zur Verfügung stehen und ebenfalls **const** sind. Eine **const** Variable ist implizit **final** und kann zur Laufzeit nicht mehr verändert werden.

Ein **final** Objekt muss zwingend initialisiert werden. Dies geschieht sobald diese Variable das erste Mal verwendet wird, beispielsweise beim Aufruf einer Klasse oder Funktion. Anschließend darf sich dieses Objekt nicht mehr ändern. Ist eine **final** Variable eine Klassenvariable, so muss die Zuweisung direkt bei der Deklaration erfolgen. Wird eine Benutzeroberfläche gebaut, wie bei Flutter beispielsweise, kann diese zur Laufzeit initialisiert werden und ab diesem Zeitpunkt statisch sein. Werden viele Datenelemente, aus denen die UI besteht, als **final** oder sogar **const** deklariert, kann viel Rechenleistung gespart werden, da definiert ist, welche Teile der UI unter welchen Umständen gerendert werden müssen. Das `StatelessWidget` (siehe Kapitel 3.2.2) wird ausschließlich mit **final** Variablen initialisiert. Bei der Variablen Initialisierung kann die Type Annotation hinzugefügt werden (z.B. **String** s = *'text'*).

Um mehrere Variablen zu gruppieren, kann `List` oder `Map` verwendet werden. Eine `List` ist eine Sammlung von Objekten. Die Länge von der Anzahl an Objekte ab, welche enthalten sind [22, 23]. Die Länge kann fix oder variabel sein. Eine `List` kann nicht nur einzelne Objekte, sondern auch weitere Listen enthalten. Der Typ einer Liste kann, wie bei einer Variablen, angegeben werden, oder durch Typinferenz erfolgen.

Eine `Map` ist ein Objekt, das aus Schlüsseln (`key`) und den dazugehörigen Werten (`value`) besteht [24, 22]. Sowohl der Wert, als auch der Schlüssel kann von einem beliebigen Typ sein, da Dart ein `Map` Objekt vom Typ `Map<dynamic, dynamic>` erstellt. Bei der ersten Zuweisung wird die Typinferenz verwendet. Pro `Map` kann ein `key` nur einmal und ein `value` mehrmals existieren.

3.1.4 String Interpolation

Ein Dart **`String`** besteht aus UTF-16 Einheiten und beginnt und endet mit einem Apostroph, oder alternativ mit Anführungszeichen oben [22]. In einem String kann auch direkt der Inhalt von Variablen eingebunden werden, was bei Flutter besonders oft benutzt wird, da so Informationen in einem Text eingebettet werden kann. Dies geschieht durch das Symbol $, gefolgt vom Namen der Variablen, deren Inhalt eingebunden werden soll. In Listing 3.6 wird der **`String`** `s3` in den**`String`** `s4` eingefügt.

```
1 String s3 = 'Interpolation';
2 String s4 = "String with $s3";
3 print(s4); // String with Interpolation
```

Listing 3.6: String Interpolation mit einem weiteren String.

In einem String kann jedoch auch auf Methoden oder Funktionen zugegriffen, oder logische Ausdrücke ausgewertet werden. Der Ausdruck wird nach dem Symbol $ in geschweiften Klammern geschrieben. Im Listing 3.7 besitzt der **`String`** `s5` zur Laufzeit den Inhalt *`"Check: (int number == 5): true"`*.

```
1 int number = 5;
2 String s5 = "Check: (int number == 5): ${number == 5}";
```

Listing 3.7: String Interpolation mit einem Ausdruck.

3.1.5 Klassen

Eine Klasse ist ein abstraktes Modell, das zur Erzeugung von ähnlichen Objekten dient. Der Name einer Klasse wird in Dart groß geschrieben. Wird eine neue Instanz einer Klasse erstellt, so wird der Klassenkonstruktor aufgerufen. In Dart besitzt eine Klasse mindestens einen Konstruktor. Wenn kein expliziter Konstruktor definiert ist, wird der Default-Konstruktor aufgerufen. Der Default-Konstruktor besitzt keine Argumente.

Ein Konstruktor teilt seinen Namen mit der Instanz einer Klasse. Ein Konstruktor kann parametrisiert werden, um Variablen zu initialisieren. In Listing 3.8 wird der String *name* initialisiert. Sobald

es zu Namens-Konflikten kommen kann, ist die Verwendung von **this** nötig. Ansonsten kann es weggelassen werden.

```
class Person {
  String name;
  Person(String name) {
    this.name = name;
  }
}
```

Listing 3.8: Aufbau einer Dart Klasse mit Konstruktor.

Alternativ kann in Dart dies direkt in einer Zeile, wie in Listing 3.9, mit **this**.name, erfolgen. Eine Mischung der Initialisierungsarten der Parameter ist möglich.

```
class Person {
  String name;
  String surname;
  Person(this.name, surname){
    this.surname = surname.toUpperCase(); // Surname in uppercase letters
  };
}
```

Listing 3.9: Vereinfachte Beschreibung eines Konstruktors in Dart.

Jede Instanzvariable generiert eine Getter-Methode. Instanzvariablen, die nicht final sind, erzeugen eine Setter-Methode [22]. Auf die Variable `name` in der Klasse `Person` (Listing 3.8) kann nach Initialisierung der Klasse per Setter-Methode der Wert der Variablen verändert werden (siehe Listing 3.10).

```
void main(){
  var person1 = Person();
  person1.name = "sampleName";
}
```

Listing 3.10: Zugriff auf Instanzvariablen einer Dart Klasse mit Setter.

Es können mehrere Konstruktoren definiert werden. Diese unterscheiden sich im Namen. Bei Java unterscheiden sich die Konstruktoren lediglich durch die Verwendung unterschiedlicher Parameter

und werden *parametrisierter Konstruktor*, oder auch *allgemeiner Konstruktor* genannt [25]. Bei Dart werden *named*-Konstruktoren aufgerufen [22]. Diese beginnen mit dem Namen der Klasse gefolgt von einem Punkt und dem Namen des Konstruktors mit den Parametern in runden Klammern. Eine Klasse kann beliebig viele Konstruktoren besitzen. Die Klasse `Person` aus Listing 3.9 kann zusätzlich zum Standardkonstruktor `Person()` z.B. den named-Konstruktor `Person.named1()` besitzen.

Mit **`extends`** kann eine **`super`** Klasse erweitert werden und eine Subklasse erstellt werden. Dabei können weitere Eigenschaften hinzugefügt werden oder Methoden der **`super`** Klasse überschrieben werden. In der Subklasse kann auch auf die **`super`** Klasse zugegriffen werden. Werden Methoden überschrieben, sind die Parameter vom selben Typ, oder Supertyp, wie die Parameter der **`super`** Klasse.

3.1.6 Dart Type System

Dart ist *type safe*, da statische Type Checks und Runtime Checks verwendet werden, um sicherzustellen, dass der Wert einer Variablen dem statischen Typ der Variablen entspricht. *Type safe* wird, von Google, auch *sound Dart* genannt [12]. Statische Typüberprüfungen ermöglichen es, Bugs bereits zur Kompilierzeit zu finden. Hierfür wird Dart's Static Analyzer verwendet [26]. Wird beispielsweise eine Liste ohne Cast erstellt und einer Funktion übergeben, die einen expliziten Typ erwartet, wird dies als Fehler erkannt. Die Funktion `printList()` in Listing 3.11 erwartet eine Liste vom Typ **`int`**. Die Liste `list1` kann ohne Fehler an die Funktion übergeben werden. Die Liste `list2` kann jedoch außer **`int`** auch Elemente anderen Typs enthalten und ist daher vom Typ **`dynamic`**. Bei Übergabe der Liste an die Funktion wird ein TypeError ausgelöst.

```
1  void printList(List<int> listToPrint) => print(listToPrint);
2
3  void main() {
4    var list0 = {1, 2, 3};
5    var list1 = <int>[];
6    var list2 = [];
7
8    list1.addAll(list0); // Appends all objects of iterable list0 to the end of list1
9    printList(list1); // Output: [1, 2, 3]
10
11   list2.add("Can also contain Strings");
12   list2.addAll(list0);
13   print(list2); // Output: [Can also contain Strings, 1, 2, 3]
14   list1.addAll(listInt); // Error: Uncaught exception: TypeError
15 }
```

Listing 3.11: Erzeugung eines Typ Fehlers, durch weglassen eines nötigen Typ Casts.

Die Eigenschaft eines Dart Programms, es zu vermeiden, in einen unzulässigen Zustand zu geraten wird von Google *Soundness* genannt [12]. Dadurch können Bugs frühzeitig erkannt und beseitigt werden und AOT kompilierter Code wird effizienter.

Ab der Version 2.1 unterstützt Dart die Konvertierung von **`int`** zu **`double`**. Dadurch kann ein Integer Wert an eine Funktion oder Klasse übergeben werden, die ein **`double`** erwartet. Voraussetzung ist, dass der Wert exakt als **`double`** konvertiert werden kann. Dies ist besonders hilfreich bei Parametern oder Funktionen, bei denen ein präziser Wert meist nicht benötigt wird, die Option ein **`double`** zuweisen zu können jedoch vorhanden sein soll. Die Platzierung eines visuellen Elements in einer UI muss beispielsweise oft nicht in sehr feinen Stufen eingestellt werden, aber ein pixelgenaues Arbeiten soll trotzdem möglich sein.

3.1.7 Reaktive Programmierung

Die Grundlage der reaktiven Programmierung bildet das „Reactive Manifesto“ aus dem Jahr 2013. Der Ansatz ist es eine Anwendung zu erstellen, die jederzeit Stimuli empfangen und darauf reagieren kann und reaktionsfähig, also „reaktiv“, bleibt [27]. Dabei gibt es nach dem Manifest vier Qualitäten, die eine Anwendung erfüllen muss: Ein System muss immer antwortbereit sein, eine Widerstandsfähigkeit besitzen, elastisch und nachrichtenorientiert sein. Mit antwortbereit wird ein System beschrieben, das innerhalb einer gewissen Zeit antwortet. Bleibt diese Antwort aus, so wird von einem Fehler ausgegangen. Widerstandsfähigkeit beschreibt die Isolation von Komponenten und Eindämmung von Fehlern. Elastizität ermöglicht es, dass eine Anwendung bei Veränderung der Last regelnd eingreift und so Grenzen vermeidet. Nachrichtenorientierung beschreibt die Verwendung von asynchronen Botschaften zwischen den einzelnen Komponenten des Systems. Ziel ist ein gute Erweiterbarkeit und Wartbarkeit.

Ein „generational garbage collector“ und die Unterstützung für kurzlebige Objekte, wie UI-Elemente in Flutter sorgen dafür, das Dart zur reaktiven Programmierung geeignet ist [8]. Dart unterstützt den asynchronen Programmierstil direkt in der Sprache und durch Schnittstellen, wobei `Future` oder `Stream` Objekte benutzt werden.

Dart Code läuft in einem Ausführungs-Thread, der bei Flutter dem UI Thread entspricht. Werden bei der Anwendung extreme CPU Berechnungen gefordert, können mehrere Threads mit `Isolates` gestartet werden. Dadurch kann Dart Code auf mehreren CPU Kernen ausgeführt werden. `Isolates` werden für I/O-Funktionen und Datenbankzugriffe nicht verwendet, sondern **`async`**/**`await`**.

Asynchrone Ereignisse können mit einem `Future` Objekt eingepflegt werden. Ein `Future` ist ein Versprechen, ein Platzhalter für ein Ereignis, das noch nicht bekannt ist. Dabei kann auf ein Ereignis mit **`await`** gewartet werden. Mit `then()` können `Future` Funktionen sequenziell abgearbeitet werden. Im Listing 3.12 ist eine **`async`** Funktion angegeben. Die `then()` Methode registriert hierbei eine Rückruffunktion (engl. callback), die vom `Future` Objekt bedient wird.

```
Future<void> outputData() async {
  final future = await getData();
  return future.then(print);
}
```

Listing 3.12: Implementieren einer asynchronen Funktion in Dart (Future).

Ein Dart `Stream` ist eine asynchrone Sequenz von Daten. Ein `Stream` übermittelt Objekte von einem Sender zu einem Empfänger (*Single-Subscription*), oder zu mehreren Empfängern (*Broadcast*). Hierbei wird der `Stream` nur aktiv, wenn ein Empfänger am anderen Ende des Streams auf Objekte wartet. Der `Stream` übernimmt die Übermittlung der Daten, der Sender muss nicht warten bis der Empfänger die Daten erhalten hat, es kann direkt ein weiteres Objekt dem `Stream` übergeben werden. Dabei werden Daten nach dem *FIFO* (First In First Out) Prinzip übermittelt.

Mit der Methode `listen()` und **`await for`** (*asynchronous for loop*) kann auf Daten von einem `Stream` reagiert werden. In Listing 3.13 werden Daten vom `Stream streamOfData` ausgegeben, sobald diese vom `Stream` übermittelt sind.

```
await for (var value in streamOfData) {
  print(value); // Prints every new value from the Stream streamData
}
```

Listing 3.13: Daten von einem Dart Stream mit await for empfangen und ausgeben.

3.2 Flutter

Flutter ist ein quelloffenes SDK zur Entwicklung plattformübergreifender mobiler Applikationen. Dabei sind Android und iOS aktuell die zwei relevanten Plattformen. Das Betriebssystem Fuchsia, das sich noch bei Google in Entwicklung befindet, soll eine Plattform für Flutter-Apps werden, wie aus Einträgen im Fuchsia Repository hervorgeht [28]. Flutter soll eine stabil hohe Bildwiederholfrequenz von 60 Hz ermöglichen [29]. Das heißt, dass spätestens alle 16.67 ms der Bildschirminhalt neu berechnet wird. Für Geräte, die über eine potente Rechenleistung und ein 120 Hz Display verfügen, kann auch 120 Hz anvisiert werden. Die spezielle Architektur von Flutter (Kapitel3.2.1) ermöglicht diese hohe Performance.

Die Basis des SDKs bildet das Google interne *Project Sky*, das im Jahr 2014 startete. Der erste Commit, von Adam Barth, hat den Titel „Open the Sky“ [30]. Die von Google im Jahr 2013

aufgekaufte Firma Flutter Ltd., die Gestenerkennungs-Technologien entwickelt hat, ist nicht am SDK Flutter beteiligt, das hier behandelt wird [31].

Flutter implementiert die 2D-Render-Engine *Skia*, die das Zeichnen von Text, Geometrie und Bildern ermöglicht und mit vielen Hardware und Software Plattformen kompatibel ist [32]. Skia ist, unter anderem aus Performance-Gründen, in C++ geschrieben. Sie wurde ursprünglich von der Firma Skia Inc. entwickelt, bevor diese 2005 von Google aufgekauft und als Open Source Software veröffentlich wurde [33]. Der Webbrowser Chrome verwendet Skia für jegliche Grafikberechnungen [34].

Bei der Entwicklung von Flutter wurde der Fokus sehr früh auf eine reaktiv-funktionale Erweiterbarkeit gelegt. Die Mindestanforderung um Flutter unter Android einzusetzen, ist die Version 4.1 Jelly Bean (2012) und unter iOS die Version 8 (2014) [35].

3.2.1 Architektur von Flutter

Der Aufbau des Flutter-Frameworks ähnelt, laut Tim Sneath, dem Produkt Manager von Flutter, eher einer Spiele-Engine, als einem Cross-Plattorm-Framework. [36]. Die Game Engine *Unity* verwendet, wie Flutter, eine optimierte Engine um auf mehreren Plattformen eine gute Performance mit einer Code-Basis zu erreichen. Eine Game Engine rendert in regelmäßigen Abständen neue Frames. Dabei versucht Flutter unnötige Berechnungen zu vermeiden und rendert Bildschirmelemente nur, wenn diese sich vom letzten Frame geändert haben. Dadurch können Teile einer komplexen UI selektiv aktualisiert werden. Flutter ist in Schichten aufgebaut, dem Framework, der Engine und als unterste Schicht, den plattformspezifischen Funktionen, siehe Abbildung 3.1 [2].

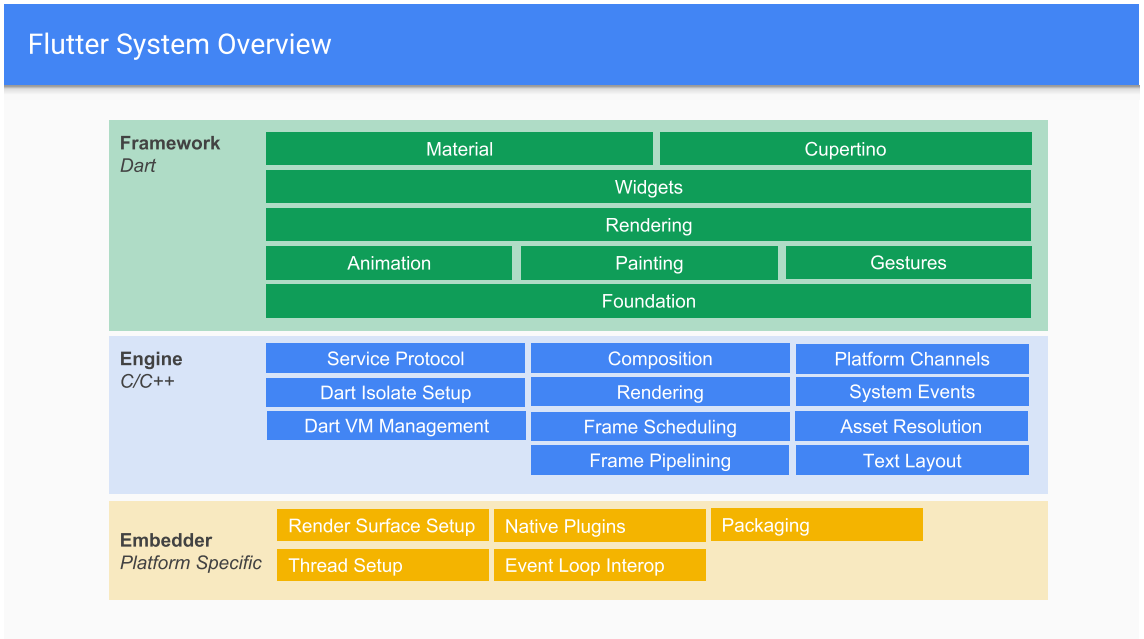

Abb. 3.1: Darstellung der Flutter System Übersicht. Quelle: Google [2]

Flutter verwendet sogenannte *Widgets,* aus denen die App aufgebaut ist, die ausführlich in Kapitel 3.2.2 beschrieben werden. Auf Framework Ebene im Flutter System wird in Dart programmiert und der Programmierer kommt wenig bis gar nicht mit der Engine und dem Rendering in Kontakt. Widgets werden auf einem Skia canvas gerendert, der von der Plattform angezeigt wird und von dem Events (Ereignisse) zurückgeschickt werden. Da Skia die UI rendert, sind die darstellbaren Elemente nicht an die Plattform gekoppelt – solange Flutter mit der Plattform kompatibel ist, kann deswegen eine beliebige UI dargestellt werden.

Flutter besitzt eine Shell, die sich auf jeder Plattform unterscheidet und die Dart-VM zur Verfügung stellt. Die Dart-VM verwaltet den canvas und ermöglicht es auf native APIs zuzugreifen. In der Shell wird die Engine ausgeführt. Die Engine beinhaltet Skia, die Dart Runtime und Platform Channels. Darauf aufbauend befindet sich das Framework, welches im Normalfall in Dart geschrieben ist. Hier kann der Entwickler Entscheidungen treffen, um die Optik der Applikation an das Betriebssystem, oder an das Unternehmensdesign anzupassen (Theme Support mit `ThemeData`).

Ein Widget ist dabei eine Blaupause, die bestimmt, mit welchem Aussehen Daten visualisiert werden sollen und wie diese sich bei Interaktion verhalten sollen. Im kommenden Kapitel wird das Widget-Konzept von Flutter vorgestellt.

3.2.2 Widgets

Kernelement einer jeden Flutter-App sind die sogenannten Widgets. Sie bilden die kleinste optische Einheit und können wie Bausteine zu einer komplexen Flutter-App zusammengesetzt werden. Die Benutzeroberfläche wird aus verschiedenen Widget Typen aufgebaut, die sich teilweise ergänzen oder aufeinander aufbauen. Die Hauptaufgabe eines Widgets ist es mit der `build()` Methode dem Framework den Aufbau des Widgets zu beschreibt. Das Flutter-Framework baut das Widget nun, angefangen mit dem kleinst möglichen Element, einem `RenderObject`, das die Geometrie eines Widgets berechnet und beschreibt, nach oben auf [37].

Um eine App zu kreieren, benötigt man verschiedene Widgets. Die Widgets werden oft in einer Reihe (mit `Row`), in einer Spalte (mit `Column`), in einem Stapel, also übereinander gestapelt und eventuell überlappend (mit `Stack`) oder in einem Container (mit `Container`) strukturiert. Eine weitere Komponente, die immer benötigt wird, ist die Darstellung von Text in der App (mit `Text`). Die verschiedenen Widget Typen können miteinander kombiniert werden und sich gegenseitig modifizieren. Im Normalfall befinden sich die Widgets innerhalb eines `MaterialWidget`, um korrekt dargestellt zu werden [37]. Viele Elemente, wie das Scroll-Verhalten in einer `ListView`, von dem der Benutzer bei iOS und Android ein ganz anderes Verhalten erwartet, wird auf die entsprechenden Plattform automatisch angepasst. Auch das Verhalten am Ende der Liste ist entweder ein *Material Glow* bei Android oder *Bounce* bei iOS. Soll eine UI mit iOS typischen Elementen aufgebaut werden, dann muss stattdessen `Cupertino` verwendet werden. Es kann auch eine Plattformabfrage erfolgen und entsprechend zwei unterschiedliche UIs entwickelt werden.

Im `MaterialWidget` wird das `Theme`, also die Gestaltung der App und das Aussehen der Texte definiert. In der Flutter-App kann dann jedes Widget darauf zugreifen und Änderungen sind nur an einer Stelle nötig. Zum Beispiel wird darin die Textausrichtung (von links nach rechts, oder rechts nach links) von `Text` Widgets festgelegt. Würde sich ein `Text` Widget nicht in einem `MaterialWidget` befinden, so müsste bei jedem `Text` Widget in der App die Ausrichtung explizit angegeben werden. Im Kontext der Internationalisierung ist eine Änderunge daher schnell möglich.

Wie mehrere Widgets miteinander verknüpft werden können zeigt Listing 3.14. Ein `Text` wird in einem farbigen (mit dem Parameter `color`) `Container` platziert. Beschreibt ein `Container` kein Widget darunter, also ein `child`, das in diesem enthalten ist, so wird der `Container` so klein wie möglich dargestellt. Eine Ausnahme ist, wenn ein Widget in der Hierarchie darüber, also ein Eltern-Widget, die Größe des Widgets beeinflusst.

```
1 Container(
2   color: Colors.lightGreenAccent,
3   child: Text("This is a Text in a colored Container"),
4 ),
```

Listing 3.14: Kombination von zwei Widgets (Text in einem Container).

Das `child` Widget des `Container` Widgets, der `Text`, gibt in diesem Fall die nötige Breite und Höhe des `Container` vor, die zur Darstellung notwendig sind. Der `Container` umschließt das `Text` Widget so kompakt wie möglich, siehe Abbildung 3.2. Bei einem kürzeren Text wird der `Container` schmäler und bei einer größeren Schriftart sowohl länger, als auch höher.

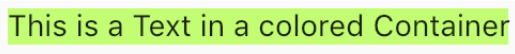

Abb. 3.2: Darstellung eines Text Widgets in einem farbigen Container.

Möchte man die Eigenschaften des `Container` ändern und die Größe verändern, so kann der `Container` parametrisiert werden (Höhe `height` und Breite `width`), wie in Listing 3.15.

```
1 Container(
2   color: Colors.lightGreenAccent,
3   height: 50.0,
4   width: 250.0,
5   child: Text("This is a Text in a colored Container"),
6 ),
```

Listing 3.15: Parametrisierung der Farbe und Dimension eines Container Widgets.

Der `Text` wird im `Container` platziert und der `Container` gibt nun seine Dimensionen vor. Dies sieht man in Abbildung 3.3, denn der `Text` ist nicht im größeren `Container` zentriert, wie in Abbildung 3.2.

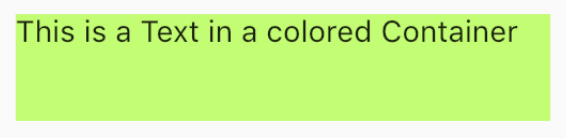

Abb. 3.3: Text in einem, mit height und width, parametrisierten Container.

Eine weitere Möglichkeit ist es, ein Widget mit einem `Padding` Widget zu umgeben, oder den `Container` Parameter `padding` anzugeben. In Listing 3.16 wird der Text von einem Padding Widget umgeben, das sich in einem weiteren Container befindet. Das `child` des `Padding` Widgets ist somit das `Text` Widget. Padding definiert den freien Raum um das `Text` Widget herum. In diesem Beispiel wird jede Seite vom `child`, also links, rechts, oben und unten ein Abstand von 10 logischen Pixeln hinzugefügt und der umgebende `Container`, dem keine Dimensionen direkt vorgegeben wurde, bekommt die neue Größe beim Build-Vorgang mitgeliefert und passt sich dementsprechend an. Der innere `Container` hat weiterhin dieselbe Farbe wie in Abbildung 3.3. Der äußere `Container` ist mit der Farbe Schwarz parametrisiert.

```
1  Container(
2    color: Colors.black,
3    padding: EdgeInsets.all(10.0),
4    child: Container(
5      color: Colors.white,
6      child: Padding(
7        padding: const EdgeInsets.all(10.0),
8        child: Text("This is a Text in a colored Container"),
9      ),
10   ),
11 ),
```

Listing 3.16: Kombination eines, mit padding parametrisierten, Container und eines Padding Widgets.

In Abbildung 3.4 ist sowohl das Padding des `Container` Widgets (schwarzer Rahmen), als auch das Padding des `Padding` Widgets (grüner Bereich um den `Text` herum) sichtbar.

Abb. 3.4: Kombination eines, mit padding parametrisierten, Container und eines Padding Widgets.

Um Widgets in einer UI sinnvoll anzuordnen, werden Abstände zwischen den Widgets definiert, da ansonsten Widgets ohne Abstand zueinander angezeigt werden würden (siehe Abbildung 3.5mit zwei Widgets in einer `Column`).

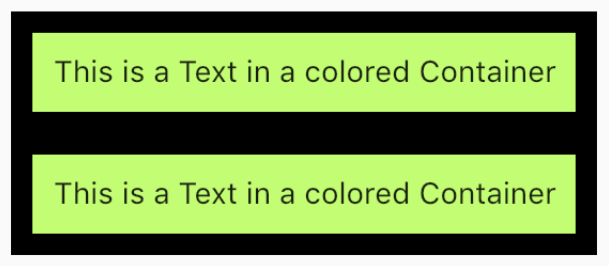

Abb. 3.5: Zwei Widgets in einer Row ohne Margin.

Analog zu `Padding`, das den Abstand in einem Widget regelt, gibt es auch ein Widget, das den Abstand zu anderen Widgets angibt. Dieser Abstand kann mit `Margin` eingestellt werden (siehe Abbildung 3.6).

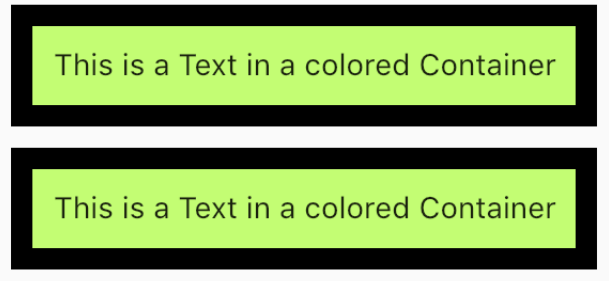

Abb. 3.6: Zwei Widgets in einer Row mit Margin

Das Widget Konzept von Flutter erlaubt die weitere Strukturierung der Widgets. `Row` kann mit `Column` kombiniert werden. Eine `Row` besteht, wie eine `Column`, aus mehreren `child` Widgets. In Listing 3.17 und Abbildung 3.7 wird ein Widget verwendet, das je nach Initialisierungsparameter, einen `Container` mit der entsprechenden Zahl anzeigt.

```
Column(
  children: <Widget>[
    WidgetNumbered(1),
    Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        WidgetNumbered(2),
        WidgetNumbered(3),
      ],
    ),
  ],
),
```

Listing 3.17: UI Aufbau durch Kombination von Row und Column Widgets.

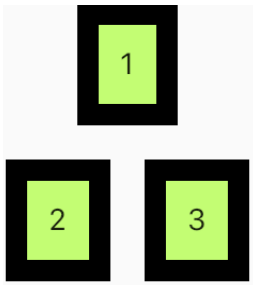

Abb. 3.7: Drei Widgets in einer Spalte und zwei Reihen angeordnet.

State

State beschreibt die Logik und der interne Zustand eines `StatefulWidget` oder der App [38]. Unter State versteht man Informationen, die synchron gelesen werden können, wenn das Widget gebaut wird und die sich im Lebenszyklus dieses Widgets ändern können. Der State besteht aus den Daten, die benötigt werden, um die UI-Elemente aktualisiert anzeigen zu können. Flutter unterscheidet zwischen *ephemeral State* und *App State* [39]. Ephemeral State (auch UI State oder *local* State genannt) ist der State, den ein Widget besitzt. Dies kann zum Beispiel der Fortschritt einer Animation sein, oder ein selektiertes Element eines Navigationselements. Der *local* State befindet sich im `State` Objekt des `StatefulWidget`. In Listing 3.17 ist der *local* State die Zahl, die im Container angezeigt wird. Diese Zahl kann z.B. mit einem vordefinierten Wert initialisiert werden. Das selbe Widget kann beliebig oft gebaut werden, mit jeweils einem unterschiedliche State. In diesem Fall gibt es das Widget dreimal, mit jeweils getrenntem State. Wird die App beendet und wieder geöffnet, so wird der *local* State neu initialisiert. Der *local* State spielt außerhalb des Widgets keine Rolle.

Unter App State, oder auch *shared* State, versteht man die Daten, die in der App benutzt werden,

oder beim erneuten Starten der App vorhanden sein sollen. Diese Daten können z.B. Einstellungen, Login-Informationen oder bereits gelesene Nachrichten in einer Chat App sein. Um den App State zu verwalten, gibt es nicht nur eine mögliche Lösung. Der State kann mit der Methode `setState()` eines `StatefulWidget`, einem `InheritedWidget` und *Scoped Model*, *Redux*, *BLoC* / *Rx* oder *MobX* verwaltet werden. Faktoren, die die Wahl des State Management Systems beeinflussen werden in Kapitel 3.2.5 betrachtet. Je nach Situation können Informationen dem *local* State oder dem *ephemeral* State zugewiesen werden. Grundsätzlich gilt, dass die Daten im *local* State nur von einem Widget verwendet werden.

StatelessWidget

Ein `StatelessWidget` ist ein Widget, das im Normalfall einmal gebaut wird und anschließend angezeigt wird. Es kann sich selbst und andere Widgets nicht aktualisieren. Es handelt sich somit um ein statisches Widget. Ein `StatelessWidget`, genauso wie ein `StatefulWidget`, besitzt immer eine `build()` Methode, die ein Widget zurück gibt. Der Minimalaufbau ist in Listing 3.18 dargestellt. Ein `StatelessWidget` kann über weitere Variablen verfügen, die mit einem Konstruktor initialisiert werden.

```
1 class NameOfAStatelessWidget extends StatelessWidget {
2   @override
3   Widget build(BuildContext context) {
4     return Container();
5   }
6 }
```

Listing 3.18: Aufbau eines StatelessWidgets.

Im Konstruktor eines `StatelessWidget` werden immer **`final`** Variablen initialisiert. Diese Variablen werden zur Laufzeit beim Bau des Widgets zugeordnet. Durch die `build()` Methode hat ein `StatelessWidget` ein `child` Widget, das vom Typ `StatelessWidget` oder `StatefulWidget` sein kann.

StatefulWidget

Der simpelste Aufbau eines `StatefulWidget` ist in Listing 3.19 beschrieben. Die `build()` Methode beschreibt, wie das Widget zusammengesetzt ist. Das `StatefulWidget` besteht hierbei aus dem für den Programmierer kleinsten UI-Element, nämlich einem Container, der nichts enthält und somit in der App nicht dargestellt wird. Zusätzlich können noch die Methoden `createState()`, `initState()`, `didChangeDependencies()`, `setState()`, `deactivate()` und `dispose()` verwendet werden.

```
class WidgetA extends StatefulWidget {
  @override
  _WidgetAState createState() => _WidgetAState();
}

class _WidgetAState extends State<WidgetA> {
  // init state here
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

Listing 3.19: Aufbau eines StatefulWidgets.

Mit dem Bau eines `StatefulWidget` wird ein `State` Objekt erstellt. In Listing 3.19 wird ein Widget `WidgetA` definiert, dem mit der Methode `createState()` der `State _WidgetAState` zugewiesen wird. Dieses Objekt beinhaltet den gesamten State des Widgets, welcher mutable ist. Der State bleibt erhalten, auch wenn das Widget erneut gebaut wird. Ein `StatefulWidget` kann nicht nur einmal in einer App verwendet werden, sondern wiederverwendet werden. Hierbei wird das `StatefulWidget` in den entsprechenden `BuildContext` eingebunden und es wird jeweils ein von anderen Instanzen des Widgets unabhängiger State zugewiesen. Im Anschluss werden die einzelnen Methoden von dem `StatefulWidget` vorgestellt, welche den Lebenszyklus beeinflussen.

createState() Diese Methode erstellt ein `State` Objekt und muss in jedem `StatefulWidget` implementiert sein. Jedes `State` Objekt wird mit einem `BuildContext` in Verbindung gebracht. Der `BuildContext` beschreibt den Ort im Widget Baum, an dem das Widget platziert werden soll. Dieser Ort kann sich ändern. Ab dem Zeitpunkt, ab dem der `BuildContext` zugewiesen ist, gilt das `State` Objekt als `mounted`.

initState() Diese Methode wird aufgerufen, wenn das Objekt in den Baum eingebunden wird, also einmal pro erstelltem `State` [40]. Sollen Initialisierungen durchgeführt werden, die vom Ort des Widgets im Widget Baum (`BuildContext`) abhängen, so muss `initState()` überschrieben werden und zu Beginn **`super`**`.initState()` aufgerufen werden, wie in Listing 3.20.

```
initState() {
  super.initState();
  // Add listener
  dataStream.listen((data) {
  _WidgetUpdate(data);
  });
}
```

Listing 3.20: Funktion initState(), um ein StatefulWidget zu initialisieren.

didChangeDependencies() Diese Methode wird unmittelbar nach `initState()` aufgerufen. Ändert sich ein Objekt, das das Widget benötigt, um gebaut zu werden, wird die Methode auch aufgerufen. Da nun der `BuildContext` definiert ist, ist hier die erste Möglichkeit, Methoden, die auf den `BuildContext` zugreifen, zu benutzen. Daher kann erst hier `inheritFromWidgetOfExactType` aufgerufen werden, um das nächste Widget eines gegebenen Typs zu erhalten [41]. So kann dann beispielsweise auf das Theme der App zugegriffen werden. Das Prinzip des `InheritedWidget`, also auf den Zugriff auf den `State` eines Widgets, das sich weiter oben im Widget Baum befindet, wird näher im Kapitel 3.2.5 abgehandelt. Wird die Methode `didChangeDependencies()` überschrieben, so muss `super.didChangeDependencies()` aufgerufen werden.

didUpdateWidget() Diese Methode wird aufgerufen, wenn sich die Konfiguration des Widgets ändert, oder das darüber liegende Widget sich verändert und dieses Widget deswegen neu gebaut werden muss [41]. Flutter ruft die Methode `build()` (siehe Kapitel 3.2.2) auf, nachdem `didUpdateWidget()` abgearbeitet ist. Ein Aufruf von `setState()` signalisiert dem Framework, dass sich der `State` verändert hat und hat daher einen Aufruf von `build()` zur Folge, weshalb dieser Aufruf in dieser Methode nicht sinnvoll ist. Von einem `StatefulWidget` ist in Abbildung 3.8 der Lebenszyklus dargestellt.

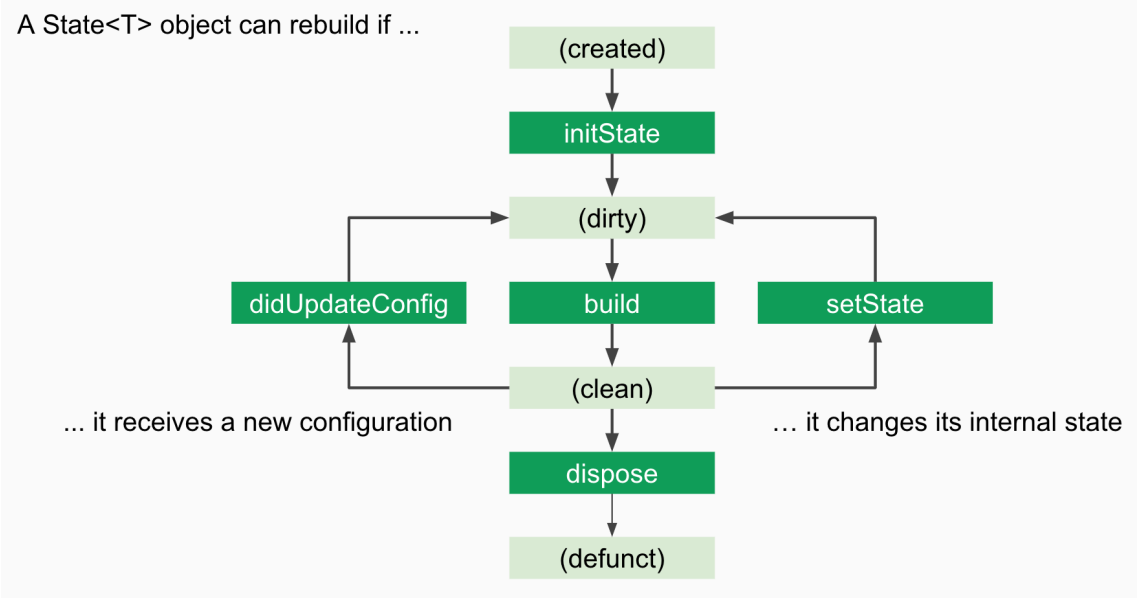

Abb. 3.8: Schaubild des Lebenszyklus eines StatefulWidget. Quelle: Google [3].

build() Jedes Widget besitzt eine eigene `build()` Methode [42]. Die Methode definiert, wie das Widget aufgebaut ist, welches zurückgegeben wird – mindestens ein leerer `Container()` (siehe Listing 3.19). Zeitlich wird die Methode unmittelbar nach `initState()`, `didUpdateWidget()` oder dem Aufruf von `setState()` ausgeführt. Ein Aufruf kann auch erfolgen, nachdem ein Objekt, von dem der `State` des Widgets abhängig ist, sich ändert, oder der Ort des Widgets im Widget Baum verändert wird. Die Methode wird also aufgerufen wird, wenn das Widget neu gebaut werden soll. Das Flutter-Framework ersetzt den Teilbaum unterhalb dieses Widgets mit dem zurückgegebenen Widget.

setState() Diese Methode benachrichtigt das Flutter-Framework, dass sich die Daten verändert haben, die Teil vom State sind [40]. Bei Änderungen am internen State des `State` Objekts ist daher der Aufruf der Methode nötig. Wird diese Methode nicht aufgerufen, so kann der Aufruf der `build()` Methode nicht geplant aufgerufen werden und die UI-Elemente werden in der Folge nicht aktualisiert und zeigen veraltete und falsche Informationen an. Die Methode darf nur aufgerufen werden, wenn das Widget im Baum noch vorhanden ist. Dies kann mit der Instanzvariable `mounted` des `State` Objekts überprüft werden. Ist der Zustand **`true`**, so befindet sich das Objekt im Baum, bei **`false`** nicht.

```
var newState = await calculateNewState();
setState(
  () => _myState = newState;
);
```

Listing 3.21: Aufruf von setState(), sodass das Framework das Widget aktualisiert.

deactivate() Diese Methode wird aufgerufen, wenn das Objekt aus dem Baum entfernt werden soll [43]. Das Objekt kann vom Framework an einer anderen Stelle im Baum wieder eingebunden werden. Dabei ruft das Framework `build()` auf. In dieser Methode werden alle Verbindungen des Objekts mit anderen Elementen gelöst. Wird `deactivate()` überschrieben, muss am Ende der Methode **`super`**`.deactivate()` aufgerufen werden.

dispose() Diese Methode wird aufgerufen, wenn das Objekt permanent aus dem Baum entfernt wird, also nicht mehr gebaut werden soll [44]. Das `State` Objekt ist *unmounted*. Jetzt darf `setState()` nicht mehr aufgerufen werden, da das `State` Objekt nicht mehr existiert. In der `dispose()` Methode werden alle Ressourcen, wie Animationen oder Streams, freigegeben, die vom `StatefulWidget` benutzt werden. Wird `dispose()` überschrieben, so muss am Ende der Methode immer **`super`**`.dispose()` aufgerufen werden.

3.2.3 Aufbau einer Flutter-App

Eine Flutter-App startet immer mit der `main()` Funktion. In `main()` wird mit der Funktion `runApp()` ein Widget als das Root-Widget im Widget Baum verankert und im Vollbildmodus angezeigt [37]. In Listing 3.22 ist das Root-Widget das Widget `Home`. Regulär wird `runApp` nur einmal aufgerufen. Jeder weitere Aufruf tauscht das Root-Widget aus [45].

```
void runApp ( Widget Home )
```

Listing 3.22: Festlegung des Root-Widgets einer Flutter-App mit der Funktion runApp.

Smartphones oder auch Tablets verfügen nur über eine begrenzte Anzeigefläche. Daher werden Informationen in einer App auf mehrere Screens aufgeteilt. Android benutzt zur Verwaltung der Navigation den *Activity Stack* und iOS den *UINavigationController* [46, 47]. Prinzipiell wird ein Stack benutzt, auf dem der neue Screen abgelegt wird, um problemlos zum letzten Screen zurückspringen zu können. Unter Android gibt es keine einheitlichen Animationen, mit denen die Navigation erfolgt,

bei iOS jedoch schon. Die Navigation soll sich bei beiden Betriebssystemen nativ anfühlen oder einen eigenen Stil aufweisen, um nicht deplatziert zu wirken.

Um von diesem Screen auf einen weiteren zu navigieren, muss in Flutter ein Navigator benutzt werden. Da bei Flutter im Normalfall der Screen aus einem Widget vom Typ `MaterialApp` aufgebaut ist, wird zur Navigation die Subklasse `MaterialPageRoute` von `PageRoute` verwendet. Um eine Datei übersichtlicher zu gestalten, kann die Navigation in eine Methode, wie z.B. `_pageRouteDebugView()` in Listing 3.23, zur Navigation auf eine andere Seite (im Beispiel: `DebugView()`), ausgegliedert werden.

```
1 void _pageRouteDebugView() {
2   print('Navigate to Debug View');
3   Navigator.push(
4     context, MaterialPageRoute(builder: (context) => DebugView()));
5 }
```

Listing 3.23: Navigation auf einen anderen Screen mit dem MaterialPageRouter.

Mit dem `Navigator` wird (mit der `push` Methode) eine neue `MaterialPageRoute` auf dem Stack abgelegt [48]. Die Navigation zur vorherigen Seite kann mit dem Aufruf von `Navigator.pop(context)` erfolgen. Hierbei kann ein `result` Parameter übergeben werden. Die asynchronen Eigenschaften von Dart ermöglichen es, mit **`await`** den Programmablauf vom Ergebnis des `result` Parameters abhängig zu machen. Der Aufruf des Navigators, um zurück zu navigieren, kann entweder durch Code ausgelöst werden, oder durch einen Klick auf den Zurück-Pfeil, links oben in der App.

Die verschiedenen Widgets, die bei der Entwicklung einer Flutter-App Verwendung finden, lassen sich in strukturierendes Element (z.B. ein Menü), stilisierendes Element (Schriftart) und Layout verändernde Widgets (z.B. Zentrierung eines Widgets mit `Center`) unterteilen [49].

Jeder neue Screen in einer App baut auf einem strukturierenden Widget auf. Dabei implementiert das `Scaffold` Widget ein Grundgerüst, bestehend aus Material-Design-Elementen [50, 10]. Das `Scaffold` Widget füllt den gesamten zur Verfügung stehenden Platz aus. Die Widgets aus denen das `Scaffold` Widget bestehen kann, sind eine Leiste am oberen Bildschirmrand (`AppBar`), ein Floating Action Button (`FloatingActionButton`), ein Navigationsmenü am linken Bildschirmrand (`Drawer`), eine Navigationsleiste am unteren Bildschirmrand (`BottomNavigationBar`), oder eine Nachrichtenleiste mit Interaktionselementen (`Snackbar`) und ein `body`. Diesem können Widgets zugewiesen werden aus denen der Großteil der UI aufgebaut ist, welche wiederum aufeinander aufbauen können.

Abhängigkeits-Management mit Pubspec

In Flutter gibt es die Datei *Pubspec.yaml*, in der sämtliche Abhängigkeiten der App definiert sind. Diese Datei definiert die eingebundenen externen Dateien und deren Versionskompatibilität. In der Datei werden *Assets*, also Bilder, Videos, Animationen etc. gelistet. In Listing 3.24 werden einzelne PNG Dateien genannt, die die App benötigt. Ein ganzes Verzeichnis, (`assets/animations/`) kann auch hinzugefügt werden. Das / am Ende der Zeile signalisiert die Einbindung eines Ordners. Es wird immer nur der angegebene Ordner ohne Unterverzeichnisse eingebunden [51].

```
1 flutter:
2   assets:
3     - assets/image1.png
4     - assets/image2.png
5     - assets/animations/
```

Listing 3.24: Einbindung von assets in der Datei Pubspec.yaml.

Die Datei *Pubspec.yaml* wird mit der Auszeichnungssprache *YAML* geschrieben [52]. Darin sind Metadaten gespeichert, um Abhängigkeiten und Informationen zu bündeln. Jedes *Pub package* hat einen Namen und eine Versionsnummer. Die Angabe der unterstützten Dart SDK Version (*Environment*) ist seit Dart 2.0 Pflicht. Mit `flutter packages get` können die Abhängigkeiten geladen werden. Will man auf die neueste kompatible Version upgraden geht dies mit `flutter packages upgrade`.

Konfiguration des App Icons

Es gibt einige Situationen, in denen auf Assets zugegriffen werden muss, bevor das Flutter-Framework geladen ist. Dies ist zum einen das Anzeigen des App Icons, welches im Betriebssystem angezeigt wird, ohne dass das Flutter-Framework involviert ist, und zum anderen der Startbildschirm, der beim Start der App erscheint. Der Startbildschirm ist jedoch weniger relevant, da dieser nur einen Moment sichtbar ist und einen neutralen Bildschirm anzeigt. Standardmäßig ist ein Flutter-App Icon eingestellt, siehe Abbildung 3.9. Bei Android navigiert man im Flutter Projekt im Android-Verzeichnis zu `app/src/main/res`. Darin können die Platzhalter-Bilder ersetzt werden. Eine Beschreibung der Eigenschaften, die ein Icon besitzen muss, findet man in den Material Design Richtlinien im Kapitel Product Icons [53]. Bei iOS funktioniert das Ganze analog. Im iOS-Verzeichnis können die Platzhalter Icons durch passende Icons ersetzt werden, die den Human Interface Guidelines entsprechen [9].

Abb. 3.9: Voreingestelltes App Icon bei Flutter

3.2.4 Hot Reload und Hot Restart

Wird eine Flutter-App in der Entwicklung auf einem Simulator, oder auf einem Smartphone, aufgespielt, so wird der JIT Compiler verwendet. Das Flutter-Team von Google verwendet den Marketingbegriff *Hot Reload*, wenn ein geänderter Code in der laufenden Dart-VM eingepflegt wird [54]. Dies hat im Gegensatz zur AOT-Kompilierung den großen Vorteil, dass jederzeit der laufende Code aktualisiert werden kann. Dabei wird der Code auf dem Gerät, auf dem die VM läuft, mit dem Code auf dem Entwicklungsgerät verglichen und nur die Änderungen übermittelt. Widgets werden nun entweder aus dem Widget Baum entfernt, hinzugefügt oder aktualisiert. Der größte Vorteil ist es, dass minimale Änderungen am Code schnell in eine laufende App eingepflegt werden können und somit eine schnelle Iteration möglich wird. Dabei wird in der Flutter Dokumentation der Begriff *Sub-Second Hot Reload* verwendet, da es je nach Leistungsfähigkeit des Computers nur Sekundenbruchteile dauert, bis die VM aktualisiert ist. Da weniger Daten auf das Gerät übermittelt werden wird Zeit gespart.

Hot Reload hat nicht nur eine direkte Zeitersparnis zur Folge, sondern erspart auch das Navigieren auf in der Hierarchie der App verschachtelten Bildschirme, wenn auf diesen Änderungen erfolgen. Dadurch ändert sich die Art und Weise, wie programmiert wird. Bei einer kleinen Änderung am Code genügt ein kurzes Speichern: Der State bleibt erhalten, aber der neue Code wird eingepflegt und kann getestet werden. Da der aktuelle State erhalten bleibt, werden Änderungen an globalen Zuweisungen und `static` Objekten nicht übernommen. Daher gibt es zusätzlich zu *Hot Reload* die Möglichkeit, mit *Hot Restart* die App neu zu initialisieren um in den Zustand zu kommen, in dem die App sich beim ersten Öffnen befindet – aber mit aktualisierten Code. *Hot Reload* aktualisiert `const` Variablen, aber keine `final` Variablen.

3.2.5 State Management

Im Gegensatz zu iOS und Android verwendet Flutter nicht den imperativen Stil, um die UI zu programmieren, sondern den deklarativen Stil [55, 56]. Eine Flutter-App oder ein einzelnes Widget, besitzt einen State. Mit der Methode `build` wird das Aussehen der App, mit dem State als Datengrundlage, beschrieben. Diese Beschreibung wird in eine Benutzeroberfläche umgewandelt und ausgegeben. Änderungen am State sollten immer den Aufruf der Methode `build` zur Folge haben. Ein jedes `StatefulWidget` besitzt einen *local* State. Oft genügt es, wenn ein Widget seinen eigenen State besitzt und diesen verwaltet. Jedoch ist es oft nötig, auf den State eines anderen Widgets

zuzugreifen. Es gibt bei Flutter keinen optimalen, oder empfohlenen Weg, wie das State Management umgesetzt werden sollte. Daher werden nun mehrere Wege der Umsetzung vorgestellt.

State eines Child Widgets erhalten Auf den State eines unmittelbaren `child` Widgets kann durch dessen `name` zugegriffen werden. Die grundsätzliche Anordnung ist in Abbildung 3.10 dargestellt. Diese eindeutige ID steht zur Build Zeit fest. Diese ID kann per `key` Parameter im Child Widget definiert werden. Das Eltern-Widget kann nun mit `key.currentState` auf den `State` zugreifen.

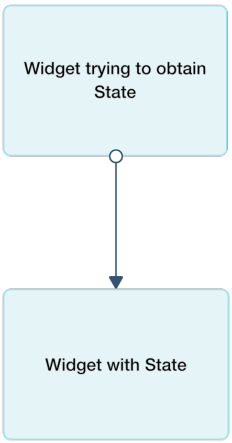

Abb. 3.10: Widget Baum mit einem Child Widget, auf dessen State zugegriffen werden soll.

State eines Ancestor Widgets erhalten Ein Ancestor Widget besitzt die Eigenschaft, dass es ein im Widget Baum hierarchisch übergeordnetes Widget ist. In Abbildung 3.11 befindet sich das Ancestor Widget oben. Ein weiteres Widget, das sich im Baum unter dem Ancestor Widget befindet, möchte auf den `State` dessen zugreifen und ist in der Abbildung rechts unten dargestellt.

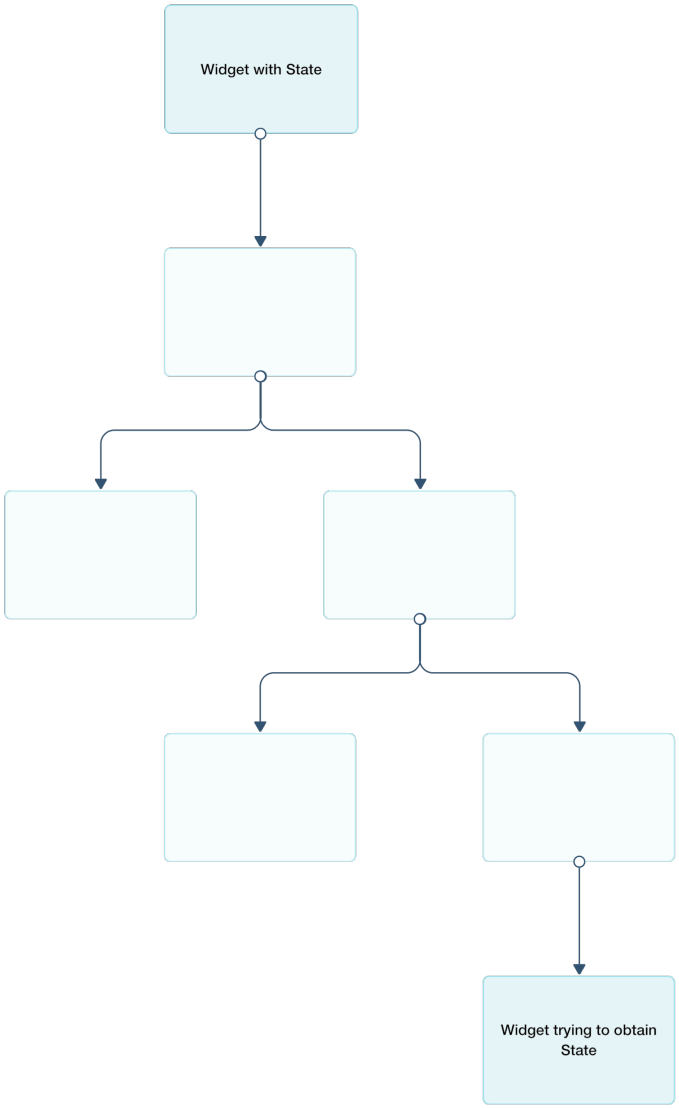

Abb. 3.11: Widget Baum mit einem Ancestor Widget, auf dessen State zugegriffen werden soll.

Der State eines `StatefulWidget` kann public, also direkt zugänglich sein, oder mit Gettern und Settern zugänglich sein. Der Zugriff auf einen **`String`** `_title` im Ancestor Widget ist in Listing 3.25 zu sehen.

```
1 class AncestorWidgetState extends State<AncestorWidget>{
2   String _title;
3   String get title => _title;
4 }
```

Listing 3.25: Zugriff auf den State eines Ancestor Widgets verwalten.

Das im Baum darunter angeordnete `child` Widgets zeigt lediglich den `State` des Anchestor Widgets

an. Problematisch hierbei ist, dass das `child` Widget Änderungen des States des Anchestor Widgets nicht mitbekommt. Es könnte passieren, dass das `child` Widget veraltete und somit falsche Daten hat und diese anzeigt.

State Management mit einem InheritedWidget Mit einem `InheritedWidget` können Daten, Informationen und somit der `State` in der App verteilt werden. Das `InheritedWidget` ist ein Widget, das als Eltern-Widget über einem Widget Baum platziert wird, siehe Abbildung 3.12.

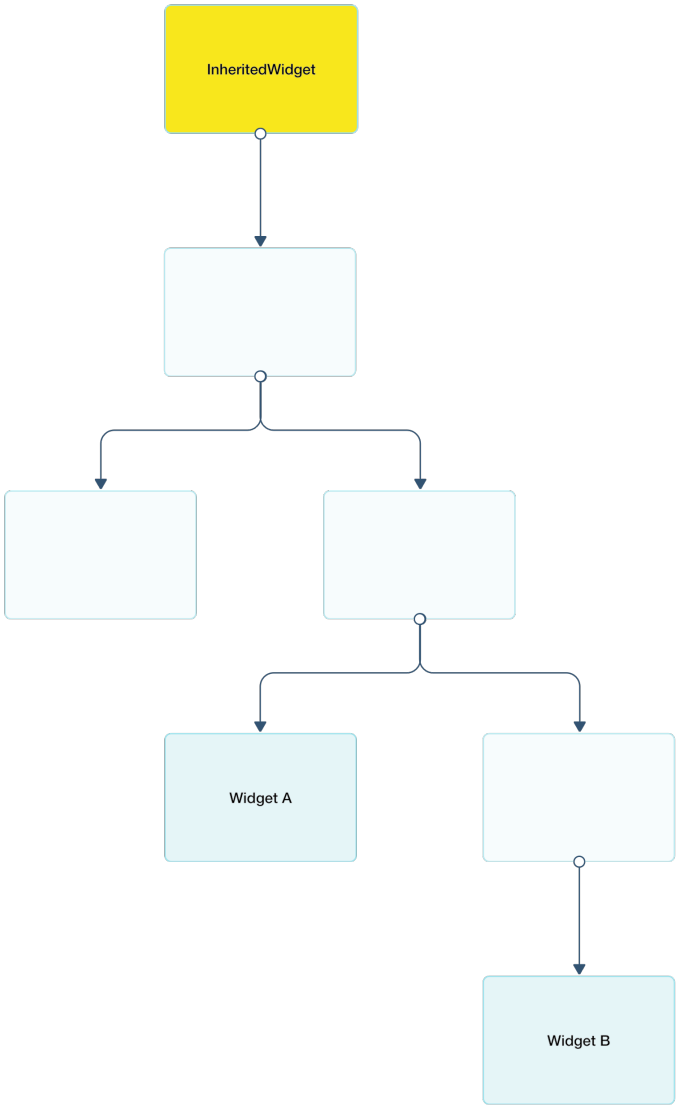

Abb. 3.12: Widget Baum mit einem InheritedWidget.

Alle Widgets dieses Teilbaums besitzen Zugriff auf die Daten, die vom `InheritedWidget` zur Verfügung gestellt werden. Um ein Widget zu aktualisieren muss diesem mitgeteilt werden, dass sich der `State` geändert hat. Dies geschieht mit der Methode `updateShouldNotify()` von dem `InheritedWidget` [57].

Beispielsweise ist `Widget A` in Abbildung 3.12 eine `Checkbox`, mit dem ein Element markiert und zu einer Favoritenliste hinzugefügt wird, die von `Widget B` dargestellt wird. Es muss auf den `State` des anderen Widgets zugegriffen und zeitnah das betreffende `Widget B` aktualisiert werden.

State Management mit dem BLoC-Pattern Google verwendet zum State Management von Flutter-Apps das *BLoC-Pattern*. Dabei steht BLoC für „Business Logic Component“, also Geschäftslogik-Komponente. Mit Geschäftslogik sind jegliche Bestandteile der App gemeint, die aktiven Einfluss auf den State der App haben. Das BLoC-Pattern ist ein relativ neues Pattern, das von Paolo Soares auf der Dart Conference 2018 vorgestellt wurde und auch bei der Google I/O 2018 gezeigt wurde [58, 59]. Intention des BLoC-Pattern ist es, Geschäftslogik einer App in BLoCs zu verschieben und dadurch Widgets simpel und wiederverwendbar zu gestalten - auch für mehrere Betriebssysteme, wie bei der App *AdWords* von Google (Android, iOS und Web). Davor gab es drei verschiedene Codes, bei denen jeweils die Programmierung und Tests der Logik durchgeführt werden mussten.

Jede Komponente die „komplex genug“ ist, soll laut Paolo Soares einen dazugehörigen BLoC erhalten, der den State enthält. Dies soll, nach Paolo Soares, jedoch nicht dazu führen, dass jedes Widget einer Flutter-App einen eigenen BLoC erhält, sondern der Entwickler dies für jeden Anwendungsfall entscheiden muss [59].

Widgets sollen Eingaben in den BLoC so senden, wie diese anfallen, also ohne weitere Vorverarbeitung. Ausgaben eines BLoCs sollen möglichst, so wie diese ausgegeben werden angezeigt werden - ebenfalls mit möglichst wenig Verarbeitung. Die Ein- und Ausgänge eines BLoCs sind dabei ausschließlich `Streams` und `Sinks`. Abhängigkeiten sollen eingebunden und plattformunabhängig sein. Die exakte Implementierung wird vom BLoC-Pattern nicht vorgegeben. In Abbildung 3.13 ist ein BLoC mit vier angeschlossenen Widgets gezeigt. Zwei Widgets liefern Daten in den BLoC ein und zwei weitere Widgets verwenden Daten von dem BLoC. Ein Widget kann aber sowohl Daten an den BLoC liefern, als auch Daten vom BLoC anzeigen.

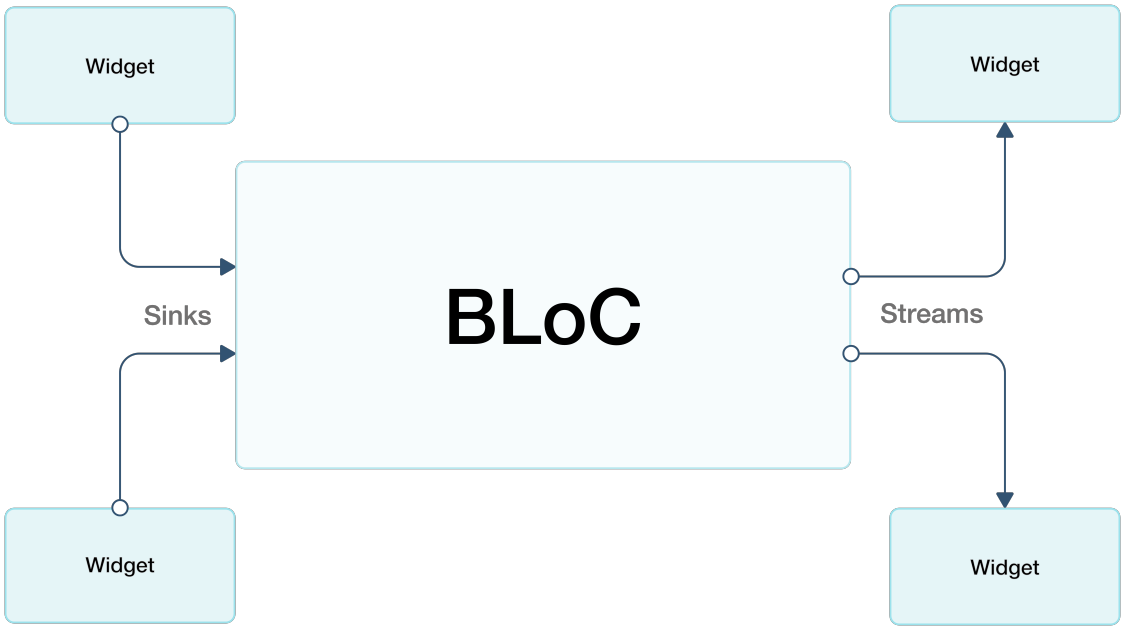

Abb. 3.13: Einbettung eines Business Logic Component (BLoC) in einer Flutter-App mit vier angeschlossenen Widgets. [4]

Das BLoC-Pattern benutzt Prinzipien der reaktiven Programmierung, um den Datenfluss in der App zu kontrollieren und verständlich zu organisieren. Ein BLoC enthält, wie es der Name schon sagt, die Geschäftslogik der App und möchte diese möglichst weit von der Präsentationsebene trennen. Durch die Auslagerung des States können `StatefulWidgets` oft in `StatelessWidgets` verändert werden, da der State extern gespeichert ist. Die Widgets müssen sich nicht mehr um die Aktualisierung des States mit `setState()` kümmern. Die Geschäftslogik enthält den State der App, also gespeicherte Daten, Funktionen, die damit interagieren und Ausgabefunktionen, die den State ausgeben. Da ein BLoC nur einmal implementiert wird, müssen Tests nur einmalig geschrieben werden, um die Funktionalität zu überprüfen. Die UI kann beliebig geändert werden, ohne die Geschäftslogik zu beeinflussen und wird also nur noch zur Darstellung des States verwendet. Soll eine einzige Funktionalität abgeändert werden, so genügt eine zentrale Änderung, die nicht mehrmals wiederholt werden muss, was die Fehlerwahrscheinlichkeit reduziert.

Im BLoC werden Events verarbeitet und der State intern aktualisiert. Widgets sind mit dem BLoC verbunden und werden durch diesen asynchron aktualisiert. Sobald neue Daten vorhanden sind, werden die betroffenen Widgets neu gerendert. Eine App kann nicht nur einen, sondern mehrere BLoCs enthalten. Unterschiedliche BLoCs sind für unterschiedliche Funktionalitäten einer App zuständig und können auch untereinander kommunizieren.

Die Kommunikation der Ein- und Ausgänge wird von einem `StreamController` verwaltet, der die `dart:async` Library verwendet. BLoCs können in einer App mehrmals initiiert werden, mit jeweils unterschiedlichen State. Will man Daten mit dem BLoC austauschen, so genügt es eine Instanz des BLoCs zu erzeugen und mit der `listen` Methode Events zu abonnieren, oder mit der `add` Methode

Daten in den BLoC zu geben. Mit dem `StreamBuilder` Widget kann ein Widget mit den Daten aus dem BLoC gebaut werden, das sich aktualisiert, sobald der `Stream` neue Daten liefert.

3.2.6 Flutter Platform Channels

Sobald Zugriff auf Sensoren, oder Benachrichtigungen erfolgen soll, wird eine Verlinkung mit der Plattform benötigt. Die bidirektionale Kommunikation zwischen dem Betriebssystem und dem Dart Code geschieht über die Klasse `MethodChannel` [60]. Nachrichten aus der Flutter-App werden aufseiten der Plattform von den Methoden `MethodChannel` (Android), oder `FlutterMethodChannel` (iOS) empfangen. Wird der plattformspezifische Code in mehreren Flutter-Apps benötigt, kann dieser in einem Flutter *package* eingebettet werden. Oft gibt es für den benötigten Anwendungsfall ein geeignetes *package*, das im Dart Code direkt verwendet werden kann. In Listing 3.26 sieht man den `MethodChannel channel` und ein möglichen Anwendungsfall für diesen: Die Abfrage der Helligkeit des Systems. Hierbei wird mit der Methode `invokeMethod()` ein Parameter an die Plattform geschickt (hier: *`'getBrightnessLevel'`*).

```
1  static const channel = const MethodChannel('samples.kurzdigital.com/brightness');
2  int brightnessLevel;
3
4  Future<void> _getBrightness() async {
5    try {
6      final int _brightness = await channel.invokeMethod('getBrightnessLevel');
7    } on PlatformException catch (e) {
8      // Possibility to catch error (e.message) here.
9    }
10   setState(() {
11     brightnessLevel = _brightness;
12   });
13 }
```

Listing 3.26: Erstellung eines MethodChannel um Nachrichten an die Plattform zu verschicken.

Das Gegenstück wird für jede unterstützte Plattform separat implementiert. Bei Android (in Java) wird hierbei beispielsweise in der *`MainActivity`* der `MethodChannel` und ein `MethodChannelHandler` implementiert, der mit dem Namen des `MethodChannel` verbunden wird. In Listing 3.27 sieht man, dass der `MethodCall call` abgefangen wird. In der Methode `getBrightnessLevel` wird nun der Wert mit Java Code vom Betriebssystem abgefragt und zurück an die Flutter-App geliefert (mit `result.success()`).

```
public void onMethodCall(MethodCall call, Result result) {
    if(call.method ==  'getBrightnessLevel'){
      result.success(getBrightnessLevel());
    }
    else  result.notImplemented(); // If MethodCall not implemented
}
```

Listing 3.27: Erstellung eines MethodChannel um Nachrichten an die Plattform zu verschicken.

Es ist jedoch auch möglich, in einer Android-, oder iOS-App Flutter (Screens) zu integrieren. Diese Einbindung wird aktuell im Flutter Repository als experimentell eingestuft [61].

3.3 Bluetooth

Zur Funkübertragung von Daten über kurze Distanzen hat sich der Industriestandard Bluetooth gemäß *IEEE 802.15.1* etabliert. BLE wird bei Android ab der Version 4.3 und bei iOS ab der Version 5 unterstützt [62, 63]. Bluetooth ersetzte viele Kabelverbindungen zwischen verschiedenen Geräten. Die Version 1.0 wurde 1999 veröffentlicht. Bluetooth funkt im *ISM-Band* (Industrial Scientifical Medical) zwischen den Frequenzen 2,4 GHz und 2,4835 GHz [64]. Bis zur Version 3.0 wurde hauptsächlich die Übertragungsgeschwindigkeit erhöht. Der Fokus der Versionen 4 und 5 besteht dagegen auf der Reduktion des Energieverbrauch und der Implementierung von *Bluetooth Low Energy* (BLE), um auch geringe Datenmengen energiesparend zu transportieren.

Bluetooth Low Energy ist der energiesparende Übertragungsmodus von Sensordaten und wird ab Bluetooth 4.0 unterstützt. BLE ist nicht abwärtskompatibel zu älteren Bluetooth Versionen. Besonders die Version 4.2 bietet im Low Energy Modus große Verbesserungen, da die Datenrate um den Faktor 2,5 größer ist, dank der Vergrößerung der Paketkapazität um den Faktor 10 im Vergleich zu vorherigen Versionen des Low Energy Standards [65]. Mit Bluetooth 5 (2016) wird der BLE Standard weiter ausgebaut. Die Datenrate beträgt bei Bluetooth 5 immerhin zwei Drittel der Geschwindigkeit von Bluetooth 3, also eine weitere Verdopplung im Gegensatz zu Version 4.2 [66]. Alternativ kann die Datenrate von 500 auf 125 kBit pro Sekunde reduziert werden, dafür jedoch die Reichweite erhöht werden. Ab Version 5.0 darf mit 100 mW Sendeleistung, statt bisher 10 mW gesendet werden.

Bluetooth Low Energy Geräte verbinden sich immer über das *Generic Attributes Profil* (GATT) [66]. *GATT* wiederum basiert auf dem *Attribute Protocol* (ATT). Dieses Profil ist für die Übertragung von Sensordaten gedacht und ist daher auf energiesparende Übertragung geringer Datenmengen optimiert. Es stehen verschiedene Profile zur Verfügung, die für verschiedene Anwendungsfälle geschaffen wurden und in den Bluetooth Spezifikationen definiert sind [67]. *GATT* besitzt eine hierarchische Datenstruktur. Das Profil entspricht der höchsten Ebene in dieser Hierarchie. Darunter befindet

sich mindestens ein *Service*, der wiederum aus *Characteristics* aufgebaut ist. Eine *Characteristic* definiert die Eigenschaften und das Verhalten des Geräts. Eine *Characteristic* besteht aus einem Typ (eine *UUID*), einem Wert und mehreren Properties, also Eigenschaften. Darüber hinaus kann eine Beschreibung mittels Metadata vorhanden sein. *GATT* verwaltet die einzelnen *Services* und übernimmt das Lesen, Schreiben, die Benachrichtigung und die Sichtbarkeit der einzelnen *Services*. Zusätzlich zu den Standard *Services* können auch individuelle *Services* definiert werden. Ein BLE Sensor kann dadurch verschiedene Rollen inne haben und je nach Bedarfsfall unterschiedlich ausgelesen werden.

Der Energieverbrauch eines BLE Gerätes kann reduziert werden, wenn Daten seltener übermittelt werden, oder die Datenmenge reduziert wird. Es gibt die Unterscheidung zwischen *Slave* und *Master*. Sensoren senden über dem *Advertisement Channel* in regelmäßigen Abständen kurze *Advertisement Events* (Aufmerksamkeitshinweise). Das Gerät, das auf diese reagiert ist der *Master* und der Sender dieser Nachrichten ist der *Slave*. Der Verbindungsaufbau kann nun initiiert werden.

Kapitel 4

4 Konzept

Mit den Erkenntnissen über das bestehende System aus dem Kapitel 2 wird nun ein Konzept entwickelt, das auf Anforderungen, die an die App und deren Entwicklung gestellt sind, fußt. Technisch wird die Art und Weise betrachtet, wie die App aufgebaut und strukturiert werden kann und welche Auswirkungen die Wahl des Design Patterns hat. Um die Anforderungen für die App zu formulieren, werden in Kapitel 4.1.1 Personen identifiziert, die direkt oder indirekt mit dem Produkt in Verbindung stehen. In Kapitel 4.2 wird für das gewünschte System ein Aufbau geplant. In Kapitel 4.3 wird das Design Pattern der App näher beschrieben, das sich für diesen Anwendungsfall eignet.

4.1 Anforderungsanalyse

Bei der Entwicklung einer Anwendung oder eines Systems ist es wichtig, dass der Anwendungsfall und die Anforderungen an ein System klar definiert und bekannt sind. Die Personen, die Interesse an der Arbeit haben, werden in Kapitel 4.1.1 beschrieben. Darüber hinaus werden anschließend weitere Anforderungen an die App formuliert, die nicht unmittelbar gefordert sind, aber der App zuträglich sind.

4.1.1 Anforderungen von Stakeholdern

Stakeholder sind Personen, oder Organisationen, die ein aktives Interesse am Produkt, oder Projekt haben. Sie kommen entweder direkt, oder indirekt damit in Kontakt. Stakeholder benutzen, entwickeln, betreiben, warten, testen, managen oder vermarkten das Produkt. Da das Produkt nicht isoliert betrachtet werden kann, sind auch die damit verknüpften Systeme relevant. Sollte es Regulatorien, oder Stakeholder geben, die negativ vom System oder Produkt beeinflusst werden, so sind diese *Negative Stakeholder* [68].

Um die Interessen der verschiedenen Stakeholder des Demonstrators zu erfassen, wurden Interviews mit Mitarbeitern bei PolyIC geführt und Anforderungen zusammengefasst. Dabei wurden sowohl Mitarbeiter aus dem Vertrieb, als auch von Seiten der technischen Entwicklung und Fertigung befragt. Hierbei stellte sich heraus, dass der Großteil der Anforderungen von zwei Personengruppen vorgegeben werden: Geschulte Vertriebsmitarbeiter und der Kunden, dem das System präsentiert werden soll.

Benutzer Der Benutzer ist in den meisten Fällen ein Vertriebsmitarbeiter der Firmen PolyIC oder KURZ. Der Benutzer möchte die bestehende Windows-Software durch eine App ersetzen. Im Vertrieb werden überwiegend Smartphones, oder Tablets mit iOS Betriebssystem eingesetzt. Der Benutzer darf jedoch auch Android Geräte verwenden.

Die UI und UX (*User* Experience, engl. für die Nutzererfahrung) der Software sind dem Benutzer sehr wichtig, da die App bei der Vorführung direkt mit einem potentiellen Kunden in Kontakt kommt. Mithilfe der App sollen die Fähigkeiten des Sensors, dem eigentlichen Produkt, überzeugend präsentiert werden. Eine hohe Geschwindigkeit der App ist wichtig. Die App soll sich schnell öffnen und mit dem Demonstrator verbinden lassen. Die Navigation in der App soll intuitiv und unkompliziert gestaltet sein.

Der Benutzer wünscht sich einen kabellosen Demonstrator, die genaue Implementierung ist hierbei unwichtig. Das Einsatzgebiet des Demonstrators ist vorwiegend beim Kunden, oder in firmeneigenen Räumen. Je nach Zuverlässigkeit der Datenübertragung ist ein Betrieb auf Messen erwägenswert – dies ist jedoch nicht das Haupteinsatzgebiet. Die Verbindung mit dem Sensor soll schnell und für den Benutzer möglichst einfach realisiert werden. Da sich mehrere Geräte im Umkreis befinden können, sollen dem Benutzer nur die Geräte angezeigt werden, mit denen die App kompatibel ist. Die Auswahl soll über ein Auswahlmenü erfolgen, bei dem das Gerät eindeutig identifiziert werden kann. Daher sollen die Geräte beschriftet und nummeriert werden.

Berührungen auf dem Sensor sollen auf das Smartphone übertragen und dort angezeigt werden. Damit die Funktionalität der PC-Software beschrieben (siehe Kapitel 2.4). Dabei soll jede Berührung einem Finger zugeordnet werden können. Die Darstellung soll aufgrund des kleinen Sensors auf fünf Finger begrenzt werden. Für technisch versierte Benutzer, meist die Personen, die an der Entwicklung des Demonstrators bei PolyIC beteiligt sind, soll ein Debug-Modus implementiert sein, der vom Sensor gelieferte Messwerte auf dem Smartphone übersichtlich chronologisch darstellt.

Die App soll über den internen App Catalog von KURZ verteilt werden. Dort werden sämtliche Apps veröffentlicht, auf die Mitarbeiter der KURZ-Gruppe Zugriff haben sollen. Dieser App Catalog ist mit Zugangsdaten vor Fremdzugriff geschützt und je nach Konfiguration durch den Administrator, stehen unterschiedliche Apps zum Download bereit. Dem App Catalog müssen auf dem lokalen Endgerät Rechte gegeben werden, dass auf diesem Apps installiert werden können. Nicht ganz so wichtig ist die Einstellung in die App Stores der beiden Betriebssysteme. Die App soll Hoch- bzw. Querformat unterstützen.

Kunde Dem Kunden wird die App vom Benutzer vorgeführt. Der Kunde hat somit die Anforderung, dass die App übersichtlich gestaltet ist, sodass sie nicht verwirrt. Vertriebsmitarbeiter berichten davon, dass der Kunde im ersten Moment Probleme mit der Auge-Hand-Koordination hat und die Fingerberührungen auf dem Sensor nicht direkt mit einer Farbe der Kreise, die Berührungen in der Anwendung symbolisieren, verknüpft. Der Kunde wünscht eine Nummerierung der in der

App dargestellten Berührungen der Finger. Berührt der Kunde den Sensor mit dem eigenen Finger und bewegt diesen, so fallen diesem leicht Verzögerungen zwischen Bewegung des Fingers und Visualisierung auf. Diese Latenz soll so gering wie möglich sein. Auch die Bildwiederholrate der App selbst kann entscheidend für einen positiven oder negativen Eindruck sein. Es ergeben sich somit mehrere Anforderungen, die in Tabelle 4.1 zusammengefasst sind.

1	Geringe Latenz zwischen Berührung auf dem Sensor und Anzeige auf dem Bildschirm
2	Es sollen Berührungen von fünf Fingern angezeigt werden (in der App nummeriert)
3	Die App soll eine schnelle Startzeit aufweisen
4	Werden mehrere Bluetooth-Geräte gefunden, soll ein Auswahlmenü angezeigt werden
5	Gleicher Funktionsumfang bei Android und iOS
6	Ein Debug Bildschirm zur Anzeige der Touch-Daten
7	Intuitive Bedienung
8	Optisch ansprechende App
9	App soll eine hohe Performance besitzen
10	Die App soll im KURZ internen App Catalog verfügbar sein
11	Akku-Betrieb des Demonstrator möglich
12	App soll Hoch-und Querformat unterstützen

Tab. 4.1: Anforderungen an die App und den Demonstrator

Nicht vorgegebene Anforderungen

Die bisher formulierten Anforderungen kommen Anforderungen hinzu, die nicht formuliert oder vorgegeben sind, aber die Bedienung erleichtern oder angenehmer gestalten. Da die App nur bei aktivem Bluetooth funktioniert, soll bei deaktiviertem Bluetooth eine Warnung oder Mitteilung ausgegeben werden. Wird bei der Suche kein Gerät gefunden, so soll dem Benutzer dies angezeigt werden. Auch das erfolgreiche Verbinden mit dem Demonstrator soll eine Rückmeldung zur Folge haben. Wird die Bluetooth-Verbindung zur Hardware getrennt, so soll die App in den Ausgangszustand übergehen, also nicht verbunden, aber bereit, um eine neue Suche nach verfügbaren Geräten zu starten. Bei Unterbrechung der Verbindung mit der Hardware soll die App automatisch wieder zurück in den Zustand „nicht verbunden“ übergehen und dem Benutzer wieder die Option geben, ein Gerät zu suchen und sich zu verbinden. Die Debug-Ansicht, also die Liste der übermittelten Messwerte, soll nur dem geschulten Nutzer bereitstehen und sich durch eine Geste aktivieren lassen.

4.1.2 Eignungskriterien für das SDK

Bei der Erwägung eines Cross-Plattform-SDKs ist natürlich von Interesse was bei der Entwicklung für beide Betriebssysteme bedacht werden muss und wie gut darauf reagiert werden kann. Gerade bei einem relativ jungem SDK wie Flutter, ist es wichtig, dass für Einsteiger eine gute Dokumentation vorhanden ist und ein aktiver Wissensaustausch in der Entwicklergemeinde stattfindet. Im Google Developer Blog wurde daher, selbst vor der Veröffentlichung von Flutter Version 1.0 von einer starken Nachfrage auf der Plattform *StackOverflow* zum Thema Flutter berichtet, wie die Statistik in Abbildung 4.1 zeigt [5].

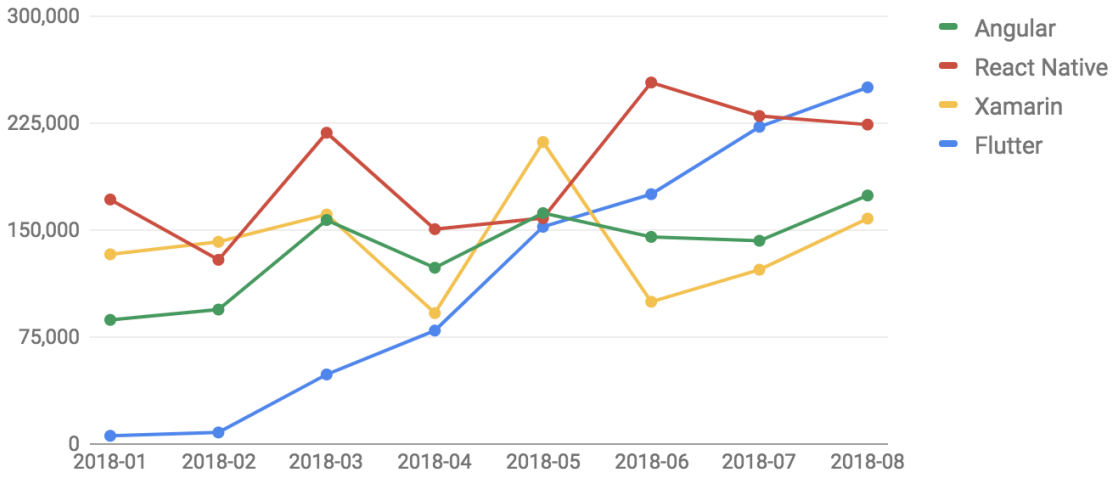

Abb. 4.1: Anzahl der StackOverflow Fragen zu vier populären UI Frameworks [5].

Es gibt einen stillschweigenden Konsens bei der Entwicklung von Erweiterungen für Flutter, dass eine Funktionalität nicht mehrmals entwickelt, sondern nur einmal, mit öffentlich zugänglichem Quellcode, gemeinsam entwickelt wird. Dabei wird bei jedem Flutter *package* von *Pub* ein Score, also ein Wert berechnet, der dem Entwickler einen Indiz dafür gibt, wie gut ein *package* gepflegt und aktualisiert wird. In Abbildung 4.2 ist die Bewertung von dem *flutter_flare package* dargestellt. Viele Faktoren spielen in die Berechnung des Score ein. In dem *flutter_flare package* ist unter anderem die Beschreibung zu kurz und kein Beispiel angegeben, was zu einem geringeren Score führt.

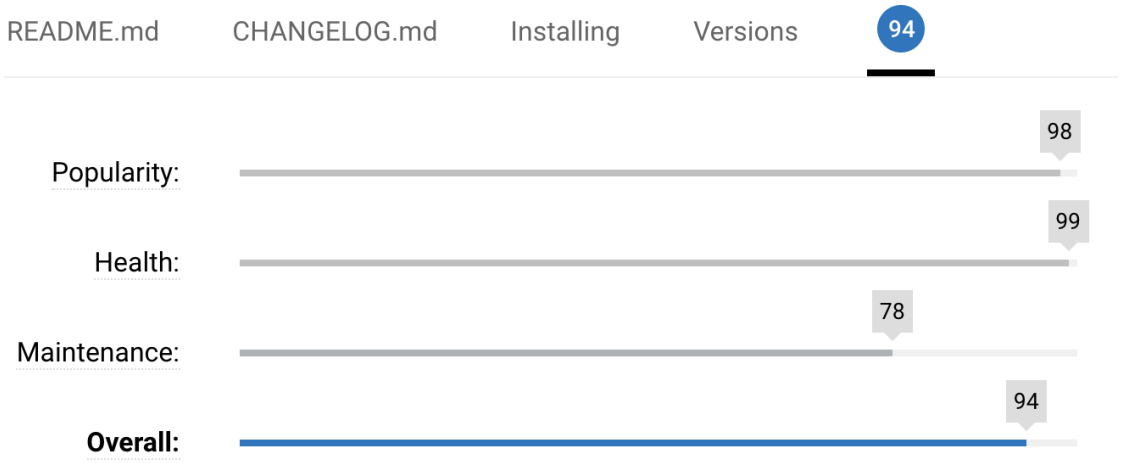

Abb. 4.2: Bewertung eines Flutter package mit einem Score auf Dart Pub.

4.2 Architektur

Mit den Ergebnissen der Anforderungsanalyse aus Kapitel 4.1 kann nun die gewünschte Architektur des gesamten Systems definiert werden. Diese ist in Abbildung 4.3 dargestellt. Von der bisherigen Architektur (siehe Abbildung 2.1) wird das FTDI-Modul durch ein Bluetooth-Modul ersetzt, um die Daten nun per Bluetooth zu versenden. Der PC und die Software sind durch ein mobiles Endgerät ersetzt.

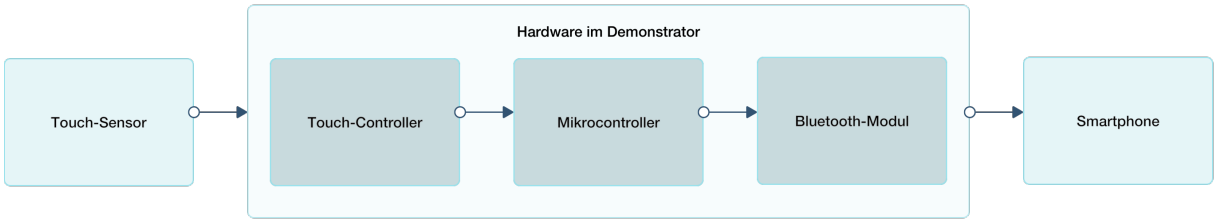

Abb. 4.3: Ziel-Architektur des Demonstrators

Das bestehende Format der Daten, die vom Demonstrator geliefert werden, ist in Tabelle 4.2 dargestellt. Der Event Typ entspricht der Information, ob der Sensor eine neue Berührung (Finger Down), eine Änderung der Koordinaten eines Fingers (Finger Move) oder das Entfernen eines Fingers (Finger Up) detektiert hat und ist mit einer Ziffer vorgesehen. Die Finger Nummer ermöglicht die Verfolgung des Fingers auf dem Bildschirm. Die X- und Y-Koordinate bestehen jeweils aus einer Zahl zwischen 0 und 1024. Die restliche Hardware des Demonstrators (Touch-Controller und Mikrocontroller) werden im Anschluss an diese Arbeit bei PolyIC überarbeitet. Das Touch-Daten-Format soll dann neu definiert werden.

Die Anbindung des Bluetooth-Moduls soll mit UART erfolgen, da die Ausgabe der Daten vom

Mikrocontroller an das FTDI-Modul mit dieser Schnittstelle erfolgt und bereits im Code implementiert ist.

Event Typ	Finger Nummer	X-Koordinate	Y-Koordinate	Y-Koordinate
(	|	|	|	|)

Tab. 4.2: Datenstruktur der Messwerte

App Entwurf

Um die Anforderungen aus Kapitel 4.1 umzusetzen, wird nun die optische Gestaltung der App und das Navigationssystem entworfen. Als Programm dient hierbei der Vektor-Grafik-Editor Sketch. Sketch wird bereits intensiv firmenintern eingesetzt. Der erste Bildschirm zeigt ein Abbild des Sensors (Abbildung 4.4), der mit der App verbunden werden kann. Eine Schaltfläche soll die Suche nach einem Demonstrator starten und die Verbindung herstellen.

Abb. 4.4: Entwurf der App-Startseite

Bei der Suche nach dem Demonstrator kann es passieren, dass sich nicht nur das gesuchte Gerät in Empfangsreichweite befindet, sondern mehrere Geräte gefunden werden. Dabei besitzen diese oft für den Endanwender kryptische oder verwirrende Namen. Ein Auswahlmenü, als Overlay, soll sinnvolle Geräte anzeigen, mit denen verbunden werden kann (siehe Abbildung 4.5).

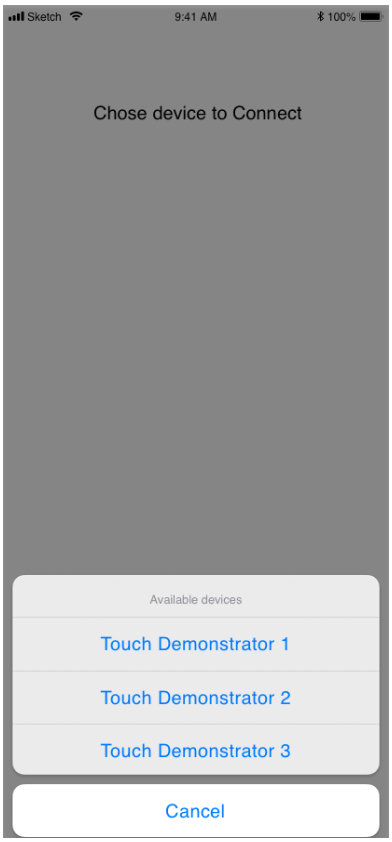

Abb. 4.5: Entwurf des Auswahl Menüs bei Verfügbarkeit mehrerer Bluetooth-Geräte

In Abbildung 4.6 ist die App mit dem Gerät namens „Touch Demonstrator 1“ gekoppelt. Ist die App mit dem Demonstrator verbunden, so sollen die Berührungen auf dem Sensor dargestellt werden. Bei Detektion einer Berührung im Bereich der drei Sensortasten auf der linken Seite des Sensors, soll in der App das Symbol farblich hervorgehoben werden (wie die Schaltfläche mit dem Haus Symbol in der Abbildung). Es soll nur eine Taste aktiv sein. Berührungen im rechten Bereich des Sensors sollen einen Schieberegler (engl. Slider) verändern. Dieser soll abhängig von der Position der Berührung einen Wert von Null bis 100 Prozent annehmen können. Die Anzeige in der App soll als Balken an der Stelle der Berührung und als Zahlenwert erfolgen. Der Schieberegler soll nach kurzer Zeit mit einer Animation ausgeblendet werden.In der Mitte des Sensors sollen Berührungen auf dem Sensor als farbliche Kreise in der App visualisiert werden. Die Farbe wird immer einem erkannten Finger zugewiesen und solange dieser sich auf dem Sensor befindet bleibt diese gleich. Zusätzlich soll im Kreis zusätzlich eine Nummer dargestellt werden, die wie die Farbe, eine Zuordnung zwischen dargestellten Kreis und Finger ermöglicht.

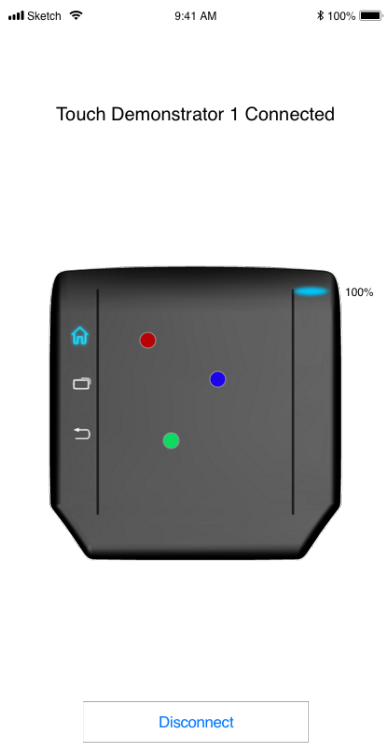

Abb. 4.6: Entwurf eines App-Screens mit dargestellten Berührungen auf dem Sensor.

Bei Flutter zeigt die UI den State der App an. Die einzelnen Widgets aus denen der Bildschirm aufgebaut ist, können daher einzeln aktualisiert werden. Jedes Bestandteil des Bildschirms kann in beliebig kleine Bestandteile unterteilt werden. Der Screen, der in Abbildung 4.6 dargestellt ist, besteht beispielsweise aus einem Button und einem Abbild des Sensors. Der Button sendet Signale an den State der App und zeigt den Verbindungsstatus an. Das Abbild des Sensors zeigt lediglich die Daten vom State an. Die Berührungen in der Mitte des Sensors können direkt auf dem canvas gezeichnet werden, oder als überlagerndes Widgets in einem `Stack` auf dem Bild des Sensors dargestellt werden. Die Farbe der Kreise kann mit dem Farbschema der App verknüpft werden, so dass dies global verändert werden kann.

4.3 Auswahl des Design Patterns der App

Bei der Entwicklung von Software wird diese in Ebenen aufgebaut, wobei jede Ebene nur soviel Rechte wie nötig besitzt. Dadurch können einzelne Komponenten einfacher getestet, ersetzt oder wiederverwendet werden. Flutter wird besonders von diesem Paradigma beeinflusst, da Widgets mehrmals verwendbar, jederzeit ersetzbar, oder durch andere Widgets konfigurierbar sein können. Je strukturierter der Code ist, der die UI beschreibt, desto mehr Code kann eventuell eingespart werden.

Jedes Widget sollte nur auf die Daten Zugriff bekommen, die für dieses von Relevanz sind. Soll beispielsweise ein Widget Informationen anzeigen, die von anderen Komponenten der App verändert werden können, so sollte der State von möglichst wenigen weiteren Widgets veränderbar sein.

Um mit mehreren Personen an einem Projekt arbeiten zu können, sollte der App Aufbau einfach verständlich sein. Ein gutes Design Pattern behindert den Programmierer nicht und schiebt sich, da wo es nur Mittel zum Zweck ist, in den Hintergrund. Code, der nur das Design Pattern stützt, ohne App-Funktionalität hinzuzufügen sollte daher minimal sein. Der Aufbau der App sollte den Zielen von Flutter, wie der guten Performance, nicht im Weg stehen.

Die Methode `setState()` teilt dem Framework mit, dass das gesamte `StatefulWidget` neu gebaut werden soll, auch wenn nur ein kleiner Bestandteil neu gebaut werden müsste. Ein `StatefulWidget` weit oben in der Hierarchie der App mit häufigen, eventuell sogar unnötigen, Aufrufen von `setState()` hat viele Berechnungen zur Folge und reduziert die Performance.

Daten werden in Flutter meist von oben nach unten im Widget Baum verteilt, also von unten nach oben hin zugegriffen. Dabei ist besonders das `InheritedWidget` zu nennen. Mit steigender Komplexität der App verliert der Programmierer jedoch schnell den Überblick, weswegen für größere Projekte oft abstraktere Modelle wie das BLoC-Pattern oder auch Redux Anwendung finden, um die Komplexität zu reduzieren.

Bei der Wahl des Design Patterns für diese App wird das BLoC-Pattern ausgewählt, da dieses von Google in Videos und Dokumentation am umfangreichsten dokumentiert ist und auch intern für größere Projekte eingesetzt wird. Viele Widgets können auf den State zugreifen, ohne diesen selbst verwalten zu müssen. Widgets können im Widget Baum beliebig verschoben werden, ohne dass geändert werden muss, wie die Daten dort hin gelangen, da dies über Streams geschieht. Im BLoC kann die Geschäftslogik ohne störende UI programmiert werden, die oft unabhängig verändert wird, wenn z.B. eine Umgestaltung der App erfolgt.

Bei Erweiterung der App, können weitere BLoCs hinzugefügt werden, ohne die bestehenden BLoCs erneut testen zu müssen (oder deren Tests zu verändern). Informationen aus mehreren BLoCs können mit dem `BlocProviderTree` Widget aus dem `flutter_bloc` *package*, gebündelt werden (siehe Listing 4.1). In dieser Arbeit wird die Implementierung ohne dieses *package* beschrieben, um näher auf die Funktionalität des Patterns einzugehen. Mit diesem *package* kann die Menge an Code weiter reduziert werden.

```
BlocProviderTree(
  blocProviders: [
    BlocProvider<BlocA>(bloc: Bloc1()),
    BlocProvider<BlocB>(bloc: Bloc2()),
  ],
  child: ChildWidget(),
)
```

Listing 4.1: Bereitstellung von Daten von mehrerer BLoCs mit einem BlocProviderTree.

Die in Dart implementierten Konzepte der reaktiven Programmierung können mit Reactive Extensions (kurz Rx) erweitert werden. Das Framework *ReactiveX* versucht Ideen des *Observer*-Patterns, des *Iterator*-Patterns und der Funktionalen Programmierung zu verbinden und ist für viele Sprachen, wie Kotlin (RxKotlin), Java (RxJava) und JavaScript (RxJS) verfügbar [69]. Das bei Dart relevante *package* `rxdart` erweitert die `Stream` Klasse mit einem `Observable`, baut also auf bestehender Funktionalität auf (`dart:async`) [70]. Das hat zur Folge, dass ein `Observable` mit APIs funktioniert, die einen `Stream` erwarten. Ein Dart `Stream` kann mit dem Konstruktor der Klasse `Observable` in ein `Observable` konvertiert werden. Ein `Observable` ist eine Variable, die eine zeitliche Information enthält. Ein `Observable` kann somit auf einem Zeitstrahl vorgestellt werden, auf dem zu bestimmten Zeitpunkten Informationen vorhanden sein können. Es kann auch periodisch der Zustand des `Observable` überprüft werden und eine weitere Verarbeitung erfolgen.

Ein Dart `Observable` ist vom Typ Single-Subscription. RxDart bietet drei Varianten des StreamControllers. Das ist einmal das `PublishSubject`, das `BehaviorSubject` und das `ReplaySubject`.

Ein `PublishSubject` ist ein normaler Broadcast `StreamController`, außer das ein `Observable` und kein `Stream` zurückgegeben wird. Ein `PublishSubject` gibt dem Listener nur Events weiter, die zeitlich ab dem Subscribe (`subscribe()`) erfolgen. Vergangene Events werden also nicht an neue Listener versendet, siehe Abbildung 4.7.

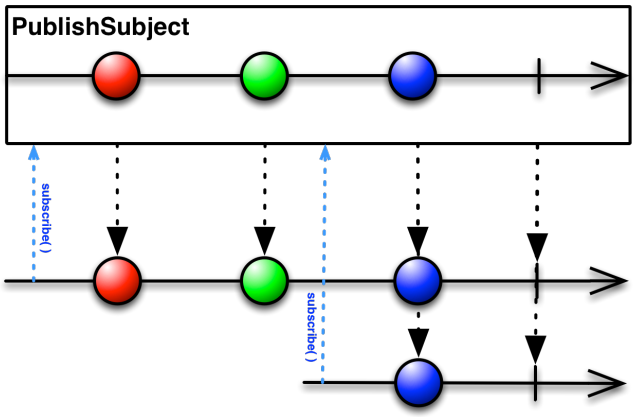

Abb. 4.7: Subscribe eines PublishSubject. Quelle: ReactiveX [6]

Das `BehaviorSubject` hingegen ist identisch zum `PublishSubject`, nur dass der letzte Event vor dem Subscribe an den Listener übermittelt wird, siehe Abbildung 4.8.

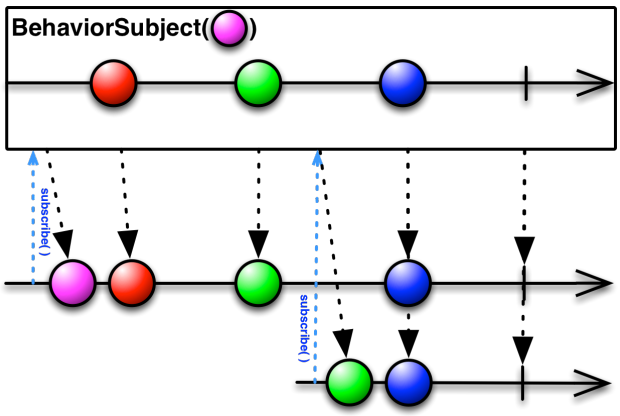

Abb. 4.8: Subscribe eines BehaviorSubject. Quelle: ReactiveX [6]

Ein `ReplaySubject` sendet wiederum alle vergangenen Events an die neuen Listeners, siehe Abbildung 4.9.

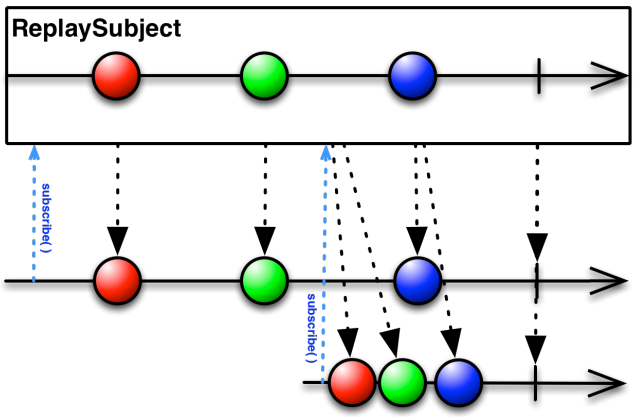

Abb. 4.9: Subscribe eines ReplaySubject. Quelle: ReactiveX [6]

Wird ein Listener nicht mehr benötigt, kann seine Ressourcen freigeben werden. Bei einem `Stream` erfolgt dies mit `cancel` und bei RxDart Subjects mit `close`.

Um auf Daten von einem `Stream` oder `Observable` in einem Flutter Widget zugreifen zu können, wird die Klasse `StreamBuilder` benutzt. Dieser hat eine `builder`-Funktionalität, die das zu bauende Widget beschreibt und jedes Mal aufgerufen wird, wenn der `Stream` Daten liefert.

Mit dem `StreamBuilder` in Listing 4.2 wird ein `Text` gebaut, der einen Zählerstand enthält. Dieses Widget kann wiederum im Widget Baum eingebaut werden. Da der `Stream` eventuell noch keinen ersten Wert geliefert hat, kann zum Bau des Widgets ein Initialwert (hier: `initialData: _initCounter`) angegeben werden, oder alternativ ein anderes Widget gebaut werden. Der `StreamBuilder` liefert asynchrone Momentaufnahmen des Streams (hier: `counterSnapshot`). Auf die Daten des `AsyncSnapshot` kann mit `counterSnapshot.data` zugegriffen werden. Liegt ein Fehler vor, so kann dies mit `counterSnapshot.error` abgefangen werden.

Bei der Deklaration des `StreamControllers` wird der Typ der zu übertragenden Daten angegeben (hier: **`int`**), um die Typ-Prüfung zu ermöglichen.

```
1  final StreamController<int> _streamController = StreamController<int>();
2  var _intCounter = 0;
3
4  StreamBuilder<int>(
5    stream: _streamController.stream,
6    initialData: _initCounter,
7    builder: (BuildContext context, AsyncSnapshot<int> counterSnapshot){
8      return Text('Counter value: ${counterSnapshot.data}');
9  }
```

Listing 4.2: Bau eines Widgets mit dem StreamBuilder um Daten eines Streams darzustellen.

Rx bietet die Option, Daten in einem `Stream` zu manipulieren. Mit `map()` wird eine Funktion auf jedes Objekt im `Stream` angewendet. Dabei kann der neue `Stream` durch Transformation einen anderen Typ aufweisen. Methoden lassen sich in Reihe schalten. Mit `where()` können die Events gefiltert werden, sodass nur auf relevante Daten reagiert wird. Mit `delay()` können Daten verzögert werden. Liefert ein `Stream` viele Daten, müsste die Benutzeroberfläche häufig aktualisiert werden und Berechnungen im Hintergrund könnten verlangsamt werden. Um dies zu vermeiden, kann mit `debounceTime()` eine Zeit eingestellt werden, die vergehen muss, in der keine weiteren Objekte dem `Stream` hinzugefügt werden. Die Methode `throttleTime()` übermittelt pro eingestellter Zeit genau ein Objekt und reduziert dadurch die Datenrate.

Eine App sollte so selten wie möglich auf Ereignisse reagieren, jedoch immer auf dem aktuellsten Stand sein. Es können auch mehrere Streams miteinander gekoppelt werden, sodass auf Kombinationen von

Ereignissen reagiert werden kann (Event bei jedem `Stream`: `zipwith()`; Event bei einem beteiligten `Stream`: `mergeWith()`).

Bei einer Flutter App greifen mehrere Widgets auf die selben Daten zu. Dies und die Tatsache, das Flutter dem Entwickler die Möglichkeit bietet für verschiedene Anwendungsfälle komplett eigenständige, oder teilweise unterschiedliche Benutzeroberflächen zu generieren, wird ein Design Pattern gesucht, das bei den Widgets so wenig wie möglich lokaler State verwendet. Daher wird für die App eine Geschäftslogik umgesetzt, die mit den Widgets per Streams verbunden ist. Die Widgets müssen somit nur noch Informationen, wie den aktuellen Stand einer lokalen Animation, die nur dieses Widget betreffen, enthalten.

Eingaben des Benutzers durch Berührungen oder Gesten gehen in die Geschäftslogik ein. Diese aktualisiert den State. Die Logik verarbeitet die Eingaben und sendet den aktualisierten State an die Widgets. Eine Rückkopplung an die Geschäftslogik gibt es nicht. Die UI kann getrennt programmiert werden, ohne dass dies den `State` beeinflusst. Eine Information kann dementsprechend einfach in unterschiedlichen Situationen dargestellt werden. Die Geschäftslogik kann in beliebig viele verschiedene Komponenten untergliedern werden und die genaue Implementierung ist nicht vorgegeben.

Bei der App verwaltet ein BLoC die per Bluetooth empfangenen Daten und ein weiterer die Aktivierung des Debug Modus. Der Debug BLoC ist relativ klein, da dieser nur ein einziges Element der App, den Zugang zum Debug Screen, aktiviert, oder deaktiviert. Die Logik soll über eine `Sink` verfügen, die einen Zähler erhöht (`debugCounterIncrement`), der beim Erreichen eines Schwellwertes den Debug Modus aktiviert und einen `Stream`, der diese Information ausgibt (`debugEnabledStatus$`). Intern sind diese mit der Logik verbunden (Listing 4.3). Sobald ein Inkrement Event erfolgt, wird der Listener vom Stream `_incrementController` aufgerufen.

```
1 final _incrementController = StreamController<void>();
2 final _debugEnabledStatus$ = BehaviorSubject<bool>();
3
4 Sink<void> get debugCounterIncrement => _incrementController.sink;
5 Stream<bool> get debugEnabledStatus$ => _debugEnabledStatus$.stream;
```

Listing 4.3: Ein- und Ausgänge der Business Logic des Debug Modus

Die zweite Logik soll die Verbindung der App über Bluetooth, mit dem Demonstrator verwalten. Der Start (`scan`)-, oder das Stoppen (`stop`), einer Bluetooth-Suche kann per Event angeregt werden. Die Verbindung mit einem Gerät kann mit `connect` der Logik mitgeteilt werden und mit `disconnect` die Verbindung getrennt werden. In der Logik selbst sollen die empfangenen Daten ausgewertet werden und jeder Messwert nach Koordinate auf dem Touchpad als ein Tastendruck (links), eine Berührung (Mitte), oder eine Veränderung des Schiebereglers (rechts) interpretiert werden.

Kapitel 5

5 Implementierung

In diesem Kapitel wird darauf eingegangen, wie das erarbeitete Konzept umgesetzt wird und welche Erkenntnisse daraus gewonnen wurden. Die Firma PolyIC hat bisher keine Apps entwickelt, oder für Projekte genutzt. Um Demonstratoren technologisch aufzuwerten, wird der in dieser Arbeit beschriebenen Demonstrator um Bluetooth erweitert. Eine Verteilung von Software über ein zentrales System, dem App Catalog, soll es ermöglichen, Updates oder Fehler zu beseitigen, auch wenn die Hardware sich schon in den Händen eines interessierten Kunden befindet, oder der Demonstrator sich auf dem Weg zu einer Messe befindet. Kunden und Mitarbeitern kann leicht Zugriff zur App gewährt werden. Durch die Bluetooth-Anbindung kann Geld eingespart werden, da kein Display zur Anzeige, oder Aussparungen im Gehäuse für Schnittstellen mehr nötig sind. Ein Smartphone genügt für mehrere Demonstratoren.

In Kapitel 5.1 wird die Umsetzung der Bluetooth Verbindung zwischen der Hardware und der Software beschrieben. Im Anschluss wird im Kapitel 5.2 die Programmierung der App mit Flutter gezeigt und wie diese sich in die bei KURZ Digital Solutions stattfindenden Entwicklungs-Prozesse einbinden lässt. In Kapitel 5.3 wird beschrieben, welche Besonderheiten Flutter im Bezug auf die Testbarkeit einer App hat.

5.1 Bluetooth

Die Firma PolyIC besitzt eine spezielle Abteilung zum Bau von Prototypen, Demonstratoren und Ausstellungsobjekten für Messen. Der direkte Ansprechpartner bei dieser Abteilung ist der Dipl.-Ing. Sean Durkin. Im Kapitel 2.1 ist der bisherige Aufbau des Demonstrators beschrieben, der nun mit Bluetooth ausgestattet werden soll. Die meisten Demonstratoren, welche bei PolyIC mit Touch-Sensoren ausgestattet werden, besitzen denselben Hardware-Aufbau, bestehend aus einem Touch-Controller und Mikrocontroller. Da sich dieser Aufbau bei kommenden Projekten ändern wird, soll die bestehende Hardware in dieser Arbeit nicht ersetzt, sondern lediglich um ein Bluetooth-Modul ergänzt werden.

Ein weiterer Aspekt, der im Rahmen dieser Arbeit nicht abgehandelt wird, ist ein Hardware-Fehler, der dazu führt, dass Datenpakete vom Mikrocontroller teilweise korrupt übertragen werden. Dieser Fehler ist wahrscheinlich auf einen Hardwaredefekt, einem Schaltungsfehler zurück zu führen. In

diesem Kapitel wird die Implementierung auf dem Bluetooth-Modul des Demonstrators und in der App behandelt.

5.1.1 Konfiguration des Bluetooth-Moduls des Demonstrators

Bisher ist der Mikrocontroller mit UART mit einem FTDI-Chip verbunden, der die Daten USB-kompatibel wandelt. Um die Architektur umzusetzen, die in Kapitel 4.2 beschrieben ist, wird ein Bluetooth-Modul im Demonstrator integriert. Die Verbindung von Mikrocontroller und FTDI-Chip wird getrennt und der Mikrocontroller direkt mit UART mit der Entwicklungsplatine *Feather M0 Bluefruit LE* der Firma *Adafruit* verbunden [71]. Das Board verfügt über einen Micro-USB-Anschluss, mit dem es an einen Computer zum Programmieren, oder an ein Netzteil zur reinen Stromversorgung, angeschlossen werden kann. Die Platine verfügt außerdem über einen Anschluss für einen Lithium-Ionen- oder Lithium-Polymer Akku, um diesen aufzuladen. Der Wechsel in den Entlade-Modus beeinflusst die Funktionsweise des Mikrocontrollers (*Cortex M0* vom Typ *ATSAMD21G18*) des Moduls nicht.

Das Adafruit Board kann mit der *Arduino* Entwicklungsumgebung programmiert werden. Das für diesen Anwendungsfall relevante Bluetooth-Modul, das auf der Platine verbaut ist, stammt von *Nordic* und ist vom Typ *nRF51822.* Dieses verfügt über das *Nordic UART RX/TX Connection Profile*, das zur Übermittlung von Daten optimiert ist [72]. Beim Start des Bluetooth-Moduls startet dieser *Service* (Base UUID: 6E400001-B5A3-F393-E0A9-E50E24DCCA9E). Der *Service* besitzt zwei *Characteristics*, einen Sendekanal TX (UUID 0x002) und einen Empfangskanal RX (UUID 0x003). Mit diesen beiden Kanälen wird eine UART Verbindung über Bluetooth simuliert. Dem Bluetooth-Modul können mit `ble.print()` Daten für den TX Kanal mitgegeben werden. Solange der Kanal offen ist (Überprüfung mit `ble.available()`) können Daten mit `ble.read()` gelesen werden. Jeder Demonstrator kann mit einem individuellen Namen versehen (z.B. „Touchpad Demonstrator 18“ in Listing 5.1) werden.

```
1 if (! ble.sendCommandCheckOK(F("AT+GAPDEVNAME=Touchpad Demonstrator
  ↪ 18"))){
2   error(F("Could not set device name?"));
3 }
```

Listing 5.1: Benennung des Bluetooth-Moduls in der Setup Funktion.

Das Bluetooth-Modul liest die Daten vom Mikrocontroller per UART-Schnittstelle aus und überprüft, ob diese Daten plausibel sind. Anschließend werden diese Daten auf dem zuvor beschriebenen TX Kanal an ein Bluetooth LE kompatibles Gerät weitergeschickt (mit `ble.print()`). Die Auswertung der Messwerte könnte bereits auf dem Mikrocontroller des Bluetooth-Moduls erfolgen (z.B. ob der

Schieberegler des Touchpads betätigt wurde) und dieser Wert per separater Characteristics an die App übertragen werden. Da die Plattform jedoch nicht final ist und der Fokus dieser Arbeit auf der Software-Lösung liegt, werden nur die Messwerte übermittelt und dort die Auswertung ausgeführt.

5.1.2 Bluetooth-Implementierung in der App

Zur Verbindung des Smartphones mit dem Bluetooth-Modul wird das Flutter *package* `FlutterBlue` eingesetzt, das von Paul DeMarco entwickelt wird [73]. Dieses *package* implementiert die Kommunikation von Bluetooth Low Energy für Android und iOS und erspart die manuelle Kommunikation mit dem Betriebssystem. Zum Vergleich: Für *Bluetooth Classic* gibt es für Flutter aktuell noch kein geeignetes *package*. Mit `FlutterBlue` kann nach Geräten, *Services*, *Characteristics* und *Descriptors* gesucht und mit diesen interagiert werden. In der App wird eine `FlutterBlue` Instanz erstellt um damit zu interagieren (Listing 5.2).

```
1 FlutterBlue _flutterBlue = FlutterBlue.instance;
```

Listing 5.2: Instanziierung von FlutterBlue.

Mit der Instanz (`_flutterBlue`) kann nach Geräten gesucht werden (Listing 5.3). Dabei ist die Dauer der Suche einstellbar. Die Geräte, die gefunden werden, werden in der `Map scanResults` gespeichert. Nachdem die Suche beendet ist, wird mit `onDone: _stopScan` die `StreamSubscription` beendet.

```
1 StreamSubscription _scanSubscription;
2 Map<DeviceIdentifier, ScanResult> scanResults = Map();
3
4 _scanSubscription = _flutterBlue
5     .scan(timeout: const Duration(seconds: 1))
6     .listen((scanResult) {
7       scanResults[scanResult.device.id] = scanResult;
8     }, onDone: _stopScan);
```

Listing 5.3: Suche nach Bluetooth Geräten mit dem package FlutterBlue.

Nun kann die Auswertung der Suchergebnisse erfolgen. Dafür wird die Funktion `findDevice()` aufgerufen. Darin werden die gefundenen Geräte überprüft, ob diese den richtigen `localName` aufweisen, den Namen, den ein Bluetooth Gerät besitzt und mit dem es sich gegenüber anderen identifiziert. Der Vorgang der „Namensgebung“ ist in Kapitel 5.1.1 dargestellt. Wird kein einziges

Gerät gefunden, also `devicesFound == 0`, wird ein Event erzeugt, um den Nutzer zu informieren. Wird genau ein Gerät gefunden, so wird direkt `_connect(deviceFound)` aufgerufen, um sich mit diesem zu verbinden. Sind mehrere Demonstratoren in Reichweite, so wird ein Event erzeugt, der alle Suchergebnisse zur Visualisierung bereit stellt. In Listing 5.4 ist dargestellt, wie die gefundenen Geräte durchsucht werden. Dabei wird der lokale Name `localName` aus den `advertisementData` jedes Eintrags auf Übereinstimmung mit dem festgelegten Namen überprüft.

```
1  var devicesFound = 0;
2  BluetoothDevice deviceFound;
3
4  scanResults.forEach((k, v) {
5    if (v.advertisementData.localName.contains(deviceNameSearched)) {
6      devicesFound++;
7      deviceFound = v.device;
8    }
9  });
```

Listing 5.4: Auswertung der gefundenen Bluetooth Geräte mit dem package flutterBlue.

Mit der Methode `connect()` kann die Verbindung mit einem Gerät (`device`) hergestellt werden (siehe Abbildung 5.5). Hierbei ist `deviceConnection` eine `Streamsubscription`, die bei einem Disconnect, oder dem Schließen der App (Aufruf von `dispose()`), auf **`null`** gesetzt wird. Wird dies nicht aufgerufen, weiß die App nicht, das die Verbindung nicht mehr besteht.

```
1  var deviceConnection = _flutterBlue
2    .connect(device, timeout: const Duration(seconds: 2))
3    .listen((s) {
4      // react to connection success/error and listen to services
5  });
```

Listing 5.5: FlutterBlue: Verbinden mit einem Bluetooth Gerät

Nach der Verbindung wird nach dem gewünschten Service und Characteristic gesucht. Der Bluetooth BLoC sendet die Daten vom Bluetooth-Modul verarbeitet weiter. Wird beispielsweise auf dem Sensor eine Taste betätigt, so wird die Berührung übermittelt, im BLoC als Tastenbetätigung erkannt und anschließend per `Stream buttonState$` ausgegeben. Analog geschieht das mit dem Bereich rechts, dem Slider (`slider$`) und normalen Berührungen auf dem Sensor (`singleTouchMessage$`).

5.2 App-Programmierung

Flutter gibt dem Entwickler große Freiheit bei der Gestaltung und der Organisation des Codes. Es gibt verschiedene Wege, ein und dasselbe Problem zu lösen. Flutter wählt den Leitsatz *Composition over Inheritance* (Komposition statt Vererbung), um Komponenten in viele kleine, möglichst leicht verständliche Bausteine aufzuteilen [2]. Viele Widgets sind wiederum aus anderen Widgets aufgebaut, die sich möglichst oft, möglichst einfach miteinander kombinieren lassen sollen.

Da jedoch jegliche Funktionalität und Aussehen im Code geschrieben wird, kann es leicht passieren, dass Code unübersichtlich wird. Flutter selbst nimmt dem Entwickler viel Arbeit ab, indem viele UI-Komponenten bereits fertig entwickelt sind und nur nach den individuellen Anforderungen angepasst werden können. Eine App kann so auf den ersten Blick sehr übersichtlich programmiert werden, mit wenigen „high-level“ Widgets, die wiederum ihrerseits komplex aufgebaut sein können. Der Entwickler kann bei Bedarf diese Widgets und deren exakten Aufbau einsehen. Sollte die Übersicht jedoch trotzdem verloren gehen, hilft die Benutzung des Werkzeugs *Flutter Inspector*, das im Kapitel 5.2.1 vorgestellt wird. Anschließend wird in Kapitel 5.2.2 die Icons für iOS und Android erstellt. Anschließend wird in Kapitel 5.2.3 beschrieben, welche Möglichkeiten Flutter zur Kommunikation mit dem Benutzer besitzt und darauf in Kapitel 5.2.4 die Continuous Integration.

5.2.1 Das Werkzeug Flutter Inspector

Der *Flutter Inspector* bezeichnet eine Sammlung von Tools zum Debuggen einer Flutter-App. In der Entwicklung einer Flutter-App spielt der *Flutter Inspector* eine große Rolle, um Layouts (und damit verbundene Probleme) zu verstehen und eine Übersicht zu behalten. Hierfür wird *Flutter Outline* benutzt, das Teil des *Flutter Inspectors* ist. Als Beispiel sei ein `Text` Widget genannt, das angezeigt werden soll, wenn keine Verbindung zur Hardware besteht. Dieses Widget ist in Listing 5.6 zu sehen.

```
Center(
  child: Container(
    padding: EdgeInsets.all(15.0),
    decoration: BoxDecoration(
      color: Colors.black45,
      shape: BoxShape.rectangle,
      borderRadius: BorderRadius.circular(8.0),
    ),
    child: Text(
      'Not connected to a device',
      style: TextStyle(color: Colors.white),
))),
```

Listing 5.6: Darstellung eines Flutter Widget Baums.

In Abbildung 5.1 ist die Ansicht im *Outline* Tool dargestellt, das direkt in der Entwicklungsumgebung integriert ist. Darin kann der Aufbau des Widgets und dessen Parameter betrachtet werden: Ein `Text` Widget in einem `Container`, der mit `Center` zentriert ist. Der Ort des Cursors im Editor-Fenster markiert in *Flutter Outline* das jeweilige Widget, sodass nicht nach dem Ort im Widget Baum gesucht werden muss. Es kann auch in *Flutter Outline* ein Widget markiert werden, um im Editor an die jeweilige Stelle zu springen. Das markierte Widget kann auch mit einem Knopfdruck über die GUI in ein anderes Widget, wie ein `Padding`, `Center`, `Row` oder eine `Column` eingebettet werden. Das Widget und somit alle Widgets darunter können auch als Methode extrahiert werden, um es mit einem Aufruf an dieser Stelle einbetten zu können. Das Widget kann auch im jeweiligen Kontext nach oben oder unten bewegt, oder entfernt werden.

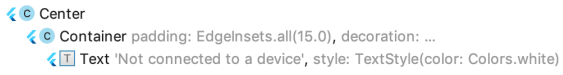

Abb. 5.1: Übersicht im Widget Tree kann mit dem Flutter Outline Tool gewonnen werden.

Der Debug Modus, um auf die Ansicht der Rohdaten des Sensors zu kommen, ist mit einem `IconButton` zugänglich. Dieses Widget wird über einen `Stream` aktiviert oder wird nicht dargestellt. Die Änderung des Widget Baums sieht man im Vergleich von Abbildung 5.2 zu 5.3. Dabei ist der Widget Baum (von oben herab) bis zum `Scaffold` und den darunter liegenden Widgets identisch. Ein `StreamBuilder` baut entweder einen `IconButton`, oder einen `Container`.

Mit dem *Select Widget Mode* des *Inspectors* kann ein einzelnes Widget im Baum lokalisiert werden. Eine Berührung des Widgets in der App hat eine Markierung des Ortes im Baum in der Programmier-

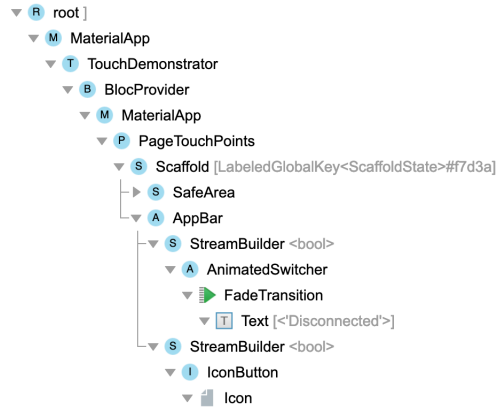

Abb. 5.2: Ein Widget Baum mit angezeigtem Debug Icon in der AppBar.

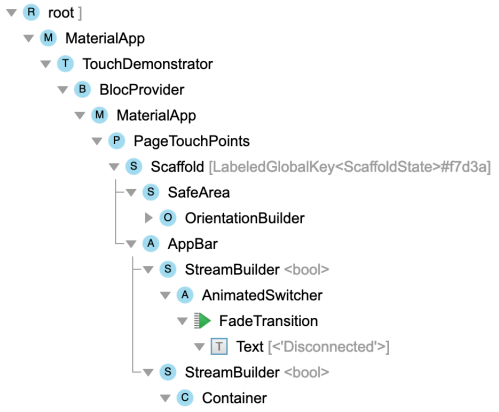

Abb. 5.3: Ein Widget Baum ohne ein Debug Icon in der AppBar.

umgebung zur Folge. Sollte beim Entwickeln ein Fehler bei einem Widget gefunden werden, kann gezielt an diese Stelle gesprungen werden, um diesen zu beseitigen.

Die nächste Funktion des *Inspectors* ist das sogenannte *Performance Overlay*, dass die Performance der App als Overlay im oberen Teil der App auf dem Entwicklungsgerät oder Simulator darstellt. Das Overlay zeigt hierbei zwei Graphen an: Ein GPU-Thread und ein UI-Thread (siehe Abbildung 5.4). Im Graph entspricht die X-Achse der zeitlichen Ebene und die Y-Achse entspricht der Zeit, die pro Frame benötigt wurde. Bei Performance-Problemen ist der Entwickler nicht auf sein subjektives Empfinden angewiesen, sondern kann dies objektiv im Normalbetrieb überprüfen. Die Performance der App kann sich jedoch vom Debug-Modus zum Release-Modus signifikant unterscheiden, weshalb Performance-Tests im Debug-Modus missverständlich sein können. Bei Benutzung dieses Performance Overlays sollte die App am besten im *Profile Mode* ausgeführt werden, bei der die Performance mit der fertigen App vergleichbar ist, aber zusätzlich noch Debug-Möglichkeiten bestehen.

Abb. 5.4: Flutter Performance Overlay

Da Flutter Cross-Plattform Produktionen ermöglicht ist es besonders interessant, dass auf einem Entwicklungsgerät auch die UI für das jeweilig andere Betriebssystem dargestellt werden kann. Das Umschalten erfolgt mit dem *Inspector* Werkzeug *Toggle Platform*. Es ist daher möglich die Android UI auf einem iPhone zu testen, ohne den Code anzupassen. Eine weitere Funktion zeigt in der App die Dimension eines Widgets an (*Show Debug Paint*). In Abbildung 5.5 ist in der `AppBar` beispielsweise der Verbindungszustand mit dem Demonstrator dargestellt. Die Dimension des Widgets ist mit einer farblichen Umrandung dargestellt. Aktuell daher, da das Widget bei einem neuen Build-Vorgang eine neue Dimension erhalten kann, siehe Abbildung 5.6 mit dem neuen Text „Connected“, statt „Disconnected“. Zusätzlich zu den Dimensionen kann sich auch der Abstand zu anderen Widgets ändern, der eingehalten werden muss (blaue Flächen in den Abbildungen).

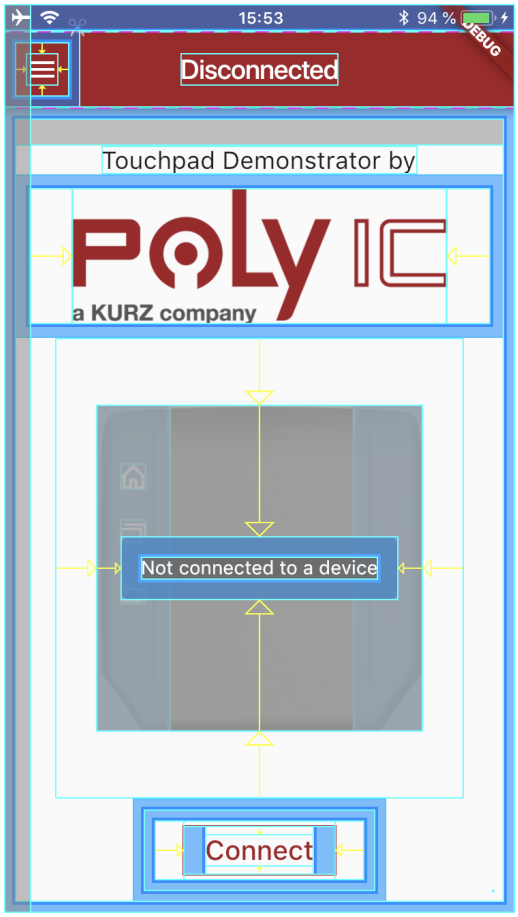

Abb. 5.5: Show Debug Paint, bei Zustand verbunden.

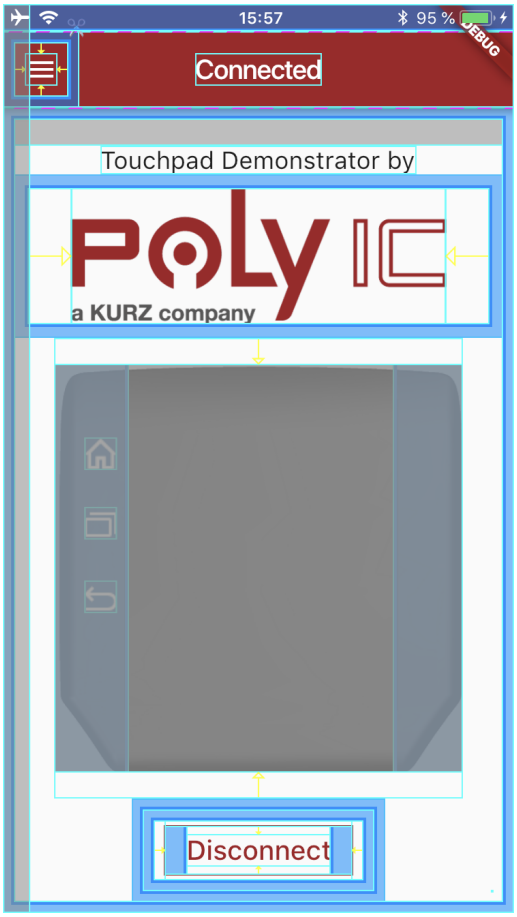

Abb. 5.6: Show Debug Paint, bei Zustand nicht verbunden.

Mit *Show Paint Baselines* kann unter Texten und Icons die „Grundlinie“ dargestellt werden. Mit der Funktion *Slow Animations* können Animationen in der App verlangsamt dargestellt werden, um den Ablauf von Animationen zu testen, die normalerweise zu schnell ablaufen würden.

5.2.2 Erzeugen von App Icons für iOS und Android

Ein Icon kann nicht für iOS und Android verwendet werden, sondern muss in unterschiedlichen Formaten vorliegen [9, 10]. Um ein Icon für beide Betriebssysteme zu generieren, kann ein *package* `flutter_launcher_icons` verwendet werden. Bei der Einbindung des *package* wird der Pfand zum Bild angegeben (siehe Listing 5.7).

```
flutter_icons:
  android: "launcher_icon"
  ios: true
  image_path: "assets/icon/icon.png"
```

Listing 5.7: App Icons dependencies in Pubspec.yaml

Mit `flutter pub get` werden alle Abhängigkeiten in das Verzeichnis geladen und anschließend die Icons generiert (siehe Listing 5.8) [74].

```
flutter pub pub run flutter_launcher_icons:main
```

Listing 5.8: App Icons in Terminal generieren

5.2.3 Interaktion mit dem Nutzer

Flutter gibt dem Programmierer mehrere Möglichkeiten den Nutzer über Ereignisse in der App zu informieren. Beispielsweise eine `SnackBar` ist ein Balken am unteren Rand des Bildschirms, der eine Information anzeigt. Benutzt wird die `SnackBar` in der App zur Darstellung folgender Zustände: Das Bluetooth-Modul ist nicht aktiv, die Suche nach Geräten erbrachte kein geeignetes Resultat oder es konnte die Verbindung nicht hergestellt werden. Diese `SnackBar` ist bei Flutter ebenfalls ein Widget und kann stark individualisiert werden. Eine kurze Einblendung versorgt den Nutzer mit den nötigen Informationen, ohne diesen lange zu stören. In Listing 5.9 ist die Methode `_showSnackBar()` dargestellt, die mit den gewünschten Inhalt aufgerufen wird.

```
_showSnackbar(String textInSnackBar) {
  /// Shows Snackbar for 0.5 s
  print('snackbar $textInSnackbar');
  mScaffoldState.currentState.showSnackBar(SnackBar(
    content: Text(
      textInSnackbar,
      style: TextStyle(fontSize: 15),
    ),
  duration: Duration(milliseconds: 500),
  ));
}
```

Listing 5.9: SnackBar Methode, um Informationen kurz einzublenden.

Statt nur einem `Text` Widget kann als Inhalt natürlich auch ein anderes Widget verwendet werden. Um z.B. eine Warnung auszugeben, dass Bluetooth nicht aktiviert ist, kann die `Snackbar` mit einem anderen Hintergrund ausgestattet werden (`backgroundColor: Colors.red`) und mehrere Informationen in einer `Row` dargestellt werden. Beispielsweise ein Symbol (`Icon(Icons.warning)`) und ein `Text`.

Die Auslösung dieser Nachrichten erfolgt durch Streams. Wird der Debug-Modus aktiviert, oder deaktiviert, so kann in der App eine `Snackbar` ausgegeben werden (vom `Stream` aus dem BLoC `debugBloc`, siehe Abbildung 5.10). In der Methode `_toggleDebug()` wird die `SnackBar` mit dem Text erzeugt.

```
1  debugBloc.debugEnabledStatus$.listen((data) => _toggleDebug(data));
2
3  _toggleDebug(bool enableDebug) {
4    _showSnackbar('Debug View: ${enableDebug ? 'enabled' : 'disabled'}!');
5  }
```

Listing 5.10: SnackBar Aufruf bei Aktivierung oder Deaktivierung des Debug-Modus.

Ebenfalls mit einem `Stream` gesteuert ist die Anzeige des Verbindungsstatus `AppBar` der App mit dem Demonstrator dem Nutzer mitteilt (siehe Abbildungen 5.5 und 5.6). Die Daten stammen von dem `Stream bBloc.isConnected$` (Bluetooth BLoC). Zuerst wird der Variable `appBarText`, siehe Abbildung 5.11, ein mintinlinedartString zugewiesen. Der `StreamBuilder` erzeugt das neue Widget und platziert das `Text` Widget in einem `AnimatedSwitcher`. Der Typ der Animation (`FadeTransition`) erzeugt bei Änderung des `child` Widgets eine (Fade-) Animation vom alten zum neuen Widget, womit ein weicher Übergang zum neuen Widget erfolgt. Das `child` Widget ändert sich, wenn der `StreamBuilder` einen Event vom `Stream` erhält, also bei einer Änderung vom Verbindungsstatus. Die Zeitdauer der Animation kann verändert werden.

```
StreamBuilder<bool> titleAppBar() {
  final bBloc = BlocProvider.of(context).bluetoothBlocGetter;
  var appBarText;
  return StreamBuilder<bool>(
    stream: bBloc.isConnected$,
    initialData: false,
    builder: (BuildContext context, AsyncSnapshot<bool> snapshot) {
      if (snapshot.hasData) {
        snapshot.data
        ? appBarText = 'Connected'
        : appBarText = 'Disconnected';

        return AnimatedSwitcher(
          duration: Duration(milliseconds: 300),
          transitionBuilder: (Widget child, Animation<double> opacity) {
            return FadeTransition(child: child, opacity: opacity);
          },
          child: Text('$appBarText'),
        );
      }
    });
}
```

Listing 5.11: Konfiguration eines Texts in der AppBar mit einem Stream.

Das selbst erstellte Widget `ConnectButtonWithAnimations` erzeugt entweder eine Schaltfläche (`OutlineButton`), zum Herstellen oder Trennen der Verbindung oder zeigt eine Animation (Widget `ScanAnimation`), die während der Bluetooth Suche angezeigt werden soll. Die Auswahl ist in Listing 5.7 dargestellt. Eine Momentaufnahme der Animation des Widgets `ScanAnimation` ist in Abbildung 5.8 dargestellt.

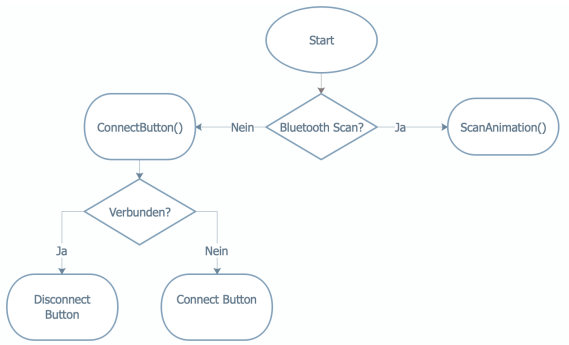

Abb. 5.7: Stream gesteuerte Widget-Auswahl in einer Build Funktion.

Abb. 5.8: Animation, die während der Geräte-Suche angezeigt wird.

Da bei Dart einer Variablen auch eine Funktion zugewiesen werden kann, kann in der Klasse `ConnectButton()` (siehe Listing 5.12) die Funktion `_buttonFunction` und auch sonstige Eigenschaften des `OutlineButton` zugewiesen werden, wie z.B. der `Text buttonText`. Ist die Verbindung hergestellt, wird die Taste so konfiguriert, dass ein Disconnect-Event (`disconnect.add(null)`) an den Bluetooth BLoC (`bBloc`) gesendet wird. Ist die Verbindung getrennt, wird die Schaltfläche umfunktioniert, so dass mit `scan.add(null)` ein Scan-Event geschickt wird. Diese Zuweisung erfolgt bei der Aktualisierung des Widgets.

```
if (snapshotIsConnected.data) {
  _buttonText = Text("Disconnect", style: _buttonTextStyle);
  _buttonFunction = () => bBloc.disconnect.add(null);
} else {
  _buttonText = Text("Connect", style: _buttonTextStyle);
  _buttonFunction = () => bBloc.scan.add(null);
}
```

Listing 5.12: Text und Funktionsbelegung eines OutlineButtons mit den Daten aus einem Stream.

Der `OutlineButton` wird anschließend konfiguriert gebaut (Listing 5.13). Die **`return`** Funktion des Widgets kann so übersichtlich programmiert werden und bei Bedarf kann die Konfiguration jeder Eigenschaft des Widgets kontrolliert werden.

```
OutlineButton(
  borderSide: BorderSide(color: _primaryColor),
  shape: BeveledRectangleBorder(),
  onPressed: _buttonFunction,
    child: _buttonText,
),
```

Listing 5.13: Tastendrücke an den BLoC übermitteln.

Hierbei entscheidet der Anwendungsfall, ob ein Widget parametrisiert werden sollte oder ob das Widget nicht komplett durch ein anderes ersetzt werden sollte, beispielsweise zwei getrennte Button Widgets. Hierbei ist die Wahl auf ein Widget gefallen, da die Schaltfläche prinzipiell das selbe Aussehen besitzt und sich nur durch die Beschriftung und die Schnittstelle an den BLoC unterscheidet.

Das Widget `ScanAnimation` setzt eine Animation aus dem Programm *Flare* (von *2Dimensions)* ein. Diese können relativ schnell erstellt werden und in einem Widget Baum bei Flutter eingepflegt werden. 2Dimensions ist Teil des *Open Design Movement*, dessen Ziel es ist, den Design-Prozess offenzulegen und nicht nur das fertige Resultat zur Verfügung zu stellen[75, 76]. Solange man die Animationen nicht privat halten möchte, fallen keine Gebühren für Flare an. Diese Animationen werden in einer Datenbank gespeichert, auf die jeder Nutzer Zugriff hat und für eigene Arbeiten modifizieren kann, wie es auch für Flutter *packages* üblich ist. Da Flutter großen Wert auf Animationen legt, kann mit `FlareActor` ein Widget erzeugt werden, das eine Flare Animation anzeigt.

Flare Animationen lassen sich auch mit in Flutter erstellten Animationen kombiniert werden. Mit `AnimationController` kann somit die Helligkeit einer Flare Animation verändert werden, oder auch die Animation selbst örtlich in der App verschoben werden. Mehrere Animationen können aneinander

gereiht, oder parallel abgespielt werden. Dies kann entweder mit einem `Tween` passieren (Veränderung eines Variablenwerts, über eine gewisse Zeit hinweg, von einem Wert zu einem anderen), oder durch Anordnung eines animierten Widgets in einem anderen erfolgen. Im Widget `Sensor` wird die Größe des Sensors durch einen `AnimatedContainer` verändert. Je nach Verbindungszustand kann dem Widget eine andere Breite und Höhe zugewiesen werden (Listing 5.14). Die Dimension eines Widgets, das sich darin befindet, wird in einer Zeitspanne `duration` vom letzten zugewiesen Wert bis zum neuen verändert.

```
1 height: snapshotIsConnected.data
2   ? _heightSensorConnected
3   : _heightSensorDisconnected,
4 width: snapshotIsConnected.data
5   ? _widthSensorConnected
6   : _widthSensorDisconnected,
```

Listing 5.14: Zuweisung der Höhe und Breite von einem AnimatedContainer

Hierbei bietet Flutter eine Vielzahl von vorgefertigten Widgets um viele Standardfälle abzudecken, bei denen Animationen erforderlich sind. Mit `FadeInImage` kann z.B. ein Bild als Platzhalter dargestellt werden, bis die Daten geladen sind - und dieses dann animiert einblenden.

Möchte der Benutzer ein neues Gerät mit der App suchen, sendet die UI lediglich einen Event an den Bluetooth BLoC. Werden mehrere Geräte gefunden, wird dies per `Stream` an die UI zurückgegeben. Diese zeigt ein Auswahlmenü an. Der Benutzer kann nun auswählen, mit welchem Gerät dieser sich verbinden will. Dies wird mit einer Funktion `showModalBottomSheet()` ausgeführt, die den lokalen Namen des Gerätes mit jeweils einem `OutlineButton` zum Verbinden anzeigt. In Abbildung 5.9 ist dieses Menü (hierbei mit einem gefundenen Gerät) dargestellt. Durch die Trennung des States von der UI kann hierbei auch ein anderes Konzept eingesetzt werden.

Abb. 5.9: Anzeige eines Auswahlmenüs bei mehreren gefundenen Bluetooth Geräten.

Die Suchergebnisse sind in einer `Map<DeviceIdentifier, ScanResult>` abgespeichert. Das Widget, das diese für den Nutzer visualisiert ist ein `ListView` Widget, dem `bottomSheetElements`, ein Widget vom Typ `List<Widget>` übergeben wird. Diese `List` wird zuvor mit Widgets gefüllt: Einem Header (`_buildHeaderBottomSheet()`) und eine variable Anzahl an Geräten, die gefunden wurden (`_buildScanResult()`). Das Design der in der Liste dargestellten Widgets kann beliebig angepasst werden. Es kann beispielsweise das global verwendete `Theme` verwendet werden um die Primär- und Sekundär-Farben auch hier zu verwenden. Es wäre hierbei auch möglich Bilder verschiedener kompatibler Geräte neben dem Namen des gefundenen Gerätes anzuzeigen.

```
connectBottomSheet(
    BuildContext context, Map<DeviceIdentifier, ScanResult> scanResults) {
  var _bottomSheetElements = List<Widget>();
  _bottomSheetElements.add(_buildHeaderBottomSheet(context));
  _bottomSheetElements.addAll(_        (context, scanResults));
  showModalBottomSheet(
    context: context,
    builder: (BuildContext context) {
      return Container(
        child: Padding(
          padding: EdgeInsets.all(0.0),
          child: ListView(shrinkWrap: true, children: _bottomSheetElements)));
    });
  }
}
```

Listing 5.15: Zuweisung der Höhe und Breite von einem AnimatedContainer

5.2.4 Continuous Integration

Um Abläufe bei der Entwicklung einer App zu verbessern, wird der Build Prozess optimiert. Die Entwicklung neuer Softwareversionen erfolgt auf einem lokalen Gerät, die per *git* und *GitLab* anschließend verwaltet werden. Der Build-Ablauf ist in einer *.gitlab-ci.yml* Datei und in einem *Fastfile* definiert. Hierbei können Tests angegeben werden, die automatisch ausgeführt werden sollen. In einer *.gitignore* Datei kann festgelegt werden, welche Dateien nicht versioniert werden sollen. Beim Erstellen eines Flutter Projektes wird automatisch eine solche Datei mit für Flutter Projekte geeigneten Einstellungen erstellt.

Der GitLab Runner ist ein Open Source Projekt, das firmenintern eingesetzt wird, um automatisiert Aufgaben ausführen zu lassen und die Ergebnisse in GitLab anzeigen zu lassen [77]. GitLab Runner wird zusammen mit *GitLab CI* zur Continuous Integration verwendet. Dabei können bei einem Flutter Projekt die iOS und Android Build Prozesse getrennt gestartet werden. Hier wird die Implementierung für iOS gezeigt.

Wenn das GitLab Projekt einen Runner benutzt, dann wird jeder Commit in die CI Pipeline verschoben. Die *.gitlab-ci.yml* Datei instruiert den Runner, was dieser ausführen soll. Dabei kann es mehrere `stages` geben: `build`, `test` und `deploy`. Im Listing 5.16 ist das *script* des iOS Builds zu sehen.

```
script:
  - cd ios && fastlane cleanproject
  - cd .. && flutter build ios --release --no-codesign -v
  - cd ios && fastlane build
```

Listing 5.16: Flutter iOS Build Script

Dabei werden in diesem Fall zwei Fastlane `lanes` gestartet: `cleanproject` und `build`. Zwischen beiden Aufrufen wird die App gebaut. Fastlane automatisiert Abläufe bei der App-Entwicklung, die zeitaufwändig sind, wie das automatisierte Erstellen von Screenshots für den App Store, Verteilung von Beta Versionen an Testern, Veröffentlichen von Versionen in den App Stores und das Code Signing der App [78].

Die Code Signing Identitäten und Profile können verschlüsselt im Repository abgelegt werden, um diese auf verschiedenen Entwicklungsgeräten zu synchronisieren. Dadurch kann jeder, der Zugriff auf das Verzeichnis hat, einen signierten Build starten [79, 80, 81]. Im Fastfile wird *cert* und *sight* aufrufen (Listing 5.17). Da hierbei auf Routinen aus der nativen Entwicklung zurückgegriffen wird, um die App zu zertifizieren, lässt sich Flutter in bestehende Prozesse gut integrieren.

```
lane :build do
  get_certificates           # invokes cert
  get_provisioning_profile   # invokes sigh
  build_app
end
```

Listing 5.17: Code aus einem Fastfile mit cert und sigh

Die App kann nun mit einer neuen Versionsnummer in den firmeninternen App Catalog hinzugefügt werden. Bis auf leichte Abweichungen ist der Build Vorgang einer Flutter-App analog zu bestehenden Prozessen bei den Android-, oder iOS-Entwicklungsabläufen.

Flutter unterstützt weitere, firmenintern nicht verwendete, CI-Systeme, wie z.B. Cirrus [82]. Google präsentierte bei der Veranstaltung *Flutter Live* das Projekt *Codemagic*, das in Partnerschaft mit *Nevercode* eine Cross-Plattform-Lösung speziell für Flutter liefert [83]. Diese CI/CD baut die App und kann Tests ausführen. Ein eigener Runner muss nicht konfiguriert werden und es bietet sich daher besonders für Firmen an, die keine dedizierte Abteilung für App-Entwicklung besitzen - wie z.B. PolyIC. Da ein eigener Runner für Flutter konfiguriert wurde, wird diese Option nicht weiter verfolgt.

5.3 Testumgebung

Steigt die Anzahl der Komponenten einer Applikation, oder deren Funktionalität und Komplexität, so erhöht sich auch der Aufwand diese zu testen. Tests können manuell ausgeführt werden, oder automatisiert ablaufen. Da jede neue Softwareversion neue Fehler hinzufügen kann, sollte vor jeder Veröffentlichung diese getestet werden. Flutter unterscheidet drei verschiedene Arten von Tests: Unit Tests, Widget Tests und Integration Tests, die im Folgenden vorgestellt werden [84].

5.3.1 Unit Tests

Ein Unit Test überprüft eine Funktion, Methode oder Klasse. Der Begriff Unit ist somit nur ein Synonym für eine Funktion, Methode oder Klasse. Ein Unit Test überprüft die Korrektheit einer Funktionalität unter verschiedenen Situationen. Dabei werden externe Abhängigkeiten *gemocked* (engl. nachgeahmt, oder vorgetäuscht), keine Daten abgespeichert oder gelesen und keine Benutzeroberfläche dargestellt [85]. Dieser Test überprüft ausschließlich die Funktionsweise. Ein Unit Test genügt sowohl für die iOS-, als auch Android-App, da dieselbe Logik überprüft wird.

Wird ein reines Dart *package*, das unabhängig von Flutter ist, getestet, so reicht es zum Testen das *`test` package* in *Pubspec.yaml* einzubinden. Wird der Test nur in Verbindung mit dem Test einer Flutter-App verwendet ist das `flutter_test` *package* nötig. Dieses *package* beinhaltet alles aus dem *`test` package*. Die Einbindung der beiden *packages* ist in Listing 5.18 dargestellt.

```
1 dev_dependencies:
2   flutter_test:
3     sdk: flutter
4   test: ^1.5.1
```

Listing 5.18: Einbindung des flutter_test package in Pubspec.yaml.

Die zu testende Datei befindet sich im *lib* Ordner. Die Datei, welche die Tests beinhaltet sollte in einem *test* Ordner abgelegt werden. Beide Ordner befinden sich in der höchsten Ebene der Flutter-App. Um eine einzelne Datei mit Test(s) auszuführen, schreibt man den Befehl *`flutter test`*, gefolgt vom Pfad der Testdatei. Um alle vorhandenen Tests auszuführen, genügt der Befehl `flutter test` ohne Angabe eines Pfades. In Listing 5.19 ist ein simpler Unit Test dargestellt.

```
1 test('Unit Test var i', () {
2   var i = 11;
3   expect(i, 11);
4 }
```

Listing 5.19: Unit Test einer Variablen

Unit Tests hängen oft von Klassen ab, welche Daten abrufen, oder auf Datenbanken zugreifen. Dies kann Tests stark verlangsamen [84]. Mit *Mock Service*s können Abhängigkeiten simuliert werden. Es können *test stubs* geschrieben werden, also Programme, die ein gewisses Verhalten simulieren. Dies lässt sich mit dem Dart *package* `mockito` vom Dart Team und Dimitry Fibulwinter implementieren. Mehrere Tests können in einer `group` zusammengefasst werden.

5.3.2 Widget Tests

Ein Widget Test, oder auch als Component Test bekannt, überprüft die Funktionsweise und UI eines einzelnen Widgets. Dieses Widget kann mehrere Klassen oder `child` Widgets beinhalten. Daher ist der Widget Test umfangreicher, als ein Unit Test [84]. Die Funktion `testWidgets()` ermöglicht es mit dem *package* `flutter_test` einen Test zu starten. Mit dem *WidgetTester* kann mit dem Widget in der Testumgebung interagiert werden (siehe Listing 5.20).

```
1 testWidgets("Description of test.", (WidgetTester tester) async {
2   // Test Code here
3 });
```

Listing 5.20: Start eines Tests mit dem package flutter_test.

Mit der Funktion `pumpWidget()` wird das Widget gebaut. In Listing 5.21 wird ein Widget (hier: `widgetUnderTest`) parametrisiert und eingebunden.

```
1 await tester.pumpWidget(widgetUnderTest());
```

Listing 5.21: Bauen des Widget *widgetUnderTest* mit einem Initialwert.

Handelt es sich um ein `StatefulWidget` sind die Funktionen `pump()` und `pumpAndSettle()` vom `tester` relevant, da ansonsten das Framework die Änderung des States nicht mitbekommt und

somit falsche Testergebnisse die Folge wären. Es muss dem Framework mitgeteilt werden, dass ein Widget in der Testumgebung neu gebaut werden muss [86]. Der Aufruf von `setState()` und die Methode `build()` wird somit im Test manuell nachgeahmt. Beim Test ist ein Verständnis des `StatefulWidget` nötig, da der Test sonst ein False Negative, oder False Positive liefern kann.

Möchte man das Widget solange aktualisieren, bis keine weiteren Frames mehr geplant sind, wie z.B. bei einer Animation, so kann dies mit `tester.pumpAndSettle()` erfolgen. Dabei wird solange `tester.pump()` aufgerufen, bis keine Aktualisierung mehr anstehen.

5.3.3 Integration Tests

Ein Integration Test überprüft eine komplette App [84]. Dabei führen mehr Abhängigkeiten dazu, dass die Ausführung langsamer als bei andere Tests ist, allerdings auch mehr abdecken kann. Die *packages* `flutter_driver` und `test` müssen für einen Integration Test eingebunden werden. Die Dateien eines Integration Tests werden in einem von den Unit- und Widget Test getrennten Ordner *`test_driver` abgespeichert.*

Hier können ebenfalls mit `SerializableFinders` Widgets gesucht werden und anschließend mit diesen interagiert werden. Die Verbindung mit der App wird vor dem ersten Test mit der Funktion `setUpAll` hergestellt und danach mit `teardownAll` wieder getrennt. Dabei kann mit dem Parameter `key` Widgets eine eindeutige ID zugewiesen werden, um dieses Widget im Test auszuwählen.

Bei der programmierten App kann beispielsweise die Suche nach Geräten mit einer simulierten Berührung per mintinlinedartdriver.tap gestartet werden. Anschließend kann überprüft werden ob die anderen Widgets den korrekten Zustand anzeigen (Listing 5.22).

```
1 test('Press Connect and check if widgets show correct state', () async {
2   expect(await driver.getText(find.byValueKey('ConnectButton')), "Connect");
3   await driver.tap(buttonFinder);
4   expect(await driver.getText(find.byValueKey('DisconnectButton'), "Disconnect");
5 });
```

Listing 5.22: Finden und Interagieren mit Widgets im Integration Test.

Der Integration Test wird mit dem Befehl `flutter drive --target=test_driver/app.dart` gestartet. Zuerst wird dabei die `--target` App gebaut und auf dem Testgerät installiert und gestartet. Anschließend wird die nach dem Gleichheitszeichen angegebene Testdatei ausgeführt. Da der Test auf einem physikalischen Testgerät oder Emulator ausgeführt wird, wird dieses mit `--target` definiert und auf diesem übertragen. Sind wenige Tests implementiert, so dauert das Bauen der App länger als die Tests selbst (z.B. unter eine Sekunde Testzeit bei einer halben Minute Zeit zum Bauen

der App). Die App zeigt hierbei jegliche Interaktionen auf dem Bildschirm des Testgerätes an. Das Gerät kann von der Leistungsfähigkeit limitieren. Durch die Verwendung von *Hot Restart* und *Hot Reload* kann der Test so konfiguriert werden, das die App nicht jedes Mal komplett neu gebaut werden muss, sondern nur der angepasste Test erneut ausgeführt wird. Dies ist muss jedoch manuell konfiguriert werden.

Die Performance einer App kann aufgezeichnet werden (Methode `traceAction()` der Klasse `FlutterDriver`). Dabei wird die Performance in einem `Timeline` Objekt gespeichert. Anschließend kann mit der Klasse `TimelineSummary` die Performance in einem JSON Dokument gespeichert werden, welches mit dem Chrome Tracing Werkezug `chrome://tracing` geöffnet werden kann. Dabei wird unter anderem angezeigt, wie viel Zeit zur Berechnung und Darstellung eines einzelnen Frames gebraucht wurde.

In einem Integration Test kann auch durch eine `List` gescrollt und durchsucht werden. Dabei gibt es mehrere Methoden, wie `scroll` (um einen fixen Wert scrollen), `scrollIntoView` (scrollt zu einem bereits gerenderten Widget) oder `scrollUntilVisible` (scrollt solange bis das gesuchte Widget gefunden wird).

Während eines Integration Tests kann jederzeit ein Screenshot aufgenommen werden. Der Speicherort kann mit `new Directory('screenshots').create()` in `setUpAll` erstellt werden. Mit der Methode `driver.screenshot()` wird der Bildschirminhalt gespeichert.

5.3.4 Erkenntnisse aus der Testumgebung

Die Tests lassen sich in die CI integrieren und die Benachrichtigung bei Fehlern per Instant-Messaging-Dienst Slack erfolgt ähnlich wie bei der Android- und iOS-Entwicklung. Unit Tests ergeben Sinn um Funktionalitäten zu überprüfen, die keinem ständigen Wandel unterzogen sind und lassen sich in Flutter leicht implementieren. Integration Tests überprüfen, ob die App komplett gebaut werden kann und ob alle Elemente in Kombination funktionieren. Der Test findet in einer praxisnahen Umgebung statt. Dabei können jedoch nicht alle Betriebssysteme, oder Software-Versionen und Geräte abgedeckt werden, da dieser Test wesentlich zeitaufwändiger ist. Um die Funktionalität einer Flutter-App in den Grundzügen überprüfbar zu machen, kann z.B. mit einem Test durch die App navigiert werden und überprüft werden, ob auf den Screens die gewünschten Widgets gefunden werden. Die Kombination von *Hot Restart*, *Hot Reload* und auf Widgets optimierten Testfälle ermöglichen eine schnelle Programmierung gängiger Testfälle, auch wenn die Einarbeitung, im Gegensatz zu Flutter und Dart im Allgemeinen, nicht immer intuitiv ist.

Die Modularität der Widgets ermöglicht es an beliebiger Stelle in der App Widget Tests zu integrieren. Bereits getestete Widgets können wiederum in anderen Widgets eingebaut werden, wobei dann die ganze App mit einem Integration Test, oder das neu konstruierte Widget mit einem weiteren Widget Test überprüft werden kann. Ein Design Pattern, dass für die App angemessen ist, kann die Testbarkeit

stark erhöhen, da dadurch die Logik einzelner Komponenten der App zentral überprüfbar gemacht wird und dies für mehrere Betriebssysteme. Wird eine Flutter App mit zwei komplett getrennten Benutzeroberflächen für Android und iOS mit getrennten Widget Bäumen konstruiert, so werden diese getrennt überprüft.

Es ist daher hilfreich, bereits früh im Entwicklungszyklus die Größe und Erweiterbarkeit der App zu planen, um frühzeitig ein passendes Pattern einsetzen zu können, das diese Entwicklung positiv lenken kann. Hierbei können fundamentale Eigenschaften der App frühzeitig in Tests überprüft werden. Die Entwicklungstools von Flutter ermöglichen eine objektive Einschätzung der App-Performance. Mit den Werkzeugen die es aktuell für Flutter gibt, konnte bei schlechter Performance das entsprechende Widget ausfindig gemacht und dieses gezielt optimiert werden. Die Kombination der verschiedenen Werkzeuge ermöglicht kombiniert eine hohe Testabdeckung.

Asynchrone Programmierung spielt bei Dart auch bei den Testabläufen eine entscheidende Rolle. Wird auf ein Ereignis gewartet, so wird dies, wie in einer Dart Anwendung, oder im Flutter App-Code mit `await` programmiert. Das Wissen aus der App-Programmierung und der Widget-Konstruktion kann somit ebenfalls bei den Tests sinnvoll angewendet werden und es wird zur Entwicklung von Tests keine zusätzliche Anwendung, oder Wissen in weiteren Sprachen benötigt.

Screenshots verzögern den Testablauf stark. Fünf Screenshots erhöhten beispielsweise die Zeit eines durchgeführten Test von zehn auf 23 Sekunden. Bei einem Integration Test werden Widgets gesucht und mit diesen interagiert. Einige Testfälle sind aktuell nur kompliziert möglich, wie das Öffnen und Schließen vom `Drawer`, bei dem aktuell die Lokalisierung des Drawer-Symbols mit dem `Tooltip` Widget nötig ist. Dieses Widget fügt Informationen für den Benutzer hinzu, die bei langem Drücken des Widgets angezeigt werden oder bei Sehbehinderung abgespielt werden. Bei Internationalisierung der App ändert sich jedoch der Inhalt des `Tooltip` Widgets und es wird nicht mehr gefunden. Auch Flutter Updates können Einfluss auf diese Problemumgehung haben. Daher sollten andere Test-Tools evaluiert werden, oder abgewägt werden, ob gewisse Tests nicht angewendet werden sollen. In der Flutter Dokumentation sind die Test-Abläufe darüber hinaus sehr stark vereinfacht, was schlechten Coding-Stil zur Folge haben kann. Zum Auffinden von Widgets werden Finder benutzt, die entweder einen `key` oder einen `Text` eines Widgets suchen. Diese Finder werden jedoch in der Testdatei separat von der App konfiguriert, was zum Kopieren und Einfügen der Parameter aus der App in die Test-Datei animieren kann.

Kapitel 6

6 Zusammenfassung und Ausblick

Aktuell wird Flutter zur Entwicklung von Apps für Android und iOS verwendet. Mit der Render-Engine Skia kann eine App mit guter Performance programmiert werden. Besonders wenn eine App mit mehr Schnittstellen kommunizieren soll bietet sich Dart mit dem Fokus auf reaktive, asynchrone Programmierung an. Da Flutter noch nicht lange auf dem Markt ist, sind allerdings noch nicht viele *packages* auf dem Markt die Flutter um Funktionalität erweitern. Deswegen sind einige Anwendungsfälle mit Flutter aktuell noch nicht sinnvoll möglich, da Flutter beispielsweise keine 3D Unterstützung besitzt, oder bestimmte Funktionen nicht von *packages* abgedeckt werden können, wie Autofill bei einem `TextField`. Für Flutter sind aktuell nur wenige Machine Learning *packages* verfügbar und werden noch evaluiert. Durch den Fokus auf UI sind rechenintensive Apps, Apps mit wenig UI oder Apps die nur im Hintergrund laufen, nicht für Flutter geeignet. Die Überprüfung der nötigen Abhängigkeiten und möglichen Implementierungen sollte daher frühzeitig stattfinden.

Da Flutter Android- oder iOS UI-Elemente lediglich nachzeichnet und diese nicht direkt einsetzt, sorgt dafür, dass nach jeder Design-Änderung der unterstützten Betriebssysteme die Widgets überarbeitet werden müssen und dadurch ein Zeitversatz zu einer nativen Entwicklung entstehen kann, jedoch auf weniger APIs geachtet werden muss. Sobald eine UI stark personalisiert sein soll, spielt Flutter seine Vorteile aus und diese kann schnell erstellt werden - das Framework stört hierbei nicht. Es ist mit Flutter möglich, sowohl für iOS als auch Android eine eigene UI zu erstellen, aber im Rahmen dieser Arbeit zeigt sich, dass dies oft gar nicht gewünscht ist. Wichtiger scheint es, dass sich eine App plattformtypisch anfühlen soll, jedoch nicht so aussehen muss.

Flutter Widgets haben sich als sehr flexibel erwiesen, sodass anstatt ein Widget neu zu designen, eine Kombination aus bestehenden Widgets oft ausreicht. Dieser Vorteil lässt sich bei Entwicklungen von App Entwürfen bei KURZ Digital Solutions nutzen, da hierbei bestimmte Bestandteile bei allen Apps vorhanden sind. Mit Flutter kann eine App mit Minimalfunktionalität erstellt werden und anschließend auf realen Endgeräten vorgeführt werden. Hierbei können früher Fehler im grundsätzlichen Bedienkonzept aufgedeckt werden, als bei einer reinen Darstellung am Computer mit Sketch, das bisher verwendet wird. Durch *Hot Reload* und *Hot Restart* kann in der Entwicklung schnell iteriert werden und somit das Konzept verbessert werden. Der erste Entwurf einer App kann nach wenigen Stunden oder Tagen vorliegen. Auch die Debug-Möglichkeiten der UI und der Performance ohne Zusatzsoftware ermöglicht bereits dabei ein gutes Verständnis über Probleme zu gewinnen. Flutter bietet sich daher auch zur Anfertigen von Prototyp-Apps an, bevor diese in der Sprache des Betriebssystem-Herstellers nativ umgesetzt werden.

Eine App lebt von flüssigen Animationen, die bei Flutter schnell programmiert werden können. Diese können leicht parametrisiert werden und sind dadurch eine der größten Stärken von Flutter. Animationen können einzeln ablaufen oder aneinandergereiht werden. Eine sinnvolle Abstrahierung der Logik der App ist bei Flutter besonders wichtig, da ansonsten durch unnötiges Aktualisieren großer Teile der UI viel Performance verloren geht. Wird die UI und Logik der App gut getrennt, kann von verschiedenen Mitarbeitern unabhängig an der App gearbeitet werden - bei klaren Definition der Schnittstellen. Designer könnten einen größeren Einfluss in der App-Entwicklung einnehmen, wodurch sich das gesamte Entwicklungstempo in der App-Entwicklung erhöhen kann. Die Design-Änderungen können hierbei an der selben Code-Basis erfolgen.

Je nach Anwendungsfall der App kann fast vollständig auf plattformspezifischen Code verzichtet werden. Dadurch genügt lediglich Verständnis für Dart, um sowohl den Code für die UI, als auch die Funktionsweise der App zu erhalten. Die Ähnlichkeit zu anderen Programmiersprachen, von denen Dart inspiriert wurde, ermöglicht vielen Programmierern unmittelbar große Teile einer Flutter-App zu verstehen, ohne dass dieser sich vorher mit Dart beschäftigt hat. Die Sprache Dart ist (nicht nur) bei der Firma KURZ Digital Solutions wenig vertreten, weswegen für zukünftige Flutter-Projekte eine Einarbeitungszeit nötig ist. Sowohl das iOS- als auch das Android-Team von KURZ Digital Solutions sind Dart nicht abgeneigt.

Nur wenn das generelle Interesse an Dart weiterhin steigt, können sich Flutter im Kontext der Cross-Plattform-Frameworks und Dart als Programmiersprache profilieren. Um dies zu erreichen, wurde bei der Veranstaltung Flutter Live im Dezember 2018 das *Projekt Hummingbird* vorgestellt, das das Ziel verfolgt, Flutter im Browser mithilfe von Skia laufen zu lassen [83]. Bestehender Dart Code soll kompatibel sein, da der Unterschied zu Flutter bei Mobilgeräten, der Einsatz der *Flutter Web Engine* ist. Dart- und JavaScript Libraries sollen sich einbinden lassen. Der Source Code des Projekts ist noch nicht veröffentlicht. Ob das zukünftige Betriebssystem Fuchsia die Popularität von Flutter erhöhen wird kann aktuell nicht abgeschätzt werden.

Da Flutter die Wahl des Betriebssystems nicht vorschreibt, steht es jedem Entwickler offen, das Framework auf andere Plattformen zu portieren, wie es zuletzt beim Ein-Platinen-Computer Raspberry Pi durch Chinmay Garde, geschehen ist [87]. Aktuell in Entwicklung beim Flutter-Team ist die Portierung von Flutter-Applikationen auf den Desktop (MacOS, Windows, Linux) [83, 88]. Hierbei wird an Widgets gearbeitet, die speziell für die Maussteuerung optimiert sind. Befindet sich der Mauszeiger auf einem Widget, so kann der Mauszeiger seine Gestalt wandeln, oder das Widget verändert sich, um Möglichkeiten der Interaktion zu symbolisieren. Die Unterstützung von Tastaturkombinationen oder dem Scrollrad sind ebenfalls geplant.

Im Rahmen dieser Arbeit konnte mit dem SDK Flutter innerhalb weniger Wochen ein App programmiert werden, die vom Kunden sehr positiv aufgenommen wurde. Die Firma PolyIC hat im Anschluss den Bau von mehreren Demonstratoren in Auftrag gegeben, die über Bluetooth 5.0 Module verfügen sollen. Da Flutter deklarativ ist beschreiben die Widgets lediglich die Darstellung des States. Die Wahl des State-Managements kann daher bei unterschiedlichen Teams anders erfolgen, ohne dass

die eine Wahl schlechter als die andere ist. Die Performance der App konnte durch die Wahl des BLoC-Pattern gesteigert und der Code übersichtlicher gestaltet werden. Die App erreichte in Tests eine durchgehend hohe Performance. Anschließend konnten Animationen schnell eingebaut werden, um so die App ansprechender zu gestalten. Das BLoC-Pattern bietet sich besonders für größere, möglichst skalierbare Projekte an.

Die reaktive Programmierung von Dart ermöglicht es die UI jederzeit für Eingaben reaktionsbereit zu gestalten und zügig die App zu erweitern. Die Einbindung der Flutter Entwicklung in die firmeneigenen Prozesse war ebenfalls möglich, weswegen dem Einsatz von Flutter bei weiteren Entwicklungen nichts im Wege steht. Für KURZ Digital Solutions ist zukünftig neben der mobilen Entwicklung besonders Flutter im Browser und für PolyIC Flutter für Embedded-Systeme im Prototypbau von hohem Interesse.

Verzeichnis der Programmlistings

Abkürzungen

AOT Ahead of Time

API Application Programming Interface (kurz API) ist eine Programmierschnittstelle die eine Anbindung von Außen an ein bestehendes System bietet. Die Schnittstelle wird vom Hersteller der Software zur Verfügung gestellt um Dritten die Möglichkeit eines Addons oder einer Erweiterung zu geben. Die Schnittstelle definiert dabei ein Standardformat und Standardbefehle.

BLE LE steht für Low Energy und findet seit der Bluetooth Version 4.0 Verwendung bei energiesparsame Chips, die nur geringe Datenraten besitzen.

BLoC Business Logic Component

Build Als Build wird der gesamten Vorgang, der Erzeugung einer eigenständig lauffähigen Software genannt. Dabei wird aus den Quelldateien ein ausführbares Konstrukt geschaffen. Ein Build beinhaltet das Kompilieren der Quellen. In der Regel hat ein Build eine eindeutige Build Nummer.

CD Continuous Delivery oder Continuous Deployment

CI Continuous Integration

OEM Original Equipment Manufacturer, oder Erstausrüster

UI User Interface

UX Nutzererfahrung bei der Benutzung der Software

VM Virtual Machine

Literaturverzeichnis

[1] PolyIC GmbH & Co. KG: *Bedienelemente - PolyIC*. http://www.polyic.de/anwendungen/bedienelemente/

[2] Google: *Technical Overview - Flutter*. https://flutter.dev/docs/resources/technical-overview

[3] Google: *Flutter System Architecture*. https://docs.google.com/presentation/d/1cw7A4HbvM_Abv320rVgPVGiUP2msVs7tfGbkgdrTy0I/edit#slide=id.p. Version: April 2017

[4] Boelens, Didier: *Flutter - Reactive Programming - Streams - BLoC*. https://www.didierboelens.com/2018/08/reactive-programming---streams---bloc/. Version: August 2018

[5] Google: *Flutter Release Preview 2: Pixel-Perfect on iOS*. https://developers.googleblog.com/2018/09/flutter-release-preview-2-pixel-perfect.html. Version: September 2018

[6] ReactiveX: *ReactiveX - Subject*. http://reactivex.io/documentation/subject.html

[7] Gartner: *Market Share: Final PCs, Ultramobiles and Mobile Phone, All Countries, 4Q17*. https://www.gartner.com/en/newsroom/press-releases/2018-02-22-gartner-says-worldwide-sales-of-smartphones-recorded

[8] Google: *Dart programming language | Dart*. https://www.dartlang.org/. Version: 2019

[9] Apple: *Human Interface Guidlines*. https://developer.apple.com/design/human-interface-guidelines/. Version: 2019. – [Online; Stand 02. Januar 2019]

[10] Google: *Material Design*. https://material.io. Version: 2019. – [Online; Stand 02. Januar 2019]

[11] PolyIC GmbH & Co. KG: *Touch Controls - PolyIC*. http://www.polyic.com/applications/touch-controls/

[12] Google: *Dart's Type System | Dart*. https://www.dartlang.org/guides/language/sound-dart. Version: 2019

[13] Google: *Ecma forms TC52 for Dart Standardization*. https://news.dartlang.org/2013/12/ecma-forms-tc52-for-dart-standardization.html. Version: Dezember 2013. – [Online; Stand 02. Januar 2019]

[14] Akopkokhyants, Sergey: *Mastering Dart*. PACKT Publishing, 2014

[15] Google: *Dart Programming Language Specification 5th edition draft*. https://www.dartlang.org/guides/language/specifications/DartLangSpec-v2.2.pdf. Version: Februar 2019

[16] Google: *Dart Tools*. https://www.dartlang.org/tools. Version: 2019. – [Online; Stand 15. Januar 2019]

[17] Google: *Flutter Packages*. https://pub.dartlang.org/flutter. Version: 2019

[18] Google: *Pub Package Manager*. https://www.dartlang.org/tools/pub/

[19] Google: *Effective Dart: Style | Dart*. https://www.dartlang.org/guides/language/effective-dart/style. Version: 2019

[20] Google: *Effective Dart: Design | Dart*. https://www.dartlang.org/guides/language/effective-dart/design. Version: 2019

[21] Ladd, Seth: *Const, Static, Final, Oh my!* https://news.dartlang.org/2012/06/const-static-final-oh-my.html. Version: Juni 2012

[22] Google: *Language Tour Dart*. https://www.dartlang.org/guides/language/language-tour. Version: 2019

[23] Google: *List class - dart:core library - Dart API*. https://api.dartlang.org/stable/2.2.0/dart-core/List-class.html. Version: 2019

[24] Google: *Map class - dart:core library - Dart API*. https://api.dartlang.org/stable/2.2.0/dart-core/Map-class.html. Version: 2019

[25] Ullenboom, Christian: *Java ist auch eine insel*. http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_05_005.htm#mj10809df0d7ff9feb8f7942bb9c23e0e5. Version: 2011

[26] Google: *Static Analyzer | Dart*. https://www.dartlang.org/tools/analyzer. Version: 2019

[27] Bonér, Jonas ; Farley, Dave ; Kuhn, Roland ; Thompson, Martin: *The Reactive Manifesto*. https://www.reactivemanifesto.org/. Version: September 2014

[28] Google: *Git repositories on fuchsia*. https://fuchsia.googlesource.com/

[29] Google: *Flutter performance profiling - Flutter*. https://flutter.dev/docs/testing/ui-performance. Version: 2019

[30] Google: *flutter/flutter: Flutter makes it easy and fast to build beautiful mobile apps*. https://edit.theappbusiness.com/flutter-the-skys-the-limit-84887c8f650d

[31] Dalal, Navneet: *Announcement from Flutter*. https://flutterapp.com/

[32] Google: *Skia Graphics Library*. https://skia.org/. Version: 2019

[33] MALIK, OM: *Google Open Sources Skia Graphics Engine*. https://gigaom.com/2008/09/02/google-open-sources-skia-graphics-engine/. Version: September 2008

[34] Google: *Graphics and Skia - The Chromium Projects*. https://www.chromium.org/developers/design-documents/graphics-and-skia. Version: 2019

[35] Google: *FAQ - Flutter*. https://flutter.dev/docs/resources/faq. Version: 2019

[36] Bohn, Dieter: *Google Flutter is out of beta*. https://www.theverge.com/2018/12/4/18125053/google-flutter-1-0-skia-mobile-app-cross-platform-developers. Version: Dezember 2018

[37] Google: *Introduction to Widgets*. https://flutter.io/docs/development/ui/widgets-intro. Version: 2019

[38] Google: *State class - widgets library*. https://docs.flutter.io/flutter/widgets/State-class.html. Version: 2019

[39] Google: *Differentiate between ephemeral state and app state - Flutter*. https://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app

[40] Google: *setState methode - State class*. https://docs.flutter.io/flutter/widgets/State/setState.html. Version: 2019

[41] Google: *didUpdateWidget - State class*. https://docs.flutter.io/flutter/widgets/State/didUpdateWidget.html. Version: 2019

[42] Google: *build method - State class*. https://docs.flutter.io/flutter/widgets/State/build.html. Version: 2019

[43] Google: *deactivate method - State class*. https://docs.flutter.io/flutter/widgets/State/deactivate.html. Version: 2019

[44] Google: *dispose method - State class*. https://docs.flutter.io/flutter/widgets/State/dispose.html. Version: 2019

[45] Google: *runApp function*. https://docs.flutter.io/flutter/widgets/runApp.html. Version: 2019

[46] Google Developers: *Understand Tasks and Back Stack*. https://developer.android.com/guide/components/activities/tasks-and-back-stack

[47] Apple Developer: *UINavigationController - UIKit*. https://developer.apple.com/documentation/uikit/uinavigationcontroller

[48] Google: *Navigator class - widgets library - Dart API*. https://docs.flutter.io/flutter/widgets/Navigator-class.html

[49] Google: *Widget catalog - Flutter*. https://flutter.dev/docs/development/ui/widgets

[50] Google: *Scaffold class - material library - Dart API*. https://docs.flutter.io/flutter/material/Scaffold-class.html

[51] Google: *Adding Assets and Images*. https://flutter.io/docs/development/ui/assets-and-images. Version: 2019. – [Online; Stand 11. Februar 2019]

[52] Google: *yaml Dart Package*. https://www.dartlang.org/tools/pub/pubspec. Version: 2019

[53] Google: *Product Icons - Material Design*. https://material.io/design/iconography/#. Version: 2019

[54] Google: *Hot Reload*. https://flutter.dev/docs/development/tools/hot-reload

[55] Google: *Start thinking declarative*. https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative. Version: 2019

[56] Google: *Introduction to declarative UI*. https://flutter.dev/docs/get-started/flutter-for/declarative

[57] Google: *updateShouldNotify method*. https://docs.flutter.io/flutter/widgets/InheritedWidget/updateShouldNotify.html. Version: 2019

[58] Google Developers: *Build reactive mobile apps with Flutter (Google I/O '18)*. https://www.youtube.com/watch?v=RS36gBEp8OI. Version: Oktober 2018

[59] Google Developers: *Flutter / AngularDart - Code sharing, better together*. https://www.youtube.com/watch?v=PLHln7wHgPE. Version: Januar 2018

[60] Google: *Writing custom platform-specific code*. https://flutter.dev/docs/development/platform-integration/platform-channels

[61] Hickson, Ian: *Add Flutter to existing apps*. https://github.com/flutter/flutter/wiki/Add-Flutter-to-existing-apps. Version: März 2019

[62] Android Developers: *Bluetooth low energy overview* . https://developer.android.com/guide/topics/connectivity/bluetooth-le

[63] *What's New in iOS - iOS 5.0*. https://developer.apple.com/library/archive/releasenotes/General/WhatsNewIniOS/Articles/iOS5.html

[64] Bundesamt für Strahlenschutz: *Sprach- und Datenübertragung per Funk: Bluetooth und WLAN*. https://www.bfs.de/SharedDocs/Downloads/BfS/DE/broschueren/emf/info-bluethooth-und-wlan.pdf?__blob=publicationFile&v=5. Version: August 2012

[65] Bluetooth Special Interest Group: *Bluetooth Core Specification 4.2 Frequently Asked Questions*. https://www.ineltek.com/wp-content/uploads/2015/09/20160330-Bluetooth4-2FAQ.pdf. Version: 2014

[66] Bluetooth SIG: *Bluetooth Core Specification 5.0*. https://developer.bluetooth.org/gatt/Pages/GATT-Specification-Documents.aspx. Version: Dezember 2016

[67] Group, Bluetooth Special I.: *GATT Specifications | Bluetooth Technology Website*. https://www.bluetooth.com/specifications/gatt. Version: 2019

[68] Glinz, Martin: A Risk-Based, Value-Oriented Approach to Quality Requirements. In: *IEEE Software* 25 (2008), Nr. 2, 34–41. http://dx.doi.org/10.1109/MS.2008.31. – DOI 10.1109/MS.2008.31

[69] ReactiveX: *ReactiveX - Operators*. http://reactivex.io/documentation/operators.html

[70] Pepermans, Frank ; Egan, Brian: *RxDart*. https://pub.dartlang.org/packages/rxdart

[71] Adafruit: *Adafruit Feather M0 Bluefruit LE ID: 2995*. https://www.adafruit.com/product/2995. Version: 2019

[72] Adafruit: *UART Service | Adafruit Feather M0 Bluefruit LE | Adafruit Learning System*. https://learn.adafruit.com/adafruit-feather-m0-bluefruit-le/uart-service. Version: 2019

[73] DeMarco, Paul: *flutter blue | Flutter Package*. https://pub.dartlang.org/packages/flutter_blue. Version: 2019

[74] Kowase, Rui: *How to generate Flutter app icons for iOS and Android*. https://dev.to/rkowase/how-to-generate-flutter-app-icons-for-ios-and-android-11gc. Version: April 2018

[75] 2D, Inc (2Dimensions): *2D - Pricing*. https://www.2dimensions.com/pricing. Version: 2019

[76] 2D, Inc (2Dimensions): *2D - Flare by 2Dimensions. Bring your apps and games to life with real-time animation.* https://www.2dimensions.com/about-flare. Version: 2019

[77] GitLab: *GitLab Runner*. https://docs.gitlab.com/runner/

[78] Google Developers: *Fastlane Tools*. https://fastlane.tools/. Version: 2019

[79] Google Developers: *fastlane - App automation done right*. https://docs.fastlane.tools/codesigning/getting-started/. Version: 2019

[80] Google Developers: *Fastlane Cert*. https://docs.fastlane.tools/actions/cert/. Version: 2019

[81] Google Developers: *Fastlane Sigh*. https://docs.fastlane.tools/actions/sigh/. Version: 2019

[82] Google: *Continuous Delivery using fastlane with Flutter*. https://flutter.dev/docs/deployment/fastlane-cd

[83] Google Developers: *Flutter Live 2018 - All Talks*. https://www.youtube.com/playlist?list=PLOU2XLYxmsILq4ysYNWXq5TOGLgYDJgVD. Version: Dezember 2018

[84] Google: *Testing Flutter apps - Flutter*. https://flutter.io/docs/testing. Version: 2019

[85] Google: *An introduction to unit testing - Flutter*. https://flutter.dev/docs/cookbook/testing/unit/introduction. Version: 2019

[86] Google: *An introduction to widget testing - Flutter*. https://flutter.io/docs/cookbook/testing/widget/introduction

[87] Garde, Chinmay: *Flutter on Raspberry Pi (mostly) from Scratch*. https://medium.com/flutter-io/flutter-on-raspberry-pi-mostly-from-scratch-2824c5e7dcb1. Version: November 2018

[88] Kozsir, Simon Lightfoot & N.: *Flutter on desktop, a real competitor to Electron*. https://medium.com/flutter-community/flutter-on-desktop-a-real-competitor-to-electron-4f049ea6b061. Version: Dezember 2018