Masterarbeit

Analyse und Bewertung von State-Management Varianten in modernen Cross-Plattform Technologien am Beispiel von Flutter

ausgeführt an der

am Studiengang
Informationstechnologien und Wirtschaftsinformatik

Von: Thomas Karner
Pers. Kennz. 1910320007

Graz, am 28. November 2020

....
Thomas Karner

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

....
Thomas Karner

Kurzfassung

Plattformübergreifende Technologien entwickeln sich ständig weiter und bieten einen kostengünstigen und zeiteffizienten Ansatz zur Entwicklung von Anwendungen auf mehreren Plattformen. Die Frage nach dem richtigen State-Management Ansatz muss dabei zu Beginn eines jeden Projekts beantwortet werden, was eine Herausforderung sein kann mit einer Vielzahl an mögliche Lösungen. Das Ziel dieser Arbeit war daher die Untersuchung von State-Management Ansätzen am Beispiel von Flutter und deren Bewertung auf der Grundlage von festgelegten Kriterien. In dieser Arbeit werden zunächst plattformübergreifende Technologien untersucht, mit Schwerpunkt auf Flutter. Danach werden gängige Ansätze von State-Management und die Theorie hinter Code Qualität Metriken analysiert. Basierend auf den gesammelten Informationen und den daraus resultierenden wirtschaftlichen und technologischen Entscheidungsgründen wurden Bewertungskriterien erstellt. Vier State-Management Ansätze wurden umgesetzt und entsprechend der Kriterien bewertet. Der Vergleich zeigt, dass BLoC die beste Lösung für Großanwendungen mit erfahrenen Entwicklern zu sein scheint. Für kleinere Projekte oder unerfahrene Entwickler bietet sich der Einsatz von Provider an.

Abstract

Cross-platform technologies are constantly evolving. They offer a cost-effective and time-efficient approach to developing applications for multiple platforms. The most appropriate state management approach is determined at the beginning of each project, which can be a challenge with a variety of possible solutions. This thesis, therefore, investigates and evaluates common state management approaches for Flutter. First, cross-platform technologies focussing on Flutter are explored. Following this, common state management approaches and code quality measures are described. Based on the collated information and the resulting economic and technological decision factors, evaluation criteria are created. Four state management approaches are then implemented and evaluated according to the set criteria. The comparison indicates that BLoC is the best solution for large-scale applications with experienced developers. For smaller projects or junior developers, Provider seems optimal.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Aufgabenstellung 1
1.2 Zielsetzung und Forschungsfrage 1
1.3 Methodik und Aufbau der Arbeit 2

2 Cross Plattform Technologien **3**
2.1 Gründe für die Auswahl 3
2.2 Varianten 4
2.3 Native Bridge 4
2.4 Vertreter 5
2.4.1 Xamarin 5
2.4.2 React Native 5
2.4.3 Ionic 5

3 Flutter im Detail **6**
3.1 Render Engine und Plattform Features 7
3.2 Alles ist ein Widget 9
3.3 Stateful und Stateless Widgets 10
3.4 Programmiersprache Dart 11
3.5 Best Practices der Technologie 12
3.6 Testing in Flutter 14
3.7 Relevanz und Verbreitung 15

4 Deklarative UI **16**

5 Reaktive Programmierung **18**
5.1 Reactive Manifesto 18
5.2 Flutter als reaktives Framework 19
5.2.1 Erstellung von Streams 19
5.2.1.1 Transformation 20
5.2.1.2 async* Funktion 21
5.2.1.3 StreamController 21
5.2.2 Verwendung von Streams 22

6 State Management **23**
6.1 Allgemeines zu State Management 23
6.1.1 Ephemeral State 23
6.1.2 App State 24

6.2 State Management Varianten in Flutter 25
6.2.1 StatefulWidget's setState 26
6.2.2 InheritedWidget und InheritedModel 27
6.2.3 Provider 30
6.2.4 Redux 32
6.2.5 BLoC 34
6.2.6 MobX 36

7 Software Qualität und Code-Metriken **38**
7.1 Einordnung und Abgrenzung 38
7.2 Werkzeuge 40
7.3 Relevanz und Auswahl 41

8 Methodik der empirischen Untersuchung **42**
8.1 Funktionsumfang der Beispielapplikation 42
8.2 Herangehensweise 43

9 Umsetzung und Bewertung der Varianten **45**
9.1 Variante - Provider 45
9.2 Variante - Redux 50
9.3 Variante - BLoC 56
9.4 Variante - MobX 62

10 Analyse und Interpretation der Ergebnisse **67**
10.1 Auswertung 67
10.2 Erkenntnisse 70

11 Zusammenfassung **71**

Acronyms **72**

Abbildungsverzeichnis **73**

Tabellenverzeichnis **74**

Listings **75**

Literaturverzeichnis **77**

1 Einleitung

Cross-Plattform Applikationen werden durch ihren reduzierten Entwicklungsaufwand und der daraus resultierenden Time-to-Market immer interessanter für Unternehmen und Entwickler. Neue Technologien versprechen eine Performance vergleichbar mit nativen Applikationen und das mit einer einheitlichen Codebasis für mehrere Plattformen. Neue Technologien bringen oft neue oder überarbeitete Ansätze hinsichtlich Architektur und State-Management, wobei durch eine stetige Weiterentwicklung und den aktiven Communities oft mehrere Lösungen für das gleiche Problem entstehen. Diese rege Weiterentwicklung führt dazu, dass es meist noch keinen empfohlenen Standard gibt, welcher für Einsteiger oder Unerfahrene als Leitfaden dienen kann.

1.1 Aufgabenstellung

Die Aufgabenstellung dieser Arbeit widmet sich der Analyse und Beurteilung möglichen State-Management Varianten am Beispiel einer aktuellen Cross-Plattform Technologie – Flutter. Dabei wird sowohl auf verschiedene Cross-Plattform Technologien und deren Aufbau, als auch auf Flutter im Detail eingegangen. Grundkonzepte zu reaktiver Programmierung und deklarativem UI werden theoretisch behandelt und Softwaremetriken werden erarbeitet. Vor der Umsetzung der einzelnen Varianten werden alle Ansätze im Theorieteil aufbereitet, um deren Unterschiede und Ansätze aufzuzeigen.

1.2 Zielsetzung und Forschungsfrage

Ziel dieser Arbeit ist die Gegenüberstellung aktueller State-Management Varianten für Cross-Plattform Technologien und deren genaueren Analyse und Bewertung am Beispiel von Flutter. Zusätzlich werden Programmier-Paradigmen wie reaktive Programmierung oder deklaratives UI genauer betrachtet und für die Analyse der Varianten berücksichtigt. Anhand der Theorie zu statischer Codeanalyse und den wirtschaftlichen sowie technologischen Anforderungen an State-Management Ansätze

soll eine Bewertungsskala für die Beurteilung der Varianten erstellt werden, um somit eine nachvollziehbare und transparente Beurteilung zu schaffen. Dabei sollen sowohl Technologie-agnostische als auch spezifische Eigenschaften der ausgewählten Technologie in die Bewertungsskala einfließen. Für die Bewertung der Varianten nach der erstellten Skala wird eine Beispielapplikation mit klar definiertem Funktionsumfang in den verschiedenen State-Management Varianten umgesetzt. Die Auswahl der Varianten erfolgt einerseits über die offizielle Liste der State-Management Ansätze des Herstellers (`https://flutter.dev/docs/development/data-and-backend/state-mgmt/options`) sowie über relevante Kennzahlen für die Verbreitung der Varianten (Likes auf `pub.dev`, GitHub Stars und aktive Weiterentwicklung).

Die Forschungsfrage dahingehend lautet: *Welche wirtschaftlichen und technologischen Entscheidungsgründe gibt es für die Auswahl einer State-Management Variante in modernen Cross-Plattform Technologien am Beispiel von Flutter?*

1.3 Methodik und Aufbau der Arbeit

Zu Beginn werden Grundlagen rund um Cross-Plattform Technologien aufbereitet und bekannte Vertreter kurz vorgestellt. Um für die Auswertung die notwendigen theoretischen Kenntnisse über das Framework zu bekommen werden Flutter und dessen Ansätze genauer betrachtet. Im nächsten Schritt werden die ausgewählten Varianten anhand ihrer theoretischen Grundlagen analysiert, um für die Umsetzung und Bewertung die notwendigen Informationen aufzubereiten. Um die Forschungsfrage nach den wirtschaftlichen und technologischen Entscheidungsgründen beantworten zu können werden die Themen Software Qualität und Code-Metriken genauer betrachtet, dessen Ergebnisse in die Bewertungskriterien einfließen. Anhand einer Beispielapplikation mit klar definiertem Funktionsumfang werden die State-Management Varianten umgesetzt, Besonderheiten des Ansatzes dokumentiert und anhand der erstellten Kriterien bewertet. Die Ergebnisse der statischen Codeanalyse werden den Bewertungen der Kriterien gegenübergestellt und zwischen den Varianten verglichen. Anhand der Ergebnisse wird die Forschungsfrage hinsichtlich der Entscheidungsgründe für eine Variante beantwortet.

2 Cross Plattform Technologien

Zwei große Plattformen dominieren in der aktuellen Zeit den mobilen Markt, Android und iOS. Dabei gibt es die Anforderung von Entwicklern und Organisationen mit möglichst geringem Aufwand beide Plattformen mit dem gleichen Umfang an Funktionen zu bedienen, sowie die stetige Weiterentwicklung von Hard- und Software mit zu begleiten (Tablets, Wearable, etc.). Beim nativen Ansatz wird die App für die jeweilige Plattform eigens entwickelt, was es ermöglicht die jeweiligen Designrichtlinien und Plattform-Features zu nutzen. Der Cross-Plattform Ansatz hingegen setzt auf eine gemeinsame Code-Basis, welche durch verschiedene technologische Ansätze für die jeweilige Plattform bereitgestellt werden kann (Rieger & Majchrzak, 2019).

2.1 Gründe für die Auswahl

Unternehmen sind heutzutage vermehrt dazu gezwungen, ihre Services für Kunden auch mobil zur Verfügung zu stellen, da immer mehr bestehende Services auf mobile Unterstützung setzen. Da die Entwicklung dieser Apps aber einen monetären Aufwand seitens der Unternehmen darstellt, wird der Cross-Plattform Ansatz als mögliche Alternative zu klassischen nativen Apps gewählt um die anfallenden Kosten zu minimieren. Dabei reduzieren sich vermeintlich nicht nur die Kosten, es kann auch plattformübergreifend ein einheitliches Design sowie ein einheitlicher Funktionsumfang sichergestellt werden (Andrade, Albuquerque, Frota, Silveira & da Silva, 2015).

Der Cross-Plattform Ansatz ermöglicht zudem eine vereinfachte Wartung und Weiterentwicklung der Applikation durch die einheitliche Codebasis, wobei auf plattformspezifische Funktionen und Schnittstellen zugegriffen werden kann. Dem Endanwender kann dabei die Applikation über die üblichen Distributionswege bereitgestellt werden (AppStore für iOS und PlayStore für Android). Die vermehrte Nachfrage an mobilen Apps mit den daraus resultierenden Aufwänden für bestehende Unternehmen führt dazu, dass Cross-Plattform-Technologien stetig wachsen und weiterentwickelt werden. Dabei wird nicht nur auf bestehende Programmiersprachen und Technologien aufgebaut, sondern es werden auch auch gänzlich neue Ansätze entwickelt (siehe Kapitel 2.4) (Xanthopoulos & Xinogalos, 2013).

2.2 Varianten

Für die Umsetzung von plattformunabhängigen Applikationen stehen eine Vielzahl an verschiedenen Technologien zur Verfügung, wobei grob unter drei verschiedenen Ansätzen unterschieden werden kann. Keine dieser Ansätze ist besser geeignet als ein anderer für die Erstellung einer Applikation, vielmehr hat jeder Ansatz seine eigenen Vor- und Nachteile, welche für eine Auswahl in Betracht gezogen werden müssen (Xanthopoulos & Xinogalos, 2013).

- **Web Apps** - diese werden über eine URL im Browser zur Verfügung gestellt und basieren auf HTML und JavaScript. Durch den HTML5 Standard können einige Device-APIs angesprochen und genutzt werden.

- **Hybrid Apps** - hier wird eine Web App mittels nativem Container gewrappt. Dieser Container besteht aus einer Webview, auf jenem die Web App ausgeführt wird, und den plattformspezifischen APIs, mit welchen auf die Hardware des Gerätes zugegriffen werden kann. Die daraus resultierende Applikation kann wie eine native App über den jeweiligen Store angeboten und installiert werden.

- **Cross-Plattform Apps** - bei diesem Ansatz wird kein Browser oder ein Container mit einer Webview für die Anzeige verwendet. Der erstellte Code wird für die jeweilige Plattform aufbereitet und hat über diverse Schnittstellen Zugriff auf die Gerätefunktionen. Durch die native Darstellung der Komponenten ähnelt dieser Ansatz am meisten einer nativen Applikation (Details in Kapitel 3.1).

2.3 Native Bridge

Der Erfolgsfaktor einer Cross-Plattform-Technologie hängt stark von der Möglichkeit ab, auf die Gerätefunktionen zugreifen zu können (Speicher, Kamera, Bluetooth, etc.). Bei der sogenannten Bridge handelt es sich um eine Schnittstelle, mit welcher auf die Funktionalitäten des Gerätes zugegriffen werden kann. Manche Cross-Plattform Ansätze benötigen durch ihren Aufbau bzw. deren Herangehensweise keine Bridge (z.B. Cross-Compiled), wohingegen z.B. Hybrid Apps diese benötigen, um nativen Code ausführen zu können. Der Vorteil liegt darin, dass durch eine Abstraktion zwischen nativen und Cross-Plattform Code der native Teil generisch gehalten und in verschiedenen Projekten wiederverwendet werden kann. Ein Beispiel dazu wäre der Zugriff auf die Kamera, wobei im nativen Teil nur der Zugriff auf die Hardware implementiert wird und im Cross-Plattform Code der weitere Prozess (Biørn-Hansen & Ghinea, 2018).

2.4 Vertreter

Die Anforderungen an eine Cross-Plattform Technologie sind nicht neu und viele Frameworks und Unternehmen haben diesen Ansatz bereits aufgegriffen um eine Lösung für das Problem zu liefern. Jede dieser Varianten hat seine eigenen Vor- und Nachteile, welche bei der Auswahl zu berücksichtigen sind. Nachfolgend werden aktuelle Vertreter von Hybrid und Cross-Plattform-Technologien kurz vorgestellt, wobei auf Flutter erst im Kapitel 3 eingegangen wird (Rieger & Majchrzak, 2016).

2.4.1 Xamarin

Xamarin bietet die Möglichkeit mittels C# native Applikationen für mobile Plattformen zu erstellen. Durch die vorhandenen Wrapper, welche eine Cross-Plattform Implementierung vom .NET Framework sind, kann plattformunabhängiger Code erstellt werden, welcher auf mehreren Plattformen eingesetzt werden kann (von Business Logik bis hin zum UI durch das Xamarin.Forms Framework). Dabei stellt Xamarin einige Wrapper für native Schnittstellen und einen Compiler, welcher nativen Code für die jeweilige Plattform erstellt, zur Verfügung (Bennett, 2018).

2.4.2 React Native

React Native ist ein Open Source Projekt von Facebook, welches die Entwicklung von Cross-Plattform Applikationen ermöglicht. Das Framework setzt dabei auf den Einsatz von JavaScript, CSS und HTML, wobei es nicht auf herkömmliche Verfahren von Web-Frameworks angewiesen ist (wie z.B. eine WebView). Die UI Komponenten werden dabei über sogenannte OEM-Widgets nativ am Gerät dargestellt, mehr dazu im Kapitel 3.1. React Native und dessen Syntax baut dabei auf React auf, einem JavaScript Framework für die Erstellung großer und komplexer Webapplikationen (Paul & Nalwaya, 2019).

2.4.3 Ionic

Ionic bietet die Möglichkeit, auf Basis von Webtechnologien, Hybrid-Apps zu erstellen. Das Framework baut dabei auf Web Standards auf und kann mit verschiedenen anderen Frameworks kombiniert werden (z.B. Angular, React oder Vue). Durch den Einsatz von Web Components erleichtert es zudem die Erstellung von kleinen und wiederverwendbaren Komponenten und bietet eine Vielzahl an häufig benötigen Komponenten bereits durch das Framework an (Cheng, 2018).

3 Flutter im Detail

Es existiert eine Vielzahl an Frameworks und Technologien, welche ein gemeinsames Problem zu lösen versuchen - die Erstellung mobiler Applikationen für relevante Plattformen mittels einer einheitlichen Codebasis. Obwohl Flutter noch zu den neueren Technologien in diesem Bereich zählt, wächst deren Verbreitung und Weiterentwicklung stetig. Initial von Google entwickelt wurde Ende 2018 das erste Stable-Release veröffentlicht, mit der Plattform-Unterstützung für iOS und Android. Inzwischen gibt es den Support für Web (beta), macOS (alpha) sowie Windows und Linux (technical preview). Seit Beginn stehen neben der Anforderung an einer hohen Geschwindigkeit der App folgende Probleme bei der Entwicklung von mobilen Applikationen mit mehreren Plattformen im Fokus der Flutter-Entwickler, welche es zu lösen gilt (Biessek, 2020; flutter.dev, 2020a, 2020i):

- **Lange / teurere Entwicklungszyklen** - Um auf die wechselten Anforderung des Marktes reagieren und diese innerhalb einer gewissen Frist anbieten zu können, muss man sich entweder auf eine Plattform fokussieren oder mehrere Teams beschäftigen. Diese Entscheidung hat Auswirkungen auf Kosten, die Anzahl an einzuhaltenden Fristen und unterschiedliche technische Möglichkeiten abhängig von der Plattform.

- **Verschiedene Programmiersprachen** - Damit ein Entwickler für mehrere Plattformen nativ entwickeln kann, benötigt er Wissen und Erfahrung in der jeweiligen Programmiersprache der Plattform. Das hat einen erheblichen Einfluss auf die Produktivität und steigert die Anforderungen an potentielle Entwickler.

- **Lange build/compile Zeiten** - Lange build/compile Zeiten haben einen großen Einfluss auf die Produktivität eines Entwicklers. Je nach Plattform und Größe des Projektes entstehen Wartezeiten, welche sowohl Zeit als auch Fokus des Entwicklers beeinträchtigen können.

- **Nachteile von bestehenden Cross-Plattform-Lösungen** - Durch den Einsatz von bestehenden Cross-Plattform-Technologien kann es zu Einbußen von Performance, Design oder User Experience kommen.

Bestehende Technologien für Cross-Plattform Entwicklung setzten für die Darstellung der Benutzeroberfläche entweder auf eine Webview oder sogenannten Original Equipment Manufacturer (OEM)-Widgets. Diese OEM-Widgets ermöglichen die Nutzung von nativen UI Komponenten der jeweiligen Plattform, jedoch ist man dadurch von den Schnittstellen der Hersteller für die Darstellung der Komponenten abhängig. Zusätzlich müssen diese Schnittstellen vermehrt angesprochen werden, was zu einem Engpass in der Applikation führen kann, da es einen zusätzlichen Schritt benötigt um die Oberfläche dem User darzustellen. Flutter hingegen setzt auf eine eigene Render-Engine für die Darstellung der Benutzeroberfläche mit dem Fokus auf eine hohe Geschwindigkeit (mehr dazu im Kapitel 3.1). Um die hohen Anforderungen an Design und Performance sicherstellen zu können setzt Flutter zudem auf Dart, eine Programmiersprache entworfen und entwickelt von Google (nähere Details dazu in Kapitel 3.4). Außerdem setzt Flutter auf einen Ahead of time compiler (AOT) um nativen Code zu erzeugen. Die daraus resultierende Applikation benötigt keine spezielle Umgebung in welcher der Dart Code interpretiert werden kann, was dazu führt, dass sie wie eine native Applikation am Gerät läuft und die Startzeit verringert bzw. die allgemeine Performance verbessert wird (Biessek, 2020).

Als Open-Source entwickelt und vorangetrieben durch Google wächst die Größe der Community hinter Flutter stetig. Dabei werden sowohl Bugfixes, Weiterentwicklungen und Dokumentationen für die Flutter SDK selbst aus der Community eingebracht, als auch zusätzliche Packages und Libraries. In Hinblick auf die Zukunft bietet Flutter zudem einen Ausblick auf zukünftige Weiterentwicklungen des mobilen Betriebssystems von Google, da Dart bzw. Flutter der native Ansatz für die Entwicklung von Applikationen für den geplanten Nachfolger von Android ist - Fuchsia (Biessek, 2020).

3.1 Render Engine und Plattform Features

Eine Eigenschaft, welche Flutter von den meisten anderen Cross-Plattform Technologien unterscheidet, ist die Darstellung der Benutzeroberfläche über eine eigene Render Engine. Dabei werden Komponenten weder über OEM-Widgets noch über eine Webview am Gerät dargestellt, sondern über die Skia Engine (auch entwickelt von Google) direkt gezeichnet. Das ermöglicht einen Geschwindigkeitsvorteil in der Darstellung, da Flutter damit keine zusätzlichen bzw. externen Aufrufe von Schnittstellen benötigt. Diese Methode, die Komponenten direkt auf einen Canvas zu zeichnen, minimiert dazu die Abhängigkeiten von Herstellern und Betriebssystemen und den damit verbundenen Regeln und Einschränkungen (vergleichbar mit aktuellen Spiele-Engines). Dabei gibt es die Möglichkeit mit den bereitgestellten Material (Android) oder Cupertino (iOS) Widgets den Designrichtlinien der Plattformen zu folgen, aber auch anhand der flexiblen Widgets-API eigene Designs und Komponenten ohne Performance-Verlust zu entwickeln (mehr zu Widgets im Kapitel 3.2) (Biessek, 2020).

Sowohl bei Web-basierenden Technologien als auch bei jenen, welche OEM-Widgets verwenden, wird zentral auf eine Bridge gesetzt, welche für die Kommunikation zum Plattform SDK genutzt wird. Bei Web-basierenden Technologien wird diese Bridge nur für die Services genutzt (z.B. GPS, Kamera, etc.), da für die Darstellung und Abarbeitung der Benutzereingaben eine Webview des Systems verwendet wird. (siehe Abbildung 3.1). Bei Frameworks, welche auf OEM Widgets setzten (wie z.B. React Native), dient die Bridge als zentrale Kommunikationsschnittstelle zwischen der Applikation und der Plattform SDK. Über diese Schnittstelle werden sowohl die Services des Gerätes angesprochen, als auch jene für die Darstellung der nativen Komponenten mittels der OEM-Widgets (siehe Abbildung 3.1) (Biessek, 2020).

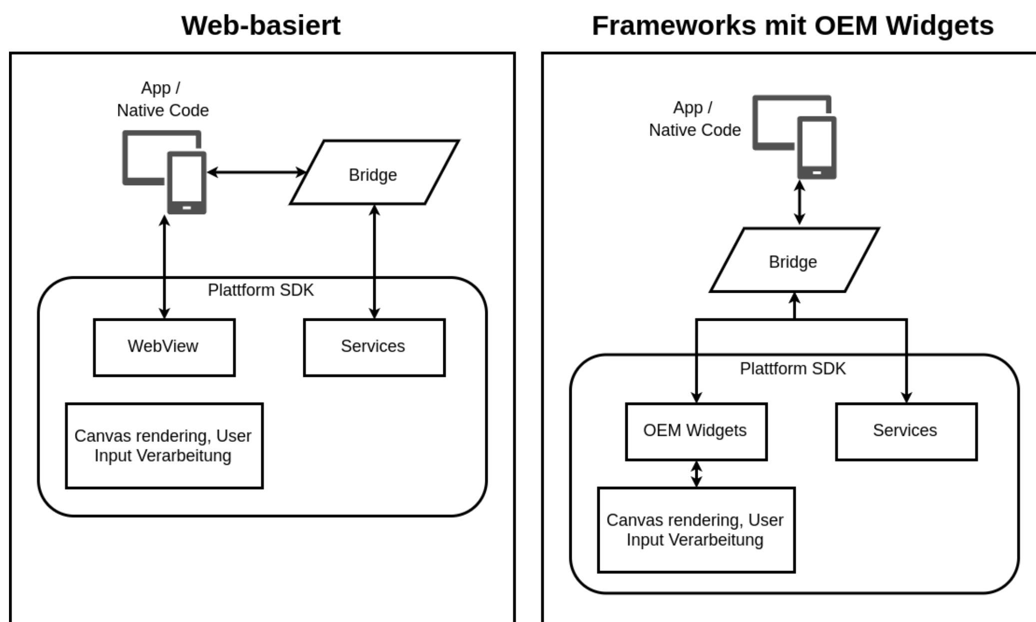

Abbildung 3.1: Aufbau Web und OEM-Widgets (Quelle: vgl. Biessek, 2020)

Flutter hingegen setzt auf eine eigene Render Engine für die Darstellung der Benutzeroberfläche, wodurch es nur den Zugriff auf einen Canvas des Systems benötigt um die Komponenten darstellen und auf Benutzereingaben und Gesten reagieren zu können. Für die Nutzung von gerätespezifischen Funktionen wie z.B. GPS und Kamera wird auf sogenannte Plattform-Channels gesetzt - vergleichbar mit einer Bridge bei herkömmlichen Ansätzen (siehe Abbildung 3.2). Durch diesen Ansatz ist es möglich, unabhängig der Plattform, eigene Designs und Animationen umsetzen zu können ohne Performance-Verlust oder Einschränkungen im Design durch Plattform-Komponenten. Im Gegensatz zu Technologien, welche auf OEM-Widgets setzten, entsteht durch diese Architektur kein Engpass bei der Schnittstelle zur Plattform-SDK durch die Darstellung der Benutzeroberfläche, wodurch eine höhere Bildrate und Geschwindigkeit zu Services sichergestellt werden kann (Biessek, 2020; Mainkar & Giordano, 2019).

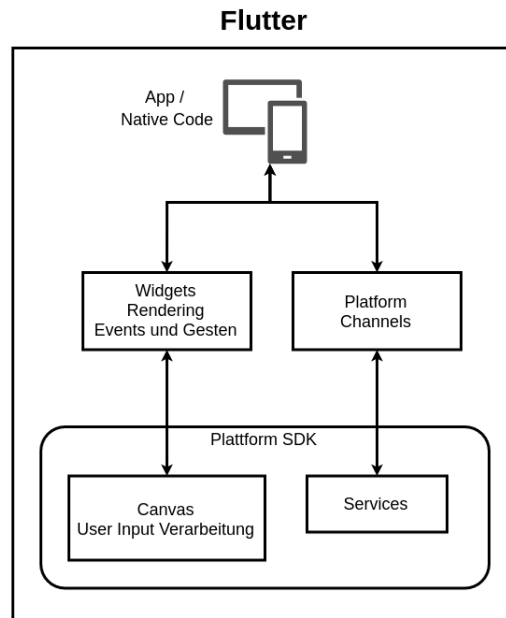

Abbildung 3.2: Aufbau Flutter (Quelle: vgl. Biessek, 2020)

3.2 Alles ist ein Widget

Widgets sind die Grundbausteine einer jeden Flutter Applikation, welche ein einheitliches und konsistentes Objektmodell innerhalb des Frameworks definieren. Ein sogenanntes Widget kann dabei ein strukturelles Element darstellen (z.B. einen Button), ein stilistisches Element (z.B. eine Schriftart), ein Layout (z.B. Padding) und vieles mehr. Im Hintergrund verwaltet Flutter diese Widgets in einem sogenannten Widget-Tree, welcher alle Widgets in einen hierarchischen Kontext setzt. Jedes Widget stellt dabei einen Knoten im Baum dar, welcher einen zugewiesenen Zustand besitzen kann. Dieser Zustand (oder im Englischen State) führt dazu, dass Flutter gezielt einzelne Knoten/Widgets erkennen kann, welche sich geändert haben. Dadurch ist es dem Framework möglich, nur jene Teile der Benutzeroberfläche neu zu zeichnen, welche sich verändert haben. Durch diese Herangehensweise kann das Framework die Anzahl der Veränderungen am Widget-Tree verringern, was die Anzahl der notwendigen Updates im UI optimiert und somit den Aufwand für das Rendering minimiert. Als Entwickler kann man diese Optimierung des Widget-Trees gezielt fördern, mehr dazu im Kapitel 3.5. Der Widget-Tree setzt sich dabei oft aus mehreren kleinen Widgets zusammen, welche alle eine einzelne bzw. spezielle Aufgabe besitzen. Durch die Komposition dieser Widgets ist es möglich, komplexere Anforderungen und Bausteine in einer Applikation als eigenes Widget abzubilden. In Flutter gibt es dazu die allgemeine Empfehlung hinsichtlich Widgets: Komposition über Vererbung. So ist z.B. das Container-Widget von Flutter ein sogenanntes Convenience-Widget, welches intern eine Komposition einer Vielzahl andere Widgets darstellt (für Positionierung, Größe, Aussehen, etc.). Obwohl der Slogan "Everything is a widget" auch in den

offiziellen Dokumentationen von Flutter zu finden ist, heißt es nicht, dass es keine anderen Objekte im Framework gibt. Vielmehr ist damit gemeint, dass die Bausteine für den Aufbau einer App aus Widgets bestehen (visuelle Repräsentation) (Windmill, 2019; flutter.dev, 2020g).

3.3 Stateful und Stateless Widgets

Wenn man in Flutter über Widgets spricht, so unterscheidet man grundsätzlich zwischen zwei Typen: Stateful- und Stateless-Widgets. Beide dieser Typen haben eines gemeinsam, sie besitzen eine sogenannte **build** Methode, in jener der Widget Tree definiert wird. Bei Stateless-Widgets handelt es sich dabei um Widgets, welche durch eine initiale Konfiguration definiert werden und sich dynamisch nicht ändern (wie der Name es schon beschreibt, hat dieses Widget keinen eigenen Zustand). Beispiele dafür wären Icon- oder Text-Widgets, welche einen Wert bekommen und anhand diesem den Inhalt darstellen (die build Methode wird nur aufgerufen, wenn sich die Konfiguration ändert, siehe dazu Abbildung 3.3). Im Gegensatz dazu haben Stateful-Widgets zusätzlich zu ihrer initialen Konfiguration einen eigenen Zustand, welcher es ihnen erlaubt sich dynamisch zu ändern. Das Widget besteht dazu aus zwei Klassen, dem eigentlichen Stateful-Widget und dem dazugehörigen State-Objekt (bekanntes Beispiel wäre eine CheckBox mit dem Zustand ob sie aktiv ist oder nicht). Durch das State-Objekt stehen dem Stateful-Widget zusätzliche Methoden des Lebenszyklus des Widgets zur Verfügung wie z.B. (Zammetti, 2019; Napoli, 2020):

Methode	**Beschreibung**
initState()	Wird aufgerufen, wenn das Widget in den Widget-Tree hinzugefügt wurde
dispose()	Wird aufgerufen, wenn das Widget aus dem Widget-Tree wieder entfernt wurde (kann z.B. genutzt werden um Ressourcen wieder frei zu geben)
didUpdateWidget()	Wird aufgerufen, wenn die Konfiguration des Widgets sich ändert
didChangeDependencies()	Wird aufgerufen, wenn sich das State Objekt vom Widget ändert
setState()	Signalisiert dem Framework, dass sich der Zustand des Widgets geändert hat

Tabelle 3.1: Lifecycle Methoden von Stateful-Widgets (Napoli, 2020)

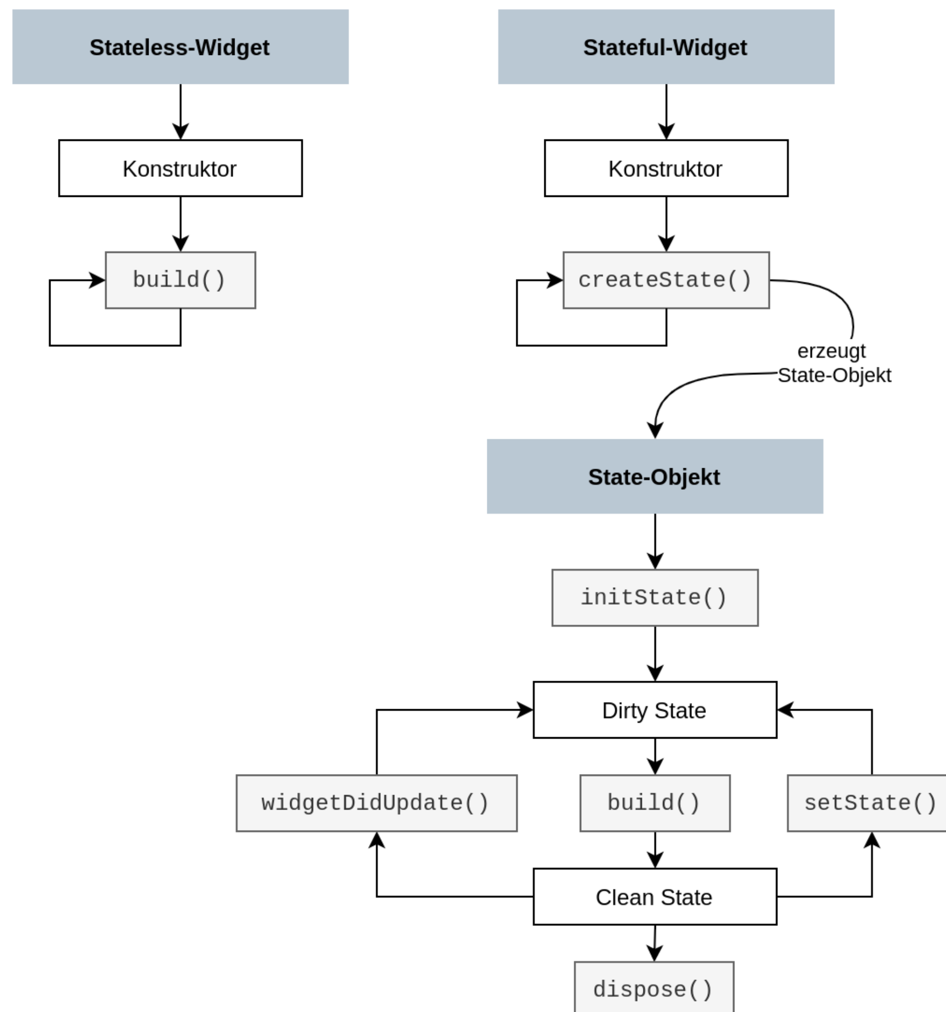

Abbildung 3.3: Widget Lifecycle (Quelle: vgl. Windmill, 2019)

3.4 Programmiersprache Dart

Dart ist eine objektorientierte Programmiersprache entwickelt und genutzt von Google (z.B. für Google AdWords). Seit der Veröffentlichung 2011 wird die Sprache für die Erstellung von Mobile-, Web-, Desktop- und Server-Applikationen verwendet und ist die verwendete Sprache für die Entwicklung von Applikationen mit Flutter. Von der Syntax ähnelt es bekannten Vertretern der Zeit (C#, Swift, Kotlin, Java sowie Javascript) und unterstützt im Kern die reaktive Programmierung. Folgende Vorteile von Dart haben unter anderem dazu geführt, dass Flutter darauf aufbaut: (Napoli, 2020):

- Dart unterstützt die Just-in-time-Kompilierung (**JIT**), was es ermöglicht, Änderungen am Code mittels Hot-Reload innerhalb kurzer Zeit dem Entwickler zur Verfügung zu stellen. In Kombination mit Flutter und dessen Stateful-Hot-Reload kann zusätzlich der aktuelle Zustand der Applikation bei Code-Änderungen beibehalten werden, was vor allem bei der Entwicklung von UI

Elementen einen iterativen Prozess ohne großen Zeitverlust ermöglicht (Entwickler bleibt z.B. auf der n-ten Unterseite der Applikation und kann eine Komponente anpassen ohne dass er bei jeder Änderung wieder auf diese Unterseite navigieren muss). Die JIT-Kompilierung wird dabei nur im Debug-Modus verwendet.

- Wird die App im Release-Modus gebaut, so wird ein Ahead-of-time-Kompiler (**AOT**) verwendet. Dieser erzeugt nativen Code, was die Notwendigkeit der Interpretation einer Programmiersprache in eine plattformspezifsche Sprache sowie die Verwendung einer Bridge eliminiert (und sich damit positiv auf die Performance auswirkt).

- Sowohl die Logik, als auch die Repräsentation in Form des UI kann mittels Dart geschrieben werden. Dadurch ist keine separate Sprache für die Benutzeroberfläche notwendig (wie z.B. Markup, XML, etc.).

- Aufbauend auf Dart kann Flutter eine Bildwiederholungsrate von 60 FPS sowie 120 FPS bei Geräten, welche 120Hz unterstützen, nutzen.

Konkurrierende Frameworks setzten gleich wie Flutter auf die Trennung von Funktionen in Komponenten und Flutter verleugnet nicht, dass das Framework stark durch bestehende Lösungen inspiriert wurde. Jedoch ist Flutter in einem Aspekt zur Gänze unterschiedlich - die Sprache für die Darstellung des UI und die der Businesslogik ist einheitlich Dart (siehe nachfolgende Tabelle 3.2) (Payne, 2019).

Framework	**Logik/Verhalten**	**Benutzeroberfläche**
Xamarin	C#	XAML
React Native	JavaScript	JSX
NativeScript	JavaScript	XML
Flutter	Dart	Dart

Tabelle 3.2: Sprachenunterschiede zwischen vergleichbaren Frameworks (Payne, 2019)

3.5 Best Practices der Technologie

Da es sich bei Dart um eine objektorientierte Programmiersprache handelt, gibt es natürlich allgemeine (technologieunabhängige) Best Practices. In diesem Kapitel werden aber nur jene, welche durch die Verwendung von Dart und Flutter zustande kommen, genauer betrachtet. Es handelt sich hierbei nicht um eine vollständige Auflistung, sondern lediglich um einen Auszug.

Empfohlene Vorgehensweisen bei der Entwicklung von Applikationen mit Flutter, welche schlussendlich auch den Endbenutzer positiv auffallen, betreffen Tipps und Tricks hinsichtlich der Performance. Grundsätzlich sind Flutter-Anwendungen standardmäßig performant, jedoch gibt es einige Tücken, welche man bei der Entwicklung beachten bzw. bei schlechter Performance für die Fehlerfindung heranziehen kann. Jedes Widget besitzt eine sogenannte **build** Methode, in welcher der Aufbau und die Struktur des anzuzeigenden Widgets definiert wird (siehe Kapitel 3.2 - Komposition von Widgets). Da diese Methode vermehrt aufgerufen werden kann bzw. aufgerufen wird (wenn sich z.B. Child-Widgets neu zeichnen oder man den State des Widgets updatet), sollte man darauf achten diese Methode so schlank wie möglich zu halten. Hat man z.B. eine rechenaufwändige Operation in der build-Methode, so kann das die Performance erheblich beeinträchtigen. Das Flutter-Framework kümmert sich um die performante Abwicklung der Darstellung der Widgets, jedoch kann man es dabei unterstützen, indem man zu große build-Methoden in einem Widget vermeidet. Spaltet man stattdessen das Widget in mehrere kleinere Widgets (mit jeweils einer eigenen build-Methode) auf, so kann das Framework diese besser verwalten bzw. entscheiden, welche upgedatet werden sollen und welche nicht. Wichtig hierbei ist, dass man die Inhalte wirklich in eigene Stateful oder Stateless Widgets auftrennt (siehe Kapitel 3.3). Strukturiert man z.B. Widgets nur durch die Aufteilung in Methoden, welche den Child-Tree zurück liefern, so hat das Framework keine Möglichkeit diese zu optimieren. Dieser Ansatz wird auch als Performance-Anti-Pattern bezeichnet (flutter.dev, 2020c).

Simples Beispiel für die Verwendung dieses Anti-Patterns:

```
1  class WidgetAntiPattern extends StatelessWidget {
2    @override
3    Widget build(BuildContext context) {
4      return Column(
5        mainAxisAlignment: MainAxisAlignment.center,
6        children: [
7          // Widget wird über eine Methode zum Tree hinzugefügt
8          _getLogoWithBorder()
9        ]);
10   }
11
12   Widget _getLogoWithBorder() {
13     return Container(
14       decoration: BoxDecoration(border: Border.all()),
15       child: FlutterLogo(),
16     );
17   }
18 }
```

Listing 3.1: Beispiel für ein Anti-Pattern bei der Erstellung von Widgets

Bessere Lösung durch die Erstellung eines eigenen Widgets mit zugehöriger build Methode:

```
class WidgetAntiPatternCorrected extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        // nun ist es ein eigenständiges Widget
        LogoWithBorder()
      ]);
  }
}

class LogoWithBorder extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(border: Border.all()),
      child: FlutterLogo(),
    );
  }
}
```

Listing 3.2: Beispiel für die Korrektur des Anti-Patterns bei der Erstellung von Widgets

Zusätzlich gibt es die Möglichkeit mit sogenannten const-Konstruktoren Widgets zu definieren, welche nicht von dynamischen Inhalten des Parents abhängig sind. Dadurch kann man dem Framework explizit mitteilen, dass dieses Widget bei Aktualisierungen gesondert behandelt werden kann. Seit Dart 2 ist das Keyword **new** bei der Erstellung von Objekten optional, jedoch muss **const** explizit angegeben werden, wenn ein const-Konstruktor verwendet werden soll (dart.dev, 2020b).

3.6 Testing in Flutter

Je mehr Funktionen eine Applikation hat bzw. je komplexer dessen Prozesse sind, desto aufwändiger ist es, diese Funktionen oder Prozesse manuell zu testen. Automatisierte Tests unterstützen dabei die Qualität und Funktionsfähigkeit der Applikation sicherzustellen. Ein hoher Grad an automatisierten Tests erleichtert zudem die Überprüfung der korrekten Funktionsfähigkeit bei Änderungen oder neuen Funktionalitäten ohne erheblichen Aufwand (z.B. durch Integration in Continuous Integration bzw. Continuous Delivery)(flutter.dev, 2020h).

In Flutter unterscheidet man bei automatisierten Tests zwischen drei Kategorien, welche sich wie folgt unterscheiden (dart.dev, 2020b):

- **Unit Test** - Testet eine einzelne Methode oder Klasse. Ziel dieser Tests ist die Überprüfung einer Logik unter verschiedenen Umständen. Abhängigkeiten (z.B. zu API-Endpunkten) werden üblicherweise gemocked und es wird kein UI getestet.

- **Widget Test** - Testet ein einzelnes Widget (wird in anderen Frameworks oft Component Test bezeichnet). Ziel dieser Tests ist die Überprüfung, ob ein Widget alle bzw. die richtigen Werte anzeigt sowie korrekt auf Benutzereingaben reagiert.

- **Integration Test** - Testet einen größeren Teil der App oder die gesamte App. Bei diesem Test überprüft man, ob alle Widgets und Services wie gewünscht miteinander interagieren und funktionieren. Außerdem kann ein Integration Test für Performance-Analysen herangezogen werden.

Da die Erstellung der Tests aber auch einen Aufwand darstellt, muss abgewogen werden, was für das aktuelle Projekt bzw. Feature sinnvoll ist und was nicht. Die unterschiedlichen Typen an automatisierten Tests haben dabei jeweils ihre eigenen Kompromisse (siehe nachfolgende Tabelle 3.3)(flutter.dev, 2020h).

	Unit	**Widget**	**Integration**
Vertrauen	niedrig	höher	am höchsten
Wartungskosten	niedrig	höher	am höchsten
Abhängigkeiten	wenige	mehr	am meisten
Geschwindigkeit	schnell	schnell	langsam

Tabelle 3.3: Kompromisse bei unterschiedlichen Testtypen (flutter.dev, 2020d)

3.7 Relevanz und Verbreitung

Trotz der Neuartigkeit des Frameworks wird Flutter bereits jetzt von einigen großen Unternehmen produktiv eingesetzt. Die Möglichkeit qualitativ hochwertige Applikationen mit nativer Geschwindigkeit für mehrere Plattformen in einer einheitlichen Sprache entwickeln zu können führt immer häufiger dazu, dass sowohl renommierte Unternehmen als auch Startups auf Flutter setzen. Im Showcase des Frameworks befinden sich bereits die Apps für AdWords von Google, den Marketplace von Alibaba, die Steuerungsapp für Leuchtkörper von Philips hue und viele mehr (Windmill, 2019; flutter.dev, 2020d).

4 Deklarative UI

Um Flutter und den Begriff deklarative UI besser verstehen zu können, muss man ihn vom Programmierparadigma aktueller nativer Entwicklungswerkzeuge (Android SDK oder iOS UIKit) abgrenzen. Dabei unterscheidet man in diesem Beispiel zwischen imperativer und logischer (bzw. deklarativer) Programmierung. Bei der imperativen Entwicklung geht es darum zu definieren **WIE** etwas funktionieren soll, bei der deklarativen darum **WAS** gemacht werden soll. Dieser Unterschied zeigt sich am besten anhand eines einfachen Beispiels (Sebesta, 2016):

```
final items = [1, 2, 3, 4, 5, 6, 7];

// imperative
List<int> resultsImperative = [];
items.forEach((item) {
  if (item % 2 == 0) resultsImperative.add(item);
});

// declarative
List<int> resultDeclarative =
  items.where((item) => item % 2 == 0).toList();
```

Listing 4.1: Imperative vs deklarative Programmierung

Im Kontext der Benutzeroberfläche ergibt sich daraus, dass mit einem deklarativen UI das Was beschrieben wird und nicht das Wie (z.B. wie in HTML). Obwohl dieses Paradigma nicht neu ist, hat Google mit Flutter diesen Ansatz neu entfacht. Durch das positive Feedback aus der Community haben auch andere Frameworks bzw. Hersteller darauf reagiert - z.B. iOS mit SwiftUI und Android mit Jetpack Compose. Beide dieser Frameworks ermöglichen den deklarativen Ansatz in der Erstellung der Benutzeroberfläche für native Applikationen (Jetpack Compose ist zum Erstellungszeitpunkt dieser Arbeit aber erst in der Developer Preview)(Barker, 2020; developer.android, 2020).

Die hohe Abstraktion durch den Einsatz deklarativer UI in Flutter ermöglicht es dem Entwickler sich auf die eigentliche Darstellung und die Funktionalität zu fokussieren und das Zuweisen bzw. Updaten der Komponenten und dessen Optimierung dem

Framework zu überlassen. Da die Benutzeroberfläche sich stets aus dem vorhandenen Zustand ergibt, ist es nicht notwendig, einzelne Bestandteile des UI explizit anzupassen oder anzusprechen. Vielmehr wird der zugehörige Zustand verändert und anhand der build-Methode die Änderungen vorgenommen bzw. der Screen neu gezeichnet (siehe Abbildung 4.1). Ein nennenswerter Vorteil bei diesem Ansatz ist die Eigenschaft, dass sich Änderungen bzw. der Inhalt einer Benutzeroberfläche nur aus dem zugehörigen Zustand ergibt (single point of truth).(Kifer & Liu, 2018; flutter.dev, 2020f)

UI = f(state)

aktuelle Benutzeroberfläche
build Methode
aktueller Zustand

Abbildung 4.1: Ansatz des deklarativen UI (Quelle: vgl. flutter.dev, 2020d)

5 Reaktive Programmierung

Bei dem Begriff der reaktiven Programmierung handelt es sich um ein weiteres Programmierparadigma, welches sich unveränderbare (immutable) Streams und der Verbreitung von Änderungen (Propagation of Change) bedient. Die durch die Verwendung von reaktiver Programmierung einhergehende Asynchronität spiegelt dabei oft die Anforderungen moderner Applikationen wider. Asynchrone Streams (oder in diesem Kontext auch oft als Observable bezeichnet) ähneln dabei stark Futures, wobei bei einem Stream nicht nur ein Wert zurückgeliefert werden kann. Ein Stream stellt dabei eine Sequenz von Daten (oder Events) in einer zeitlichen Hierarchie dar. Bei der Umsetzung wird dabei auf das Observer-Pattern zurückgegriffen, welches aus Publisher, Subscriber, Subscription und Messages besteht. Der Publisher ist dabei diejenige Komponente, welche Daten zur Verfügung stellt und die Subscriber jene, welche an den Daten interessiert sind. Über die Subscription (Verbindung zwischen Publisher und Subscriber) erhalten die Subscriber die propagierten Messages (auch als Events oder Daten bezeichnet). Die Verwendung dieses Ansatzes reicht dabei von einzelnen Bausteinen oder Funktionalitäten einer Applikation bis hin zu ganzen Systemen (hier spricht man dann von reaktiven Systemen). Die Funktionen und Vorteile eines solchen reaktiven Systems wurden im Reactive Manifesto beschrieben, mehr dazu im Kapitel 5.1 (Urma, Fusco & Mycroft, 2018).

5.1 Reactive Manifesto

Das Reactive Manifesto wurde 2013/14 erstellt und formuliert Grundprinzipien für die Erstellung reaktiver Applikationen und Systeme. Laut der Definition sind solche Systeme flexibel, skalierbar, lose gekoppelt und reagieren sehr gut auf Fehler. Dabei stützen sie sich auf vier Grundprinzipien, welche zusammenhängen bzw. voneinander abhängig sind (siehe Abbildung 5.1)(Bonér, Farley, Kuhn & Thompson, 2014):

- **Responsive** (Antwortbereit) - Sofern möglich, antwortet das System unter allen Umständen zeitgerecht. Auch bei Fehlern ist eine entsprechende Antwort zu liefern und eine Antwortzeitgrenze zu definieren, sonst ist eine Erkennung und Behandlung von Fehlern nicht möglich. Eine konsistente Antwortzeit dient als Zeichen von Qualität und bildet Vertrauen.

- **Resilient** (Widerstandsfähig) - Selbst bei Ausfällen von Hard- und Software bleibt das System verfügbar und antwortbereit. Ein Ausfall eines Teilsystems soll auf jenes begrenzt sein.

- **Elastic** (Elastisch) - Auch unter verschiedenen Lastbedingungen bleibt das System einsatzbereit. Das System darf dabei keine Engpässe aufweisen, um Aufgaben auf beliebig viele Ressourcen verteilen zu können.

- **Message driven** (Nachrichtenorientiert) - Asynchrone Nachrichtenübermittlung wird zum Zweck der Entkoppelung zwischen Komponenten verwendet (ermöglicht transparente Skalierung - kein Unterschied, ob es auf einem Gerät oder in einem verteilten Netzwerk verwendet wird).

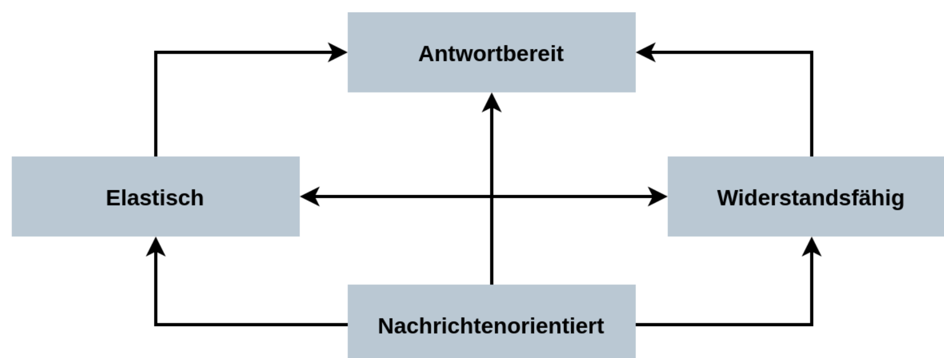

Abbildung 5.1: Reactive Manifesto (Quelle: vgl. Bonér, Farley, Kuhn Thompson, 2014)

5.2 Flutter als reaktives Framework

Streams sind ein grundlegender Bestandteil von Dart und ermöglichen damit eine reaktive Programmierung durch die Dart API. Wer bereits Erfahrung mit ReactiveX hat, kann auch auf RxDart zurückgreifen. Dabei handelt es sich um ein Package, welches zusätzliche Funktionen rund um Observables/Streams der ReactiveX Library für Dart bereitstellt. Da dieses Package aber nicht zwingend notwendig ist, wird in den nachfolgenden Erklärungen und Beispielen nur auf die Streams der Dart API eingegangen(Napoli, 2020; Windmill, 2019).

5.2.1 Erstellung von Streams

Es gibt dabei drei verschiedene Arten in Dart Streams zu erstellen: Transformation bestehender Streams, Erstellung eines Streams von Grund auf mittels *async** Funktion oder die Erstellung mittels StreamController (dart.dev, 2018).

5.2.1.1 Transformation

Wenn man bereits einen Stream hat und dessen Werte nur in einer anderen Form benötigt, bietet sich der Einsatz von sogenannten Transformationen an. Dadurch ist es möglich, aufbauend auf einen Source-Stream, einen neuen und angepassten Stream zu erstellen ohne den ursprünglichen zu verändern. Die Dart API stellt dabei Methoden für die Transformation zur Verfügung wie z.B. *map()*, *where()*, *expand()* oder *take()*. Eine beispielhafte Anwendung wird im nachfolgenden Listing 5.1 beschrieben (dart.dev, 2018):

```
1  // Erzeugt einen Stream, welcher 15 Sekunden lang
2  // jede Sekunde einen zunehmenden int liefert (0,1,2,3,...)
3  Stream<int> secStream = Stream<int>
4       .periodic(Duration(seconds: 1), (x) => x)
5       .take(15);
6
7  // Transformiert den Stream, dass der Wert immer
8  // verdoppelt wird (0,2,4,6,..)
9  Stream<int> doubleSecStream = secStream.map((x) => x * 2);
10
11 // Transformation mit mehreren Schritten (0,0,2,2,4)
12 Stream<int> specialSecStream = secStream
13      .where((x) => x.isEven) // nur gerade Zahlen
14      .expand((x) => [x, x])  // verdoppelt jedes Event
15      .take(5);               // nimmt nur die ersten 5 Events
```

Listing 5.1: Anwendung von Transformationen auf Streams

Für komplexere oder öfters benötigte Transformationen gibt es zusätzlich die Möglichkeit sogenannte StreamTransformer zu definieren, welche mit der *transform()* Methode auf Streams angewandt werden können. Dart bietet über die *dart:convert* Library bereits einige solcher StreamTransformer an, welche genutzt werden können (dart.dev, 2018):

```
1  Stream<List<int>> content = File('file.txt').openRead();
2  List<String> lines = await content
3       .transform(utf8.decoder)     // Decoder für UTF8
4       .transform(LineSplitter())   // Teilt einen String in
5                                    // seine Zeilen auf
6       .toList();
```

Listing 5.2: Anwendung von StreamTransformer auf Streams

5.2.1.2 async* Funktion

Eine weitere Möglichkeit einen Stream in Dart zu erstellen, ist über eine *async** Funktion. Dabei wird der Stream erstellt sobald die Funktion aufgerufen und gestartet nachdem auf den Stream zugegriffen wird (z.B. durch *listen()*). Innerhalb der Funktion kann über ein *yield* Statement ein Event gesendet werden und der Stream wird beendet sobald das Ende der Funktion erreicht wird. Mit diesem Ansatz würde das Beispiel aus Kapitel 5.2.1.1 folgendermaßen umgesetzt werden (dart.dev, 2018):

```
// Erzeugt einen Stream, welcher 15 Sekunden lang
// jede Sekunde einen zunehmenden int liefert (0,1,2,3,...)
Stream<int> getSecStream() async* {
  int i = 0;
  while (i < 15) {
    await Future.delayed(Duration(seconds: 1));
    yield i++; // hier wird das Event geschickt
  }
}
```

Listing 5.3: Erstellung eines Streams mittels async* Funktion

5.2.1.3 StreamController

Können Events an mehreren Punkten in der Applikation auftreten oder möchte man mehr Kontrolle über den Stream, so ist ein StreamController zu verwenden. Dieser kapselt die notwendigen Funktionen und beinhaltet alle notwendigen Methoden für die effiziente Nutzung von Streams. Der StreamController besitzt dabei die Variablen *sink* für den Input und *stream* für den Output. Soll auf einen Stream mehrfach zugegriffen werden können (Standard erlaubt nur einen Listener), kann der Broadcast-Konstruktor verwendet werden. Das folgende Beispiel soll vereinfacht die Nutzung demonstrieren (dart.dev, 2018; Napoli, 2020):

```
StreamController<int> controller = StreamController<int>();

// Events zum Stream hinzufügen
controller.add(1);
controller.sink.add(2); // beide Varianten sind ident

// auf den Stream zugreifen
controller.stream.forEach(print);
```

Listing 5.4: Verwendung von StreamController

5.2.2 Verwendung von Streams

Um die Daten aus Streams zu nutzen und in der Benutzeroberfläche dem User anzeigen zu können, bietet sich das StreamBuilder-Widget des Flutter Frameworks an. Man könnte auf neue Events des Streams händisch reagieren und anhand der *setState* Methode das UI updaten, mit dem StreamBuilder-Widget wird jedoch der Aufwand minimiert und die Performance verbessert (es wird nur jener Teil neu gezeichnet, der durch den StreamBuilder erstellt wird). Das Widget hört automatisch auf neue Events vom Stream und zeichnet seine Widgets dementsprechend neu. Dem Widget kann zusätzlich ein initialer Wert übergeben werden, welcher genutzt wird bevor der Stream das erste Event liefert. Im *builder*-Callback bekommt man als Parameter einen Snapshot des Events. Dieser beinhaltet neben den eigentlichen Daten potentielle Errors sowie einen ConnectionState (Status des Streams - none, waiting, active und done). Das resultierende Ergebnis ist dabei ein reaktives Widget, welches einen Stream als Input bekommt und selbst für die Darstellung verantwortlich ist (praktische Anwendung von reaktiver Programmierung). Das nachfolgende Beispiel soll demonstrieren wie ein StreamBuilder genutzt werden kann um die Werte aus dem Stream vorherigen Kapiteln in einem Textfeld anzuzeigen (Napoli, 2020; Windmill, 2019):

```
1  @override
2  Widget build(BuildContext context) {
3    return Scaffold(
4        body: Center(
5            child: StreamBuilder<int>(
6              initialData: 0,      // initialer Wert
7              stream: secStream,   // Stream der verwendet wird
8              builder: (context, snapshot) {
9                // Im builder hat man Zugriff auf den Wert des
                     Streams
10               return Text(snapshot.data.toString());
11             },
12           )
13       )
14   );
15 }
```

Listing 5.5: Verwendung des StreamBuilders

6 State Management

Die oberflächlichste Beschreibung für State bzw. State Management ist die Speicherung und Verwaltung von allem, was im Speicher vorhanden ist, während eine Applikation ausgeführt wird bzw. alle Informationen, welche notwendig sind, um die Benutzeroberfläche mit den korrekten Daten anzeigen zu können. Da eine solch grobe Beschreibung für die Entwicklung keinen erheblichen Mehrwert liefert, wird in den folgenden Kapiteln genauer auf dieses Thema eingegangen sowie Ansätze für Flutter genauer betrachtet und beschrieben (flutter.dev, 2020b).

6.1 Allgemeines zu State Management

Wenn man über State Management oder allgemein über State spricht, unterscheidet man je nach Plattform oder Technologie oft zwischen unterschiedlichen Kategorien und Bezeichnungen. Die Grundidee und zugrundeliegende Notwendigkeit ist dabei meist ident: Jede Applikation besitzt State und gewisse Anforderungen oder die Komplexität einer Applikation erfordert, dass man sich mit der Verwaltung dieses States auseinandersetzt. Klassische Beispiele hierbei wären der Warenkorb eines Online-Shops oder der Gelesen-Status einer Nachrichten-App. Dabei gibt es keine allgemeingültige Lösung oder eine Library, welche alle Aspekte abdeckt, sondern Lösungsansätze für spezifische Problemstellungen. Je nach Anforderungen, Vorwissen im Team und technologischen Abhängigkeiten können Lösungen variieren und dies sollte beim Design einer Applikation berücksichtigt werden. Im Kontext von Flutter unterscheidet man im Groben dabei zwischen dem Ephemeral State (Kapitel 6.1.1) und dem App State (Kapitel 6.1.2). Hierbei ist zu beachten, dass es keine eindeutige, universelle Regel gibt, wann ein State zu welchem Typ gehört. So beginnt man beispielsweise mit einem eindeutig Ephemeral State, wenn die Applikation jedoch an Funktionen zunimmt, muss dieser möglicherweise in den App State verschoben werden (flutter.dev, 2020b; Savkin, 2017).

6.1.1 Ephemeral State

Ephemeral State (auch genannt UI State oder Local State) ist jener Zustand, welcher üblicherweise innerhalb eines Widgets oder einer Komponente gehalten werden

kann. Auf diese Art von State muss außerhalb des Widgets nicht zugegriffen werden können und damit ist es nicht notwendig, diesen mittels State Management Ansatz zu verwalten. Beispiele hierfür wären der aktuelle Index einer *BottomNavigationBar*, der Fortschritt einer Animation oder die aktuelle Seite einer *PageView*. Für solche Anwendungsfälle kann auf ein einfaches *StatefulWidget* und dessen *setState()* Methode zurückgegriffen werden (siehe Beispiel aus Listing 6.1) (flutter.dev, 2020b).

```
class MyHomepage extends StatefulWidget {
  @override
  _MyHomepageState createState() => _MyHomepageState();
}

class _MyHomepageState extends State<MyHomepage> {
  int _index = 0;

  @override
  Widget build(BuildContext context) {
    return BottomNavigationBar(
      currentIndex: _index,
      onTap: (newIndex) {
        setState(() {
          _index = newIndex;
        });
      },
      // .. items ...
    );
  }
}
```

Listing 6.1: Ändern des Zustandes mittels *setState()*

6.1.2 App State

Handelt es sich nicht um einen Ephemeral State, so spricht man in Flutter von einem App State oder Shared State. Dieser wird nicht nur im Widget selbst gehalten, sondern mit mehreren Teilen der Applikation geteilt und kann auch Sessionsübergreifend persistiert werden. Beispiele hierfür wären Benutzereinstellungen, Login-Informationen und Tokens sowie Benachrichtigungen innerhalb der App. Für diese Art von State gibt es keine allgemeingültige Herangehensweise, vielmehr muss anhand der Anforderungen und Komplexität sowie vielen anderen Einflussfaktoren eine geeignete State Management Variante gewählt werden. Mit der Frage, welche Ansätze es gibt und für welche Anwendungsfälle diese geeignet sind, befasst sich diese Arbeit und die nachfolgenden Kapiteln (flutter.dev, 2020b).

6.2 State Management Varianten in Flutter

State Management ist ein komplexes Thema und in Flutter ist das keine Ausnahme. Dabei wird vom Framework kein Ansatz gegenüber einem anderen bevorzugt, vielmehr geht es darum, die beste Lösung für ein vorhandenes Problem zu finden und es dem Entwickler dabei offen zu lassen, wie er es lösen möchte. Man hat die Möglichkeit auf Werkzeuge des Frameworks zurückzugreifen, welche in dessen Kern mit ausgeliefert werden, sowie Packages von Dritten zu verwenden oder auf eine eigene Implementierung zu setzen bzw. auf ein Pattern aufzubauen. Einige bekannte Lösungsvarianten aus anderen Plattformen gibt es bereits als Package, wie z.B. Redux (Kapitel 6.2.4) oder MobX (Kapitel 6.2.6). In den nachfolgenden Kapiteln werden relevante Vertreter aller drei Kategorien (Framework, Package und Pattern) analysiert, um für die Erstellung der Prototypen in Kapitel 9 das notwendige Hintergrundwissen zu schaffen. Die Einstufung der Relevanz bzw. die Auswahl erfolgt anhand der Erwähnungen auf der offiziellen flutter.dev Webseite sowie der Verbreitung auf pub.dev/GitHub und stellt keine vollständige Liste dar (Windmill, 2019).

In deklarativen Frameworks wie Flutter gilt grundsätzlich: Halte den Zustand über den Widgets bzw. Komponenten, welche ihn verwenden. Möchte man die Benutzeroberfläche mit den aktuellen Daten aktualisieren, so muss man sie neu bauen. Es widerspricht den Ansätzen des Frameworks, wenn man versucht, imperativ von außen den Zustand eines Widgets über eine Methode zu verändern (z.B. über *MyWidget.updateValue(newValue)*). In Flutter wird dazu der Konstruktor des Widgets verwendet und somit jedes Mal ein neues Widget erzeugt, wenn sich der Inhalt von außen ändert, was dazu führt, dass der dazugehörige Zustand sich im Parent oder höher befinden muss. Durch diesen Top-down-Datenfluss muss sich ein Child-Widget nicht um den State kümmern, da es die notwendigen Informationen vom Parent mitbekommt und somit jederzeit neu gezeichnet werden kann (vereinfachtest Beispiel für den Zustand eines Warenkorbs siehe Abbildung 6.1) (flutter.dev, 2020e).

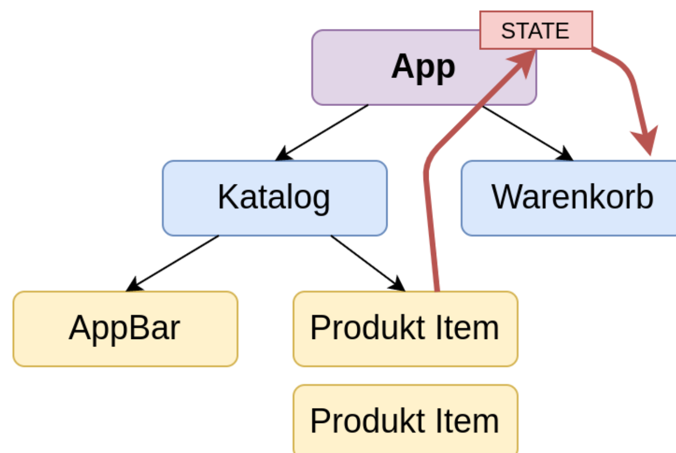

Abbildung 6.1: Beispiel für Lift state up (Quelle: vgl. flutter.dev, 2020e)

6.2.1 StatefulWidget's setState

Bei der *setState* Methode handelt es sich um den Grundbaustein eines jeden Stateful-Widgets (siehe dazu Kapitel 3.3). Diese Methode kann dazu verwendet werden State innerhalb einer Komponente zu verwalten. Vor allem bei simplen Zuständen und jenen, welche außerhalb des Widgets nicht benötigt werden, bietet es sich an, auf diese Methode zurückzugreifen, da keine externe Library oder ein komplexeres Setup notwendig ist. Sollte sich der Inhalt eines Zustandes im State-Objekt des Widgets ändern, so kann man durch den Aufruf der *setState* Methoden dem Framework mitteilen, dass das Widget neu gezeichnet werden soll. Zu beachten ist hierbei, dass stets das gesamte Widget innerhalb der build Methode neu gezeichnet wird, was bei komplexeren bzw. größeren Komponenten oft nicht notwendig ist und sich negativ auf die Performance auswirken kann (Cheng, 2019).

Möchte man bei diesem Ansatz dennoch auf den Zustand bzw. auf Interaktionen zugreifen (z.B. wenn eine Checkbox ausgewählt wird), so kann man in Dart auf Callback-Funktionen zurückgreifen. Diese werden wie Objekte behandelt und können somit wie Variablen übergeben werden (Beispiel siehe Listing 6.2). Für simple Aufgaben ist das ein gangbarer Weg, jedoch wird in Hinblick auf ein globales State-Management dieser Ansatz bei größeren Applikationen sehr aufwändig und unübersichtlich. Aus diesem Grund bietet Flutter die Möglichkeit, Daten und Services im Widget-Tree allen Nachfolgern (also allen Children und dessen Children) strukturiert zur Verfügung zu stellen (mehr dazu im Kapitel 6.2.2) (flutter.dev, 2020e).

```
1  // State Objekt des Parent Widgets
2  class _CallbackUsageState extends State<CallbackUsage> {
3    bool _isSelected = false;
4
5    @override
6    Widget build(BuildContext context) {
7      return Column(mainAxisAlignment: MainAxisAlignment.center
         , children: [
8        Text(_isSelected ? 'selected' : 'not selected'),
9        CustomCheckbox(
10         isSelected: _isSelected,
11         selectedChanged: (selected) {
12           // setState durch Callback Funktion
13           setState(() => _isSelected = selected);
14         },
15       )
16     ]);
17   }
18 }
19
```

```
20 // Child Widget welches den Zustand verändert
21 class CustomCheckbox extends StatelessWidget {
22   final bool isSelected;
23   final Function(bool selected) selectedChanged;
24 
25   const CustomCheckbox({this.isSelected, this.selectedChanged
       });
26 
27   @override
28   Widget build(BuildContext context) {
29     return Checkbox(
30       value: isSelected,
31       // Callback Funktion verwenden
32       onChanged: (value) => selectedChanged(value),
33     );
34   }
35 }
```

Listing 6.2: Verwendung von Callback Funktionen - Lifting State up

6.2.2 InheritedWidget und InheritedModel

Neben dem *StatelessWidget* und dem *StatefulWidget* gibt es im Flutter Framework noch einen weiteren Typ von Widget - das *InheritedWidget*. Wenn ein Widget auf Daten eines Parent-Widgets zugreifen möchte (wie in Kapitel 6.2.1 beschrieben), so können die benötigten Daten bzw. Callbacks über Konstruktoren bis hin zum Child weitergereicht werden. Da dieser Ansatz aber bei größeren bzw. komplexeren Komponenten sehr aufwändig ist und die Wartbarkeit oder Erweiterbarkeit negativ beeinträchtigt, kann auf ein *InheritedWidget* zurückgegriffen werden. Durch die Verwendung dieses Widget-Typen ist es möglich, Daten und Services allen Child-Widgets zur Verfügung zu stellen, ohne die Notwendigkeit die Werte über die Konstruktoren weiterreichen zu müssen. Durch die Methode *inheritFromWidgetOfExactType* kann auf die nächstgelegene Instanz des angefragten Typs von *InheritedWidget* zugegriffen werden. Dabei wird ausgehend vom aktuellen Widget der Widget-Tree nach oben hin durchgesucht, was bedeutet, es können mehrere Instanzen des gleichen Typen existieren und es wird nur der nächstgelegene zurückgeliefert. Ein bekanntes Beispiel aus dem Framework, welches auf diese Methode setzt, ist die Theme Klasse. Mit *Theme.of(context)* (of() ist eine Hilfsmethode um den Zugriff zu erleichtern) kann dabei auf das nächste Theme Objekt im Widget-Tree zugegriffen werden. Ein exemplarisches Beispiel ist in Listing 6.3 zu finden (Biessek, 2020).

```
1  // Das InheritedWidget mit den benötigten State Informationen
2  class ImportantInfo extends InheritedWidget {
3    final ImportantInfoContent infoContent;
4
5    const ImportantInfo({
6      this.infoContent,
7      @required Widget child,
8    }) : super(child: child);
9
10   static ImportantInfoContent of(BuildContext context) =>
11       context.dependOnInheritedWidgetOfExactType<
           ImportantInfo>().infoContent;
12
13   @override
14   bool updateShouldNotify(ImportantInfo old) => infoContent
        != old.infoContent;
15 }
16
17 class InheritedParent extends StatelessWidget {
18   @override
19   Widget build(BuildContext context) {
20     // Das InheritedWidget wird im WidgetTree hinzugefügt
21     return ImportantInfo(
22       // Daten welche im InheritedWidget geahlten werden
23       infoContent: ImportantInfoContent(version: '1', info: '
           Test'),
24       child: InheritedChild(),
25     );
26   }
27 }
28
29 class InheritedChild extends StatelessWidget {
30   @override
31   Widget build(BuildContext context) {
32     // Jedes Child unter dem InheritedWidget kann
33     // auf den State zugreifen
34     final importantInfo = ImportantInfo.of(context);
35     return Text(importantInfo.info);
36   }
37 }
```

Listing 6.3: Verwendung eines InheritedWidgets

Wie im Listing 6.3 ersichtlich, benötigt ein *InheritedWidget* die Methode *updateShould-Notify*, welche signalisiert, ob bei Änderungen die abhängigen Children neu gezeich-

net werden sollen. Da diese Methode nur den alten Zustand zum Vergleich als Input bekommt und einen Boolean als Rückgabewert erlaubt, werden bei jeder Änderung des States alle Children neu gezeichnet, unabhängig davon, ob sie die jeweilige Eigenschaft benötigen oder nicht. Da dies ein potenzielles Performanceproblem darstellt (bei komplexen States und Widgets), kann stattdessen auf ein *InheritedModel* zurückgegriffen werden. Dieses baut auf das *InheritedWidget* auf und ermöglicht es zusätzlich sogenannte Aspekte für das Model zu definieren. Diese Aspekte erlauben es, dass Children nur bei relevanten Änderungen des Models sich neu zeichnen. Möglich wird es durch die neue Methode *updateShouldNotifyDependent* und einem zusätzlichen Parameter in der of-Methode. Die neue Methode ermöglicht Notifizierungen anhand des übergebenen Aspekts zu steuern, wodurch nur jene Children die Änderungen bekommen, welche sie benötigen. Der neue Parameter in der of-Methode definiert dabei den benötigten Aspekt im Widget. Das nachfolgende Beispiel 6.4 zeigt die notwendigen Anpassungen um ein *InheritedModel* zu nutzen (Cheng, 2019).

```
1  // Die of-Methode im InheritedModel beinhaltet zusätzlich
      einen Aspekt
2  static ImportantInfoContent of(BuildContext context, String
      aspect) {
3    ImportantInfo infoModel = InheritedModel.inheritFrom(
        context, aspect: aspect);
4    return infoModel?.infoContent ?? ImportantInfoContent.
        fallback();
5  }
6
7  // Die neue Methode regelt, wann eine Notifizierung geschickt
      wird
8  @override
9  bool updateShouldNotifyDependent(ImportantInfo oldWidget, Set
       dependencies) {
10   return (infoContent.info != oldWidget.infoContent.info &&
11               dependencies.contains('info')) ||
12           (infoContent.version != oldWidget.infoContent.
              version &&
13               dependencies.contains('version'));
14 }
15
16 // Wenn man in einem Widget auf das InheritedModel
17 // zugreift, kann man durch den Aspekt steuern welche
18 // Notifizierungen man bekommen möchte
19 final importantInfo = ImportantInfo.of(context, 'info');
20 return Text(importantInfo.info);
```

Listing 6.4: Verwendung eines InheritedModels

6.2.3 Provider

Die Widgets aus Kapitel 6.2.2 sind low-level APIs aus dem Flutter Framework. Um diese vereinfacht und effizienter nutzen zu können, gibt es einen Wrapper in Form eines Packages - dem Provider Package. Es bietet einige Möglichkeiten die Konzepte von *InheritedWidgets* vereinfacht und wiederverwendbar zu nutzen und baut dabei auf die offizielle InheritedWidgets-API auf. Die offizielle Flutter Website empfiehlt und nutzt dieses Package als Beispiel für die Umsetzung von State-Management anhand ihrer Beispielapplikation (siehe (flutter.dev, 2020e)). Laut offizieller Dokumentation bietet Provider dabei folgende Mehrwerte gegenüber der Verwendung von *InheritedWidgets* (dash-overflow.net, 2020; flutter.dev, 2020e):

- Vereinfachte Erstellung und Entsorgung von Ressourcen
- Unterstützt Lazy-Loading von Ressourcen
- Reduziert die Anzahl an zu schreibenden Codezeilen
- Unterstützt die Fehlerfindung mittels DevTools
- Vereinfachter Zugriff mittels Provider.of/Consumer/Selector

Dabei bietet das Package die Möglichkeit, einzelne Eigenschaften bis hin zu komplexen Objekten mit sich veränderten Zuständen, im Widget-Tree seinen Children zur Verfügung zu stellen. Eine gängige Möglichkeit ist dabei die Verwendung bzw. Kombination von *ChangeNotifier*, *ChangeNotifierProvider* und *Consumer*. Beim *ChangeNotifier* handelt es sich um eine simple Klasse aus dem Flutter-SDK, welche es ermöglicht, Benachrichtigungen an jene Objekte zu senden, welche darauf hören (vergleichbar mit einem Observable). Sehr einfache Applikationen können dabei nur einen *ChangeNotifier* besitzen, bei komplexeren gibt es verschiedene Models und damit mehrerer *ChangeNotifier*. Die Besonderheit an dieser Klasse ist die *notifyListeners* Methode, mittels derer Benachrichtigungen ausgesendet werden können (siehe Beispiel 6.5) (flutter.dev, 2020e).

```
1 class ItemModel extends ChangeNotifier {
2   final List<String> items = [];
3
4   void add(String item) {
5     items.add(item);
6     // Hier wird die Benachrichtigung ausgesendet
7     notifyListeners();
8   }
9 }
```

Listing 6.5: Erstellung eines ChangenNotifiers

Mittels einem *ChangeNotifierProvider* kann der zuvor erstellte *ChangeNotifier* im Widget-Tree seinen Children zur Verfügung gestellt werden. Hier gilt es, den Provider nicht höher im Tree anzusetzen als notwendig, sondern nur über jenen Widgets, welche auf ihn zugreifen müssen. Der Provider kümmert sich dabei sowohl um die Erstellung des Models (falls notwendig), als auch um das Disposen, sollte die Ressource nicht mehr benötigt werden. Sollten mehrere Provider benötigt werden, kann auf einen *MultiProvider* zurückgegriffen werden. Eine beispielhafte Verwendung ist in Listing 6.6 ersichtlich (flutter.dev, 2020e).

```
// Das Model wird im Widget Tree zur Verfügung gestellt
ChangeNotifierProvider(
  create: (_) => ItemModel(),
  child: MyApp(),
),

// Es können auch mehrere Provider zugleich verwendet werden
MultiProvider(
  providers: [
    ChangeNotifierProvider(create: (_) => ItemModel()),
    Provider(create: (_) => SomeOtherClass()),
  ],
  child: MyApp(),
),
```

Listing 6.6: Verwendung eines ChangeNotifierProviders bzw. MultiProviders

Auf die mittels Provider bereitgestellten Werte kann nun mit einem Consumer-Widget zugegriffen werden, vorausgesetzt man ist ein Child des Providers. Dabei definiert man den genauen Typ des Models, welches man verwenden möchte (Provider benötigt den Typ um das korrekte Model im Widget-Tree zu finden), sowie eine builder-Methode, welche bei Änderungen des Models aufgerufen wird. In dieser builder-Methode kann auf die Werte zugegriffen und die dazugehörigen Widgets gebaut werden. Auch hier gilt, dass das Consumer-Widget so tief wie möglich im Widget-Tree eingesetzt werden soll um die Anzahl der Widgets, welche bei Änderungen neu gezeichnet werden, gering zu halten. Eine beispielhafte Verwendung ist in Listing 6.7 ersichtlich (flutter.dev, 2020e).

```
// Hier wird der benötigte Typ angegeben
Consumer<ItemModel>(
      builder: (context, itemModel, child) {
        // Im builder kann auf das Model zugegriffen werden
        return Text(itemModel.items.join(','));
      })
```

Listing 6.7: Zugriff auf Models mittels Consumer-Widget

Es gibt Situationen, da benötigt man keine Updates oder muss nur auf Methoden des Models zugreifen. Man könnte ein Consumer-Widget dafür verwenden, jedoch würden dadurch Teile der Benutzeroberfläche neu gezeichnet werden ohne dass sie es benötigen. Aus diesem Grund stellt das Provider-Package Methoden zur Verfügung, mittels derer sowohl über Erweiterungen des *BuildContext* als auch über die statische Methode *Provider.of<T>(context)* auf die Models zugegriffen werden kann. Hierbei ist anzumerken, dass der Aufwand ein Model zu finden bei O(1) liegt, da in der Umsetzung nicht Schritt für Schritt im Widget-Tree nach oben gesucht wird. Folgende Methoden stehen dabei zur Verfügung (flutter.dev, 2020e; dash-overflow.net, 2020):

- Alternative Möglichkeit auf das Model zuzugreifen und Updates zu bekommen: *context.watch<T>()* oder *Provider.of<T>(context)*
- Nur auf das Model zugreifen ohne Updates zu bekommen: *context.read<T>()* oder *Provider.of<T>(context,listen: false)*
- Nur für einen Teil des Models Updates bekommen: *context.select<T, R>(R cb(T value))*

6.2.4 Redux

Redux ist ein bekannter Vertreter für State Management aus der Webentwicklung, wobei die Idee unabhängig der zugrundeliegenden Technologie ist und Implementierungen in verschiedensten Sprachen existieren. Die offizielle Kurzbeschreibung dieses Frameworks bezeichnet den Ansatz als vorhersagbaren Zustands-Container für JavaScript-Anwendungen. Dabei werden die Verantwortungen für die Verwaltung des States grob in drei Kategorien unterteilt: dem Store, Actions und Reducers. Der Store beinhaltet dabei den Zustand der gesamten Applikation innerhalb eines Objektes ohne jegliche Logik. Dieser Store kann nur über Actions abgeändert werden, wobei Actions lediglich simple Objekte darstellen, welche die notwendigen Informationen beinhalten. Da eine Action nur beschreibt, dass sich etwas geändert hat und nicht wie der State angepasst werden muss, gibt es zusätzlich Reducers. Diese definieren wie der State anhand der jeweiligen Action angepasst werden muss (dabei wird der aktuelle State nie verändert, sondern immer eine neue Kopie mit den Änderungen erstellt). Nachdem der neue State übernommen wurde, werden alle Komponenten benachrichtigt, welche die Änderungen benötigen, um die aktualisierten Daten anzeigen zu können (siehe Ablaufbeschreibung in Abbildung 6.2). Zusätzlich gibt es die Möglichkeit, eine Middleware zu verwenden, welche zwischen der Action und dem Reducer agiert. Die Middleware funktioniert dabei wie ein Interceptor, der die Actions abändern, neue Actions erstellen oder auch vorhandene verwerfen kann (Ein Beispiel wäre eine Middleware für die Protokollierung, welche alle Actions aufzeichnet und in einem Log festhält) (Garreau & Faurot, 2018; Dinkevich & Gelman, 2017).

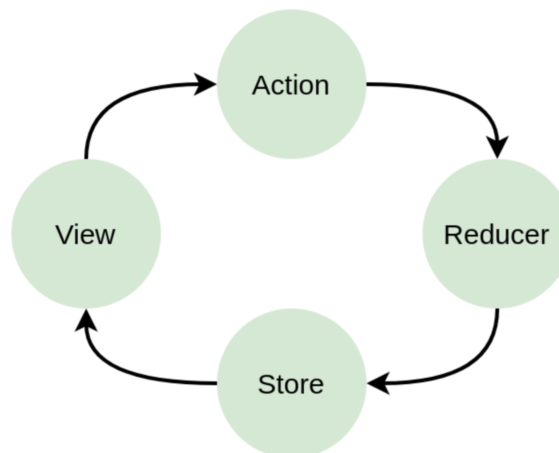

Abbildung 6.2: Ablaufbeschreibung von Redux (Quelle: vgl. Garreau, 2018)

Auf die Frage, warum man Redux als State-Management Ansatz verwenden soll, wird oft auf die Grundprinzipien des Frameworks verwiesen, welche mit der Implementierung einhergehen (Garreau & Faurot, 2018; Dinkevich & Gelman, 2017):

- **Single source of truth** - Der gesamte Zustand der Applikation ist in einem Objekt zusammengefasst, dem Store. Diese Eigenschaft soll dem Entwickler dabei unterstützen, den aktuellen Zustand der Applikation besser zu verstehen sowie das Ergebnis einer Aktion besser abschätzen zu können. Der zentrale Zustand ermöglicht es zudem Aktionen rückgängig zu machen bzw. sie wiederherzustellen zu können. Für die Fehlerfindung kann z.B. auch ein gewisser Zustand während des Auftretens des Fehlers hilfreiche Informationen liefern.

- **Zustand ist read-only** - Die einzige Möglichkeit den Zustand zu verändern ist über Actions. Dadurch gibt es eine zentrale Stelle, welche Änderungen am Zustand durchführt. Da die Actions reine Datenobjekte sind, können diese protokolliert, serialisiert, gespeichert und mehrfach (z.B. für Testzwecke) ausgeführt werden. Dabei wird nie ein Zustands-Objekt abgeändert, sondern immer eine Kopie mit den Änderungen erstellt und zurückgeliefert.

- **Verwendung von Pure Functions** - Pure Functions sind jene Funktionen, die bei gleichem Input immer den gleichen Output liefern. Sie dürfen dadurch nicht von externen Faktoren abhängig sein bzw. diese verändern. Für Änderungen am Zustand werden in Redux ausschließlich Reducer verwendet und diese sind als Pure Functions zu implementieren. Dabei bekommt ein Reducer das aktuelle Zustands-Objekt und die Änderung in Form einer Action als Input und liefert ein neues Zustands-Objekt mit den aktualisierten Eigenschaften zurück. Dieser Ansatz ist notwendig, damit es ein vorhersehbares Transaktionsprotokoll gibt und z.B. die Rückgängig- und Wiederherstellfunktion möglich ist.

6.2.5 BLoC

Bei BLoC handelt es sich mehr um ein Design-Pattern als um eine Library (es gibt Packages, welche die Umsetzung erleichtern, die Grundidee des Patterns ist dabei ohne Abhängigkeiten anwendbar). Von Google entwickelt und 2018 als Business Logic Components (kurz BLoC) vorgestellt, baut der Ansatz stark auf die reaktive Programmierung auf und nutzt dabei die vorhandenen Streams von Dart. Dabei soll das Widget sich nur um die Anzeige kümmern und die Logik in separaten Komponenten gehalten werden. Die Grundprinzipien beruhen dabei auf der Separation von Businesslogik und Benutzeroberfläche, der Verwendung von Streams sowohl für Input als auch Output eines BLoCs und der Plattformunabhängigkeit der Komponenten. Die Grundidee entstand aus der Notwendigkeit den gleichen Code für Web-, Mobile- und Backendapplikationen nutzen zu können. Vereinfacht ausgedrückt nimmt ein BLoC Events über Streams entgegen, wendet anhand des Events die notwendigen Business-Logiken an und stellt die Ergebnisse über ein oder mehrere Streams zur Verfügung (siehe Abbildung 6.3). Für die Verwendung des Patterns gibt es dabei sowohl für das Applikationsdesign als auch für die Benutzeroberfläche einige Regeln, die zu beachten sind und als nicht verhandelbar gelten. Für das Applikationsdesign gilt (Alessandria, 2020; Windmill, 2019):

- Inputs und Outputs bestehen nur aus Sinks und Streams (siehe Kapitel 5.2.1.3 für Details). Es dürfen keine Funktionen, Konstanten oder Variablen verwendet werden.
- Abhängigkeiten müssen injizierbar sein. Wenn Flutter-spezifische Bibliotheken im BLoC benötigt werden, handelt es sich um Aufgaben der Benutzeroberfläche und nicht um Business Logik. Dadurch sollten die Aufgaben in das Widget verschoben werden.
- Plattformspezifischer Code ist nicht erlaubt. Damit ist eine Unterteilung z.B. mittels *if(device == Android){...}* untersagt.
- Alles andere ist erlaubt, solange die obigen drei Regeln befolgt werden.

Für die Erstellung der Benutzeroberfläche in Kombination mit BLoCs gelten folgende Regeln (Windmill, 2019):

- Jede Komponente mit einer komplexen Logik besitzt einen eigenen BLoC (der Grad, ab wann eine Komponente als komplex gilt, kann dabei variieren).
- Komponenten sollen den Input so an den BLoC senden wie er entsteht. Es soll keine Logik im Widget geben, welcher Daten zuvor transformiert, da dies die Aufgabe des BLoCs ist.

- Die Werte, die ein BLoC an ein Widget liefert, sollen möglichst ohne Transformation übernommen werden können. Benötigt man z.B. von einer Zahl einen formatierten String, so wird dieser innerhalb des BLoCs aufbereitet.

- Jegliche Unterscheidungen sollen anhand einfacher Boolean-Logik erfolgen. So ist es erlaubt z.B. die Farbe anhand eines Wertes zu setzten: *color: bloc.isDestructive ? Colors.red : Colors.blue*. Komplexere Logik wie z.B. *if (bloc.buttonIsDestructive && bloc.buttonIsEnabled && bloc.userIsAdmin) {...}* ist nicht zulässig. Eine solche Abfrage sollte aggregiert und in den BLoC verschoben werden.

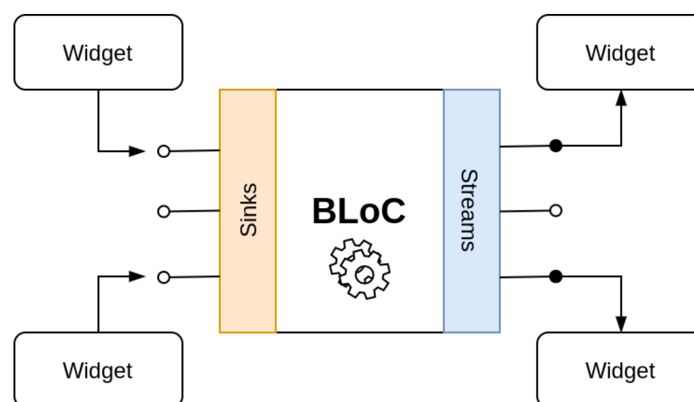

Abbildung 6.3: Datenfluss in einem BLoC (Quelle: vgl. Boelens, 2018)

Das BLoC Pattern definiert somit wie die Daten gehalten werden sollen und der Datenfluss zu gestalten ist. Die Separation von Benutzeroberfläche und Businesslogik und die damit einhergehende Trennung der Verantwortlichkeiten ermöglicht es, einzelne Teile der Businesslogik gezielter testen zu können. Es kann somit auch das Layout unabhängiger zur Businesslogik erstellt und angepasst werden, da potentiell von jeder Stelle auf die Streams zugegriffen werden kann. Aus Performance Sicht bringt es zusätzliche Verbesserungen, da mittels Streams und den dazugehörigen *StreamBuildern* gezielt einzelne Teile des Widget-Trees neu gezeichnet werden können, anstelle das komplette Widget neu zu zeichnen. Ein essentieller Punkt bei der Verwendung des Patterns wird jedoch dem Entwickler überlassen, und zwar wie die einzelnen BLoCs im Widget Tree zur Verfügung gestellt werden. Die Möglichkeiten reichen dabei von globalen Singeltons über lokale Instanzen oder Parent-Widgets bis hin zu Packages von Drittanbietern. Eine Möglichkeit, welche auch von einem bekannten Package für die Umsetzung des BLoC Patterns verwendet wird (*flutter_bloc*), ist der Einsatz des Provider Packages (siehe Kapitel 6.2.3). Damit kombiniert man die Möglichkeit mittels Provider Klassen im Widget-Tree zur Verfügung zu stellen mit den klaren Richtlinien und Schnittstellen von BLoC und dessen effizienter Nutzung von Streams (Boelens, 2018).

6.2.6 MobX

Bei MobX handelt es sich um einen weiteren Vertreter einer reaktiven Library für die Verwaltung von State. Dabei baut der Ansatz auf drei Konzepten auf: Observables, Actions und Reaction. Bei den Observables handelt es sich um den reaktiven Zustand der Applikation oder einzelnen Komponenten. Reaktiv bedeutet in diesem Kontext, dass bei Änderungen des Zustands all jene Komponenten informiert werden, welche sich auf die Änderungen subscribed haben (Anwendung des klassischen Observer-Patterns). Die Observables unterteilen sich zudem in den Core- und Derived-State. Der Core-State spiegelt dabei genau den Domänen-Zustand ab, in welchem man sich befindet (z.B. existiert ein Kontakt-Eintrag, wobei die Variablen *firstName* und *lastName* den Core-State des Kontaktes abbilden). Die Variable *fullName* kann dabei durch die Kombination der Variablen des Core-States abgeleitet werden, wodurch sie als Derived-State bezeichnet wird. Solch ein Derived-State kann vom Core-State oder weiteren Derived-States abhängen und wird als Computed Observable bezeichnet. Der Wert der Variable wird automatisch aktualisiert, sollten sich die zugrundeliegenden Observables ändern. Da bei der konkreten Implementierung mit Observables eine Menge an Code geschrieben werden muss, gibt es von der Library zusätzlich Annotations, welche die Logik hinter den Observables automatisch generiert und die Verwendung für den Entwickler vereinfacht (Beispiel in Listing 6.8) (Podila & Weststrate, 2018; Wieruch, 2017).

```
@observable
String firstName;

@observable
String lastName;

@computed
String get fullName => firstName + ',' + lastName;
```

Listing 6.8: Verwendung der MobX Annotations

Grundsätzlich erlaubt der Aufbau von MobX, dass der Inhalt von Observables direkt geändert werden kann. Die Library sieht jedoch sogenannte Actions für die Abänderung des States vor, welche durch die *@action* Annotation an einer Methode leicht erstellt werden können. Eine Action definiert dabei explizit, wie ein State abgeändert wird (z.B. anstelle nur *value++* zu verwenden gibt die Methode *increment()* mehr Aufschluss über den Nutzen). Außerdem gruppiert die Library alle Änderungen innerhalb einer Action, wodurch Benachrichtigungen von Observables nur am Ende der Action an alle ausgesendet werden, anstelle bei jeder Änderung innerhalb der Methode. Das letzte Konzept von MobX beschreibt die Reactions, welche auf Änderungen der Observables reagieren können. Ein großer Vorteil ist dabei, dass sie ohne Setup oder manuelles Registrieren automatisch auf die Änderungen des Observables reagieren. Es genügt dabei, wenn innerhalb der Reaction auf das Observable

zugegriffen wird. Es werden dabei mehrere Varianten von Reactions zur Verfügung gestellt, wobei jede dieser einen *ReactionDisposer* zurückliefert, mittels dem man die Reaction jederzeit verwerfen kann. Die Library stellt folgende Reactions zur Verfügung (mobx.netlify.app, 2020):

- **autorun(Function(Reaction) fn)** - führt die Funktion initial und bei jeder Änderung eines Observables, welches innerhalb der Funktion verwendet wird, auf.

- **reaction<T>(T Function(Reaction) fn, void Function(T) effect)** - bei jeder Änderung der Observables, welche in der fn-Funktion verwendet werden, wird die effect-Funktion aufgerufen.

- **when(bool Function(Reaction) predicate, void Function() effect)** - bei jeder Änderung der Observables wird die predicate-Funktion aufgerufen, welche entweder *true* oder *false* zurückliefert. Sollte sie *true* zurückliefern, wird die effect-Funktion aufgerufen. Im Gegensatz zu den anderen Methoden wird diese Methode automatisch verworfen, nachdem einmal die effekt-Methode aufgerufen wurde.

- **asyncWhen(bool Function(Reaction) predicate)** - Ähnlich zur when-Methode, jedoch wird eine Future zurückgeliefert sobald die predicate-Funktion *true* liefert.

Da der State über die Benutzeroberfläche dem Benutzer zur Verfügung gestellt werden muss, ist ein essentielles Thema die Darstellung und Aktualisierung der Daten in der App. Aus diesem Grund bietet die Library über das Zusatzpackage *flutter_mobx* eine einfache Möglichkeit, den State und die UI synchron zu halten. Mittels dem Observer-Widget können Observables innerhalb des Widget-Trees verwendet werden, wobei das Widget dem StreamBuilder (siehe Kapitel 5.2.2) ähnelt. Jedes Observable, welches in der builder-Funktion des Observer-Widgets verwendet wird, führt dazu, dass sich das Widget bei Änderungen neu zeichnet (mobx.netlify.app, 2020).

7 Software Qualität und Code-Metriken

Als Entwickler ist man daran interessiert, qualitativ hochwertigen Code zu schreiben. Qualität bedeutet dabei nicht nur, dass die Applikation fehlerfrei läuft, sondern dass der Code auch verständlich, wartbar, erweiterbar, nicht unnötig komplex und lose gekoppelt ist. Um die Qualität von Software bewerten und über einen längeren Zeitraum verfolgen zu können, gibt es Software Metriken, mittels denen einzelne Qualitätsaspekte als Kennzahlen abgebildet werden können. Mit vorhandenen Kennzahlen als Grundlage können wichtige Entscheidungen im Entwicklungsprozess anhand erhobener Daten getroffen werden. Um qualitative Kennzahlen und daraus resultierenden Aussagen treffen zu können, ist es essentiell, die richtigen Daten zu erheben (Bird, Menzies & Zimmermann, 2015).

7.1 Einordnung und Abgrenzung

Da die Aufzeichnung, Aufbereitung und Auswertung von Metriken zusätzliche Aufwände zum Entwickeln darstellen, gilt es zu entscheiden ob und welche Metriken relevant sind. Sie sollen dabei konkret unterstützen ein Ziel zu erreichen bzw. eine Entscheidung mit vorliegenden Daten treffen zu können. Essentiell hierbei ist, dass nur Metriken betrachtet werden, welche die Ziele bzw. die Entscheidungen beeinflussen können und dass diese dabei immer über einen Zeitraum betrachtet werden. Eine Metrik zu einem Stichtag liefert oft keine Aussagekraft, vielmehr sollte man über einen Zeitraum die Veränderungen beobachten und interpretieren. Bei Software Metriken unterscheidet man zwischen verschiedenen Typen, welche sowohl technische als auch organisatorische Aspekte abdecken. Bei den organisatorischen Kennzahlen handelt es sich z.B. um den Aufwand für den Betrieb der Software, die durchschnittliche Zeit für eine Fehlerbehebung, die Leistung eines Entwicklerteams über einen gewissen Zeitraum oder auch der Grad der Kundenzufriedenheit. Da diese Arbeit sich aber auf die technischen Aspekte bezieht, wird in den nachfolgenden Kapiteln nur auf technische Metriken eingegangen. File-Metriken können dabei grob in fünf Bereiche eingeteilt werden (Bird et al., 2015; Mistrik, Bahsoon, Kazman & Zhang, 2014):

- **Größe** - einer der naheliegendsten Messgrößen ist die Anzahl an Codezeilen. Sie ist einfach zu erheben, jedoch hat sie Schwächen. Beispielsweise ist es mög-

lich, dieselbe Funktionalität mit weniger oder mehr Codezeilen zu schreiben, während eine ähnliche Komplexität beibehalten wird.

- **Komplexität** - man geht davon aus, dass eine geringere Komplexität innerhalb eines Files die Wartbarkeit und Erweiterbarkeit unterstützt. Zur Bestimmung der Komplexität kann z.B. die McCabe-Metrik (auch zyklomatische Komplexität) verwendet werden, welche die Anzahl an linear unabhängigen Pfaden bestimmt.

- **Kopplung** - neuere Metriken legen den Fokus auf die Koppelung bzw. Abhängigkeit von Dateien und Klassen zu anderen Dateien und Klassen. Eine geringere Koppelung (oft auch lose Koppelung genannt) ist dabei ein Indikator für eine gute Modularität und Wiederverwendbarkeit. Für die Bestimmung der Koppelung kann dabei auf eine Design Structure Matrix zurückgegriffen werden.

- **Zusammenhang** - bei dieser Kennzahl wird der Zusammenhang der Verantwortlichkeiten innerhalb einer Datei oder Klasse bestimmt. Die zugrundeliegende Idee hierbei ist, dass Dateien oder Klassen nur eine Verantwortung bzw. einen Bereich abdecken sollten. Die daraus resultierende Trennung von Zuständigkeiten soll die Wartbarkeit unterstützen.

- **Vererbung** - diese Metrik misst die Anzahl der Hierarchien von Vererbungen und ist deshalb nur für objektorientierte Programmiersprachen relevant. Flachere Hierarchien gelten als simpler und sind dadurch leichter zu verstehen und zu warten.

Ein weiterer wichtiger Aspekt ist Code Qualität, da sie direkte Auswirkungen auf die Qualität der Software bzw. des Produkts hat. Im gesamten Entwicklungsprozess sollte hohe Qualität ein klares Ziel sein und dabei von jedem Beteiligten berücksichtigt werden (vom Entwickler bis hin zum Geschäftsführer). Es gibt dabei fünf Hauptkategorien, auf die es zu achten gilt bzw. welche eine Kennzahl für Code Qualität darstellen können: Reliability, Maintainability, Testability, Portability und Reusability. Bei der Reliability geht es darum die Wahrscheinlichkeit zu messen, mit welcher ein System über einen bestimmten Zeitraum fehlerfrei läuft. Die Metriken beziehen sich dabei oft auf die Anzahl der Mängel und die Verfügbarkeit der Software. Die Maintainability hingegen bewertet die Eigenschaft, wie einfach die Software zu warten ist. Dabei kann das oft nicht durch eine einzelne Kennzahl festgestellt werden, sondern ergibt sich aus der Kombination vieler einzelner Bereiche, welche in diesem Kapitel bereits angesprochen wurden bzw. angesprochen werden (File-Metriken, Testablitiy, etc.). Die Testablity soll Aufschluss darüber geben, wie gut einzelne Teile oder gesamte Abläufe einer Software testbar sind. Wichtig dabei ist, feststellen zu können wie gut man Tests steuern, einsehen, isolieren und automatisieren kann. Portability und Reusability beschäftigen sich beide mit der Wiederverwendbarkeit von Code. Reusability bewertet dabei die grundsätzliche Möglichkeit Code wiederzuverwenden, was

z.B. durch Modularität oder einer losen Koppelungen zwischen Komponenten ermöglicht werden kann. Portability hingegen beschreibt die Eigenschaft, den identen Code in verschiedenen Bereichen der Software oder auf verschiedenen Plattformen einsetzen zu können (Richard Bellairs, 2019).

Durch das Messen der Qualität kann man seinen aktuellen Standpunkt bestimmen, jedoch ist es essentiell, Maßnahmen zu definieren um diese Qualität beizubehalten bzw. zu verbessern. Eine der besten Möglichkeiten ist dabei die Definition und Einhaltung eines Coding Standards. Dieser Standard soll dabei einen einheitlichen Stil vorgeben und dabei die Konsistenz und Lesbarkeit der Codebasis sicherstellen. Um während der Entwicklung direkt Feedback zu bekommen und die Einhaltung eines Coding Standards zu erleichtern, bietet sich der Einsatz von statischen Code-Analysen an. Diese können den Code sowohl während der Entwicklung in Echtzeit analysieren und dem Entwickler ein sofortiges Feedback liefern als auch im Zuge von Continuous Integration im Detail durchgeführt werden. (Richard Bellairs, 2019).

7.2 Werkzeuge

Verschiedene Programmiersprachen bieten unterschiedliche Werkzeuge für die Analyse sowie Hilfsmittel für die Erstellung ihres Codes an. Hinsichtlich Konsistenz und Lesbarkeit des Codes stellt Flutter bzw. Dart mit dessen *dartfmt* Tool eine Möglichkeit zur Verfügung, vorhandenen Code anhand einheitlicher Richtlinien zu formatieren. Zusätzlich kann über das inkludierte *analyzer* Package eine statische Codeanalyse durchgeführt werden, welche dem Entwickler bereits während des Schreibens oder durch explizite Ausführung auf etwaige Fehler aufmerksam macht. Die Funktionsweise kann dabei nach eigenen Coding Standards und Vorgaben definiert werden, indem die anzuwendenden Regeln und deren Ausmaß (soll es nur als Warnung oder als Fehler aufscheinen) konfigurierbar sind. Dieses Package soll dabei unterstützen den Code-Stil zu verbessern, möchte man stattdessen konkrete Kennzahlen erfassen oder zusätzliche Funktionen nutzen, so kann man auf Packages von Drittanbietern zurückgreifen. Zusätzlich zu statischen Metriken vom Code können in Dart auch individuelle Laufzeitmetriken aufgezeichnet und analysiert werden. Über das Standardpackage *dart:developer* kann mittels der *Metrics* Klasse eine zu protokollierende Eigenschaft erstellt und aufgezeichnet werden. Unabhängig der Tools und Werkzeuge, die durch die Sprachen mitgeliefert werden, kann auch auf externe Tools wie z.B. SonarQube zurückgegriffen werden. Da die Auswahl und Verwendung solcher Tools über den Umfang dieser Arbeit hinausgehen, wird in den nachfolgenden Kapiteln nicht genauer darauf eingegangen (dart.dev, 2020a; Mainkar & Giordano, 2019).

7.3 Relevanz und Auswahl

Ziel dieser Arbeit ist die Analyse verschiedener State-Management Ansätze, wodurch der Fokus auf technischen Kennzahlen liegt. Für die Berechnung und Auswertung dieser Kennzahlen wird das Dart-Package *dart_code_metrics* in der Version 2.0.0 verwendet. Damit können die erstellten Sourcecode Files anhand folgender Eigenschaften analysiert und verglichen werden (pub.dev, 2020; *Maintainability Index Range and Meaning*, 2007):

- **Zyklomatische Komplexität** - (wie in Kapitel 7.1 beschrieben)

- **Anzahl ausführbarer Codezeilen** - die Anzahl der ausführbaren Codezeilen oder Operationen in einer Funktion oder Methode. Leer- oder Kommentarzeilen werden nicht gezählt. Funktionen mit hohen Werten dieser Metrik sind oft komplex und schwer zu warten.

- **Anzahl der Methoden** - die Gesamtzahl der Methoden (oder Funktionen) in einer Klasse. Wenn eine Klasse über zu viele Methoden verfügt, weist dies oft auf eine hohe Komplexität der Klasse hin.

- **Maximale Anzahl an Argumenten** - Maximale Anzahl der in einer Methode (oder Funktion) verwendeten Argumente. Wenn eine Methode zu viele Argumente enthält, ist es schwierig sie aufzurufen und zu ändern, wenn sie an vielen verschiedenen Orten verwendet wird.

- **Wartbarkeitsindex** - dieser Index ergibt sich aus der Kombination der Halstead-Metrik, der Zyklomatische Komplexität und der Anzahl der Codezeilen. Der Index hat dabei einen Wertebereich von 0-100 (ein höherer Wert bedeutet eine bessere Wartbarkeit) und berechnet sich wie folgt:

$$\begin{aligned} MaintainabilityIndex = MAX(0, (171 - 5.2 * ln(HalsteadVolume) \\ - 0.23 * (CyclomaticComplexity) \\ - 16.2 * ln(LinesofCode)) * 100/171) \end{aligned}$$

8 Methodik der empirischen Untersuchung

Um die Forschungsfrage hinsichtlich der wirtschaftlichen und technologischen Entscheidungsgründe für den Einsatz von State-Management Varianten beantworten zu können, werden im Zuge dieser Arbeit die zuvor beschriebenen Ansätze (Kapitel 6.2) anhand einer Beispielapplikation umgesetzt und bewertet. Um die Ergebnisse der einzelnen Ansätze vergleichen zu können, werden in den nachfolgenden Kapiteln der Funktionsumfang der zu erstellenden Beispielapplikation sowie die zu erhebenden Kennzahlen definiert.

8.1 Funktionsumfang der Beispielapplikation

Um aussagekräftige Vergleiche zwischen den State-Management Varianten erstellen zu können, ist es notwendig, in jeder Variante die identen Funktionen abzubilden. Die gewählten Funktionen sollen dabei gängige Anforderungen an eine State-Management Lösung repräsentieren. Für diese Arbeit wurde eine Todo-App als Beispielapplikation gewählt. Die Anforderungen ergeben sich dabei wie folgt:

Anforderung an State	**Umsetzung in der Beispielapplikation**
Lokalen State bereitstellen	Liste an Todos wird in der App bereitgestellt.
Hinzufügen, Löschen und Bearbeiten des States	Todo-Einträge können in der App hinzugefügt, gelöscht und als erledigt markiert werden.
Abgeleiteter State	Statistik über die Todos wird über den State zur Verfügung gestellt (Anzahl der offenen und erledigten Todos).
Zugriff von mehreren Stellen	Anzahl der offenen Todos wird zusätzlich als Badge angezeigt.

Asynchrone Operation	Die Liste an zuweisbaren Benutzern wird über einen API-Call initial geladen, sollte die Liste lokal noch nicht zur Verfügung stehen. Ansonsten wir die lokale Liste bereitgestellt.
Loading State	Beim Laden der zuweisbaren Benutzer wird dem User über die Benutzeroberfläche signalisiert, dass asynchron Daten geladen werden.
Error State	Ein Benutzer kann nur einem Todo zugewiesen werden (Anforderung kommt von der Notwendigkeit einen Error-Fall in der Applikation herbeiführen zu können). Sollte ein Benutzer dennoch zu einem zweiten Todo zugewiesen werden, wird eine Fehlermeldung angezeigt.

Tabelle 8.1: Funktionsumfang der Beispielapplikation

Für den asynchronen API-Call der zuweisbaren Benutzer wird der Online REST API Service *JSON Placeholder* verwendet (`https://jsonplaceholder.typicode.com/users`).

8.2 Herangehensweise

Um die Analyse und Bewertung der Umsetzungen spezifisch für die Varianten durchführen zu können, wird der Sourcecode zwischen allgemeinen und spezifischen Funktion der State-Management Varianten unterteilt. Der allgemeine Teil wird dabei separat und unabhängig zu den Varianten entwickelt (z.B. der API-Aufruf für die zuweisbaren Benutzer). Für die Bewertung wird nur der spezifische Code der jeweiligen Varianten herangezogen, welcher pro State-Management Ansatz in eigene Unterordner aufgeteilt wird. Die Bewertung erfolgt anhand zwei unterschiedlicher Hauptkategorien: Kennzahlen aus statischer Codeanalyse und die Einstufung bzw. Bewertung der Umsetzung anhand definierter Kategorien. Für die statische Codeanalyse werden die in Kapitel 7.3 beschriebenen Kennzahlen erhoben, wobei diese sowohl ganzheitlich als auch auf File-Ebene (sofern aussagekräftig) betrachtet werden. Für die Einstufung bzw. Bewertung der Umsetzung wurden anhand der Informationen aus Kapitel 7 Kategorien definiert, welche auf einer Skala von 0-5 (wobei 0 den schlechtesten und 5 den besten Wert repräsentiert) bewertet werden. Um die Einstufung nachvollziehbar zu gestalten, gibt es zusätzlich zum numerischen Wert eine Begründung für die Auswahl. Folgende Kategorien werden erhoben:

Kategorie	**Beschreibung**
Testbarkeit	Wie gut bzw. unabhängig kann die Businesslogik getestet werden. Exemplarisch werden hierzu das Hinzufügen, Löschen und Bearbeiten des States getestet.
Integration im Framework	Wie gut passt der State-Management Ansatz zum restlichen Aufbau der Flutter SDK. Unterstützt es den reaktiven Ansatz oder andere Best Practices des Frameworks.
Verbreitung	Ist der Ansatz komplett neu oder bereits von anderen Technologien oder Programmiersprachen bekannt. Bekannte Ansätze können den Einstieg für Entwickler erleichtern bzw. Unternehmen finden leichter qualifizierte Mitarbeiter.
Dokumentation	Existiert eine detaillierte Dokumentation bzw. gibt es Beispielimplementierungen für bekannte Problemstellungen.
Widget Rendering	Wie gut kann gesteuert werden, welche Widgets neu gezeichnet werden sollen. Eine Grundregel für eine gute Performance lautet, nur jene Widgets neu zu zeichnen, welche sich geändert haben.
Debugging	Wie gut unterstützt der Ansatz die Fehlerfindung und Analyse des Codes während der Entwicklung.
Tooling	Sind externe Tools notwendig bzw. unterstützen diese Tools oder erzeugen sie zusätzlichen Aufwand.
Änderbarkeit und Erweiterbarkeit	Wie aufwändig ist die Umsetzung eines Features in der gewählten Variante und wie viele bestehende Klassen oder Dateien müssen angepasst werden. Wie gut können Funktionen auf längere Sicht angepasst und erweitert werden.
Architektur und einheitliche Vorgehensweise	Unterstützt die Variante, eine einheitliche Architektur zu definieren bzw. bei größeren Teams eine einheitliche Vorgehensweise sicherzustellen.

Tabelle 8.2: Subjektive Bewertungskriterien

9 Umsetzung und Bewertung der Varianten

In den nachfolgenden Kapiteln werden zum Teil nur Ausschnitte des Codes gezeigt, der komplette Source Code steht auf der beiliegenden CD zur Verfügung. Bei der Auswertung der Kennzahlen mittels dem *dart_code_metrics* Package wurden die einzelnen Werte händisch aggregiert, da das Tool mit einigen Klassen ohne Logik falsche Werte geliefert hat. Bei einer reinen Datenklasse kann z.B. das Tool keinen Wartbarkeitsindex berechnen und liefert den Wert 0 zurück, was das Gesamtergebnis verfälscht (aus diesem Grund wurden diese Werte bei der Berechnung nicht berücksichtigt). Die Werte für den Wartbarkeitsindex pro File wurden dabei mittels arithmetischem Mittel zu einem Wert zusammengefasst (um Ausreißer zu berücksichtigen). Da die Varianten setState (Kapitel 6.2.1) und InheritedWidget bzw. InheritedModel (Kapitel 6.2.2) laut offizieller Dokumentation nur eine low-level API darstellen, wurden sie in den nachfolgenden Kapiteln nicht umgesetzt. Die restlichen Varianten wurden jeweils mit der Flutter Version 1.20.4 erstellt.

9.1 Variante - Provider

Für die Umsetzung mit dem Provider Package (in der Version 4.3.2+2) wurde auf drei Kernelemente der Library zurückgegriffen: *Provider*, *ChangeNotifierProvider* und *Consumer*-Widget. Ein einfacher Provider wurde für die Verwaltung der zuweisbaren Benutzer eingesetzt. Der erstellte *UserProvider* beinhaltet dabei eine interne Liste an Benutzern und eine Methode, um eine Liste an Benutzern bereit zu stellen. Der Provider hält sich den Zustand der Liste und holt sich die notwendigen Daten von einer externen API nur, wenn lokal noch keine vorhanden sind (siehe Listing 9.1). Der Loading-State kann dabei als einfache Future abgebildet werden, welche mittels *FutureBuilder* im Widget-Tree verwendet wird (solange die Future nicht abgeschlossen wurde, wird ein Loading-Indicator angezeigt).

```
1 class UserProvider {
2   final _userList = List<User>();
3
4   Future<List<User>> getUserList() async {
```

```
    if (_userList.isEmpty) {
      final result = await ApiClient.getUsers();
      _userList.addAll(result.userList);
    }

    return _userList;
  }
}
```

Listing 9.1: Implementierung des UserProvider

Der Provider für die Todos wurde als *ChangeNotifierProvider* umgesetzt, da Änderungen an den Todos an alle propagiert werden sollen, welche darauf hören. Der Aufbau ähnelt einem normalen Provider, jedoch erbt die Klasse von *ChangeNotifier*. Dadurch steht die Methode *notifyListeners()* zur Verfügung, mittels derer die Benachrichtigung bei einer Änderung ausgesendet werden kann. Ein Auszug aus der Implementierung des *TodoProvider* ist in Listing 9.2 ersichtlich:

```
class TodoProvider extends ChangeNotifier {
  final _todos = List<Todo>();

  List<Todo> get todos => List.unmodifiable(_todos);

  int get todoCount => _todos.length;

  void updateTodo({String id, bool isDone}) {
    _todos.firstWhere((todo) => todo.id == id).isDone =
       isDone;
    notifyListeners();
  }
}
```

Listing 9.2: Auszug aus der Implementierung des TodoProvider

Sowohl der *UserProvider* als auch der *TodoProvider* wurden mittels *MultiProvider* verwendet. Dieser ermöglicht, mehrere Provider zugleich im Widget-Tree zur Verfügung zu stellen, was die Tiefe des Trees minimiert. Da die Provider zum Widget-Tree gebunden sind, in jenem sie definiert wurden, sind sie auch nur in diesem verfügbar. Da das Hinzufügen eines Todos auf einer eigenen Seite geschieht (und dadurch in einem separaten Widget-Tree), müssen die Provider in diesem Tree explizit zur Verfügung gestellt werden. Das Provider Package bietet dafür zwei Möglichkeiten an, einen Provider zu verwenden - die Erstellung einer neuen Instanz mittels create-Callback oder die Verwendung einer bestehenden Instanz über den Factory-Konstruktor *.value()* (siehe Listing 9.3).

```
// Eine neue Instanz des Providers erstellen
Provider(create: (_) => UserProvider())
// Eine bestehende Instanz des Providers verwenden
Provider.value(value: widget._userProvider)
```

Listing 9.3: Provider im Widget-Tree zur Verfügung stellen

Auf die Werte und Methoden der Provider wurde sowohl mittels dem *Consumer*-Widget als auch über die Extension-Methode *context.read<T>()* zugegriffen, wie in Kapitel 6.2.3 beschrieben. Das *Consumer*-Widget ermöglicht dabei, dass das Widget sich bei jeder Änderung des Providers, welche durch die *notifyListeners()* Methode signalisiert wird, neu zeichnet (Beispiel anhand der Admin-Seite in Listing 9.4).

```
class TodoAdmin extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer<TodoProvider>(
      builder: (_, todoProvider, __) {
        return Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: [
            Text('Gesamtanzahl der Todos: ${todoProvider.
               todoCount}'),
            Text('Anzahl offener Todos: ${todoProvider.
               openTodoCount}'),
            Text('Anzahl abgeschlossener Todos: ${
               todoProvider.doneTodoCount}'),
            RaisedButton(child: Text('alle l schen'),
               onPressed: () => todoProvider.clear()),
            RaisedButton(child: Text('alle als offen
               markieren'), onPressed: () => todoProvider.
               setAllOpen()),
            RaisedButton(child: Text('alle als erledigt
               markieren'), onPressed: () => todoProvider.
               setAllDone()),
          ],
        );
      },
    );
  }
}
```

Listing 9.4: Verwendung des Consumer Widgets

Da es sich bei Provider um einfache Klassen handelt, kann die darin enthaltene Businesslogik unabhängig vom Flutter Framework getestet werden. Ein solcher Unittest ist dabei durch die fehlenden Abhängigkeiten einfach zu erstellen, ein Beispiel dafür ist in Listing 9.5 ersichtlich:

```
test('Delete Todo', () {
  final todoProvider = _givenTodoProviderWithOpenTodo();
  expect(todoProvider.todoCount, 1);
  todoProvider.deleteTodo(id: todoProvider.todos.first.id);
  expect(todoProvider.todoCount, 0);
});
```

Listing 9.5: Testen eines Providers

Objektive Kennzahlen	
Kennzahl	**Wert**
Zyklomatische Komplexität	36
Anzahl ausführbarer Codezeilen	186
Anzahl der Methoden	25
Maximale Anzahl an Argumenten	2
Wartbarkeitsindex	66

Tabelle 9.1: Objektive Kennzahlen - Provider

Kriterium	Wert	Begründung
Testbarkeit	5	Da es sich bei Providern um einfache Klassen handelt, können diese unabhängig vom Flutter Framework mittels Unit Tests getestet werden. Eine lose Koppelung zwischen den Provider erleichtert dabei das Testen einzelner Funktionen.
Integration im Framework	5	Das Package ist eine Erweiterung des *InheritedWidgets* vom Framework und erleichtert dessen Verwendung. Alle Konzepte bauen dabei auf das Flutter Framework auf (z.B. Integration in den Widget-Tree).

Verbreitung	4	Mit 2601 Likes (Stand 31.10.2020) ist Provider das beliebteste Package auf `pub.dev`. Es zählt zu den offiziellen Flutter Favorites und wird auf der offiziellen Flutter Website als State Management Ansatz empfohlen. Da es sich aber um eine spezielle Lösung für das Flutter-Framework handelt, ist die Verbreitung auf das Framework beschränkt.
Dokumentation	4	Die offizielle Dokumentation beinhaltet genauere Beschreibungen für die gängigsten Anwendungsfälle und FAQs. Durch die starke Verbreitung existieren zudem zusätzliche Dokumentationen und Beispiele von der Community.
Widget Rendering	4	Mittels *Consumer*-Widget kann granularer gesteuert werden, welcher Teil des Widget-Trees neu gezeichnet werden soll. Ist nur ein Teil des Providers relevant, kann mittels *Selector*-Widget definiert werden, welche Eigenschaften verwendet werden und bei Änderungen zu einem Rebuild führen sollen. Nachteil ist, dass bei jeder Verwendung explizit definiert werden muss, welche Teile des Providers verwendet werden sollen.
Debugging	5	Da ein Provider ein Widget ist, kann durch das Flutter DevTool auf dessen Werte im Widget-Tree zugegriffen werden (dafür muss die Applikation nicht im Debug-Modus gestartet werden). Zusätzlich kann das Objekt, welches durch den Provider zur Verfügung gestellt wird, die *ReassembleHandler* Klasse implementieren um Hot-Reload zu unterstützen.
Tooling	5	Für den Einsatz von Provider sind keine zusätzlichen Tools notwendig. Das Debugging setzt dabei auf das DevTool des Frameworks auf.
Änderbarkeit und Erweiterbarkeit	4	Durch die Aufteilung verschiedener Funktionen in eigene Provider kann unabhängig an mehreren Features gearbeitet werden. Durch die Verwendung von *MultiProvider* und *ProxyProvider* können mehrere einzelne Provider zusammengefasst und vereinfacht verwendet werden. Da es aber keine zentrale Stelle gibt, an welcher die Provider definiert werden, kann es bei einer großen Anzahl an Providern unübersichtlich und schwer wartbar werden.

Architektur und einheitliche Vorgehensweise	1	Das Package ermöglicht Klassen im Widget-Tree seinen Children zur Verfügung zu stellen, lässt dabei aber die Aufteilung und Art der Implementierung dem Entwickler offen. Die Verwendung von Provider alleine stellt noch keine Architektur dar und legt dadurch keine einheitliche Vorgehensweise in einem Team fest.

Tabelle 9.2: Bewertungen für Provider

9.2 Variante - Redux

Für die Umsetzung der Variante mittels Redux wurde auf zwei Packages zurückgegriffen, dem Basispackage *redux* in der Version 4.0.0+3 und das darauf aufbauende Package *flutter_redux* in der Version 0.7.0. Da es in Redux klare Verantwortungen und Schnittstellen gibt, wurde die Ordnerstruktur dahingehend erstellt:

- **Models** - Beinhaltet die Datenklassen des States.
- **Actions** - Enthält alle Actions, welche auf den State angewendet werden können.
- **Reducers** - Beinhaltet die Reducer-Funktionen, anhand welcher die Actions angewandt werden.
- **Middleware** - Beinhaltet die benötigen Middlewares
- **Containers** - Beinhaltet alle Schnittstellen zwischen Store und UI Komponenten
- **UI** - Enthält die UI Komponenten

Für die Beispielapplikation wurde ein AppState mit einer Liste an Todos und einem eigenen Sub-State für die Userliste definiert. Da die Userliste von einer API geholt wird, beinhaltet dieser State zusätzlich zur Liste an Usern ein Flag, ob die Liste gerade geladen wird. Anhand der Actions wurde definiert, welche Änderungen am State vorgenommen werden können. Diese können sowohl leere Datenklassen sein als auch Variablen beinhalten (siehe Listing 9.6). Um diese Actions auch auf den State anwenden zu können, wurden die notwendigen Reducer definiert. Für eine bessere

Aufteilung wurden die Reducer für die Todos und User in separaten Dateien bzw. Methoden erstellt, welche zum globalen State zusammengeführt werden. Ein Auszug der einzelnen Schritte ist in Listing 9.6 ersichtlich.

```
// Definition der Actions
class AddTodoAction {
  final Todo todo;

  const AddTodoAction(this.todo);
}

class ClearTodosAction {}

// Globaler AppState Reducer
AppState appReducer(AppState state, action) {
  return AppState(
    todos: todoReducers(state.todos, action),
    users: userReducers(state.users, action),
  );
}

// Reducer für Todos
final todoReducers = combineReducers<List<Todo>>([
  TypedReducer<List<Todo>, AddTodoAction>(_addTodo),
  TypedReducer<List<Todo>, ClearTodosAction>(_clearTodos),
]);

List<Todo> _addTodo(List<Todo> todos, AddTodoAction action) {
  return todos..add(action.todo);
}

List<Todo> _clearTodos(List<Todo> _, ClearTodosAction __) {
  return [];
}
```

Listing 9.6: Einsatz von Actions und Reducern

Da eine Action in Redux ein synchroner Aufruf ist und über die App die Liste an zuweisbaren Benutzern aber asynchron geladen werden muss, wurde zusätzlich eine Middleware erstellt. Diese kann zwischen Action und Reducer eingesetzt werden und kann somit bei der Ausführung einer Action zusätzliche Aktionen einleiten. Im Falle der Beispielapplikation wird bei einer *FetchUsersAction* ein API-Call asynchron abgesetzt und nach dessen Vollendung eine weitere Action zum Store mit dem Ergebnis abgesetzt. Um in der Benutzeroberfläche aber auf den Fortschritt anhand eines Loading-Indicator reagieren zu können, wird anhand der *FetchUsersAction* synchron

im Store das Loading-Flag gesetzt (Implementierung der Middleware ist in Listing 9.7 ersichtlich).

```
final List<Middleware<AppState>> userMiddleware = [
  TypedMiddleware<AppState, FetchUsersAction>(_fetchUsers),
];

void _fetchUsers(
    Store<AppState> store,
    FetchUsersAction action,
    NextDispatcher next) {
  ApiClient.getUsers()
      .then((users) => store.dispatch(
         FetchUsersSucceededAction(users.userList)));
  // Mit next wird die FetchUsersAction synchron an den Store
     weitergegeben
  next(action);
}
```

Listing 9.7: Asynchrone Funktionen durch Middleware

Um den Inhalt des Stores in der App verwenden zu können, gibt es das *StoreConnector* Widget. Mittels der durch das Widget zur Verfügung gestellten *converter*-Funktion kann aus dem gesamten Store das notwendige ViewModel abgeleitet und in der *builder*-Funktion verwendet werden. Um eine klare Trennung der Verantwortlichkeiten sicherzustellen, wird dabei zwischen Container-Widget und dem eigentlichen Widget unterschieden. Das Container-Widget bildet dabei die Schnittstelle zum Redux-Store und bereitet die Daten auf bzw. setzt Action ab. Das eigentliche Widget für die Benutzeroberfläche kann unabhängig von Redux entwickelt und verwendet werden (ein Beispiel für ein Container-Widget ist in Listing 9.8 ersichtlich).

```
class TodoListContainer extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return StoreConnector<AppState, _ViewModel>(
      converter: _ViewModel.fromStore,
      builder: (context, vm) {
        return TodoList(
          todos: vm.todos,
          onChanged: vm.onChanged,
          onDelete: vm.onDelete,
        );
      },
    );
  }
```

```
15 }
16
17 class _ViewModel {
18   final List<Todo> todos;
19   final TodoCheckboxChanged onChanged;
20   final TodoOnDelete onDelete;
21
22   const _ViewModel({this.todos, this.onChanged, this.onDelete
       });
23
24   static _ViewModel fromStore(Store<AppState> store) {
25     return _ViewModel(
26         todos: store.state.todos,
27         onChanged: (id, isDone) => store.dispatch(
             UpdateTodoAction(id: id, isDone: isDone)),
28         onDelete: (id) => store.dispatch(DeleteTodoAction(id:
               id)));
29   }
30 }
```

Listing 9.8: Container Widget in Redux

Da der gesamte Applikationszustand in einem Objekt zusammengefasst wird, können mit Unit Tests einzelne Funktionen bzw. Actions in verschiedensten Konstellationen getestet werden. Für einen Test benötigt man dabei die Instanz eines Stores, mit beliebigen Eingangsbedingungen wie z.B. mit einem offenen Todo, und die zu testende Action (ein Beispiel ist in Listing 9.9 ersichtlich).

```
1 test('Delete Todo', () {
2   final store = _givenStoreWithOneOpenTodo();
3   expect(store.state.todos.length, 1);
4
5   store.dispatch(DeleteTodoAction(id: store.state.todos.first
       .id));
6   expect(store.state.todos.length, 0);
7 });
```

Listing 9.9: Testen von Redux

Objektive Kennzahlen	
Kennzahl	**Wert**
Zyklomatische Komplexität	50
Anzahl ausführbarer Codezeilen	224
Anzahl der Methoden	26
Maximale Anzahl an Argumenten	3
Wartbarkeitsindex	78

Tabelle 9.3: Objektive Kennzahlen - Redux

Kriterium	Wert	Begründung
Testbarkeit	4	Der gesamte Zustand der Applikation ist in einem Objekt zusammengefasst und über Unit Tests können Actions und die darunter liegenden Reducer getestet werden. Der zentrale State erleichtert es dabei, verschiedene Testkonstellationen zu erstellen. Asynchrone Aufrufe über eine Middleware lassen sich durch die Abstraktion der Actions nicht einfach Ende zu Ende testen (Middleware wird oft getrennt getestet).
Integration im Framework	2	Durch das *flutter_redux* Package stehen Widgets zur Verfügung einen Redux Store im Widget Tree zu verwenden, jedoch schränkt der Ansatz in einigen Gebieten ein. Da z.B. ein Store lediglich Daten beinhaltet, ist der Einsatz von Streams nicht ohne Weiteres möglich.
Verbreitung	4	Wenn man aus der Webentwicklung kommt, ist die Wahrscheinlichkeit hoch, dass man bereits mit Redux gearbeitet oder davon gehört hat. In der Flutter-Community ist die Library jedoch noch nicht stark vertreten (Likes auf `pub.dev` mit Stand 01.11.2020: *redux* - 134 und *flutter_redux* - 169).
Dokumentation	5	Es existiert sowohl eine allgemeine Dokumentation zu Redux als auch eine spezifische für den Einsatz in Flutter. Beispielapplikationen mit Best Practices unterstützen dabei das Verständnis.

Widget Rendering	2	Grundsätzlich wird der gesamte Widget-Tree eines *StoreConnectors* jedes Mal neu gezeichnet, sollte sich etwas am globalen State ändern. Um nur bei relevanten Änderungen den Tree neu zu zeichnen, gibt es die Möglichkeit bei der Erstellung des *StoreConnectors* das Flag *distinct* auf *true* zu setzen. Damit würde der Tree nur neu gezeichnet werden, sollte sich der Inhalt des ViewModels ändern (die Entscheidung kann nur auf Ebene des ViewModels getroffen werden, nicht auf einzelne Eigenschaften des ViewModels). Damit der Abgleich funktioniert, muss die *equals* und *hashCode* Methode des ViewModels korrekt implementiert werden.
Debugging	5	Durch die Möglichkeit von Redux Time Travel und der Verwendung einer Logging-Middleware können Fehler gezielt analysiert werden. Durch den zentralen State kann jederzeit eingesehen werden, in welchem Zustand die Applikation sich befindet.
Tooling	4	Für die Erstellung und Verwendung von Redux in Flutter benötigt man keine externen Tools. Möchte man Redux jedoch genauer debuggen, gibt es eine Reihe von Tools die dabei unterstützen. Diese Tools sind unter anderem von Drittanbietern und benötigen ein initiales Setup oder Anpassungen im Code.
Änderbarkeit und Erweiterbarkeit	5	Durch die Actions und Reducers ist klar definiert, von wo der State angepasst werden kann, wodurch ein nachträgliches Anpassen oder Erweitern erleichtert wird.
Architektur und einheitliche Vorgehensweise	5	Es gibt zwar nur einen globalen State, jedoch kann dieser je nach Anforderungen unterteilt werden. Die vordefinierte Struktur, wie Daten vom Store gelesen und in den Store geschrieben werden, ermöglicht dabei ein einheitliches Vorgehen innerhalb des Teams.

Tabelle 9.4: Bewertungen für Redux

9.3 Variante - BLoC

Für die Verwendung des BLoC Patterns ist grundsätzlich kein Package notwendig, jedoch gibt es einige Themen, die bei der Umsetzung beachtet werden müssen (z.B. Dependency Injection, Verwaltung von Streams etc.). Um diese Themen nicht manuell behandeln zu müssen, hat sich in der Community das Package *flutter_bloc* für die Umsetzung des BLoC Patterns hervorgehoben (für diese Arbeit wurde das Package in der Version 6.0.6 verwendet). Dieses Package baut dabei stark auf das Provider Package auf und bietet einige Hilfsmittel für den Aufbau und die Nutzung von Blocs. Dabei setzt sich ein Bloc aus mehreren Bestandteilen zusammen:

- **States** - Bei den States handelt es sich um die Zustände, welche durch einen Bloc abgebildet werden können. Diese sind z.B. TodosLoadInProgress, TodosLoadSuccess und TodosLoadFailure. Diese Zustände können in der UI unterschieden und anhand ihrer Eigenschaften verwendet werden.

- **Events** - Events beschreiben, wie sich der Inhalt eines Blocs ändern kann (Änderungen können nur durch Events entstehen). Beispiele wären TodoAdded, TodoUpdated oder TodoDeleted.

- **Bloc** - Der eigentliche Bloc verbindet die States und die Events. Durch einen Sink können Events einen Bloc übergeben werden, der anhand dessen einen abgeänderten State über einen Stream zurückliefert.

- **Barrel File** - Um alle erstellten Dateien eines Blocs einfach in anderen Klassen importieren zu können, empfiehlt es sich, ein sogenanntes Barrel File zu erstellen. Dieses kapselt alle Dateien und erleichtert somit das Importieren.

Für einen State wird jeweils eine abstrakte Klasse definiert und die jeweiligen Implementierungen für die vorhandenen Zustände. Für die Benutzerliste in der Beispielapplikation ergaben sich z.B. drei verschiedene Ausprägungen: für den initialen Zustand, den Ladevorgang und der resultierenden Benutzerliste (siehe Listing 9.10). Der initiale Zustand wird dabei genutzt um einmalig die Liste von der API zu holen, der Ladevorgang um dem Benutzer den Loading-Indicator anzuzeigen und die Benutzerliste um das Dropdown Menü mit Werten zu befüllen.

```
1 abstract class UsersState extends Equatable {
2   const UsersState();
3 
4   @override
5   List<Object> get props => [];
6 }
7 
```

```
class UsersInitial extends UsersState {}

class UsersLoadInProgress extends UsersState {}

class UsersLoadSuccess extends UsersState {
  final List<User> users;

  const UsersLoadSuccess([this.users = const []]);

  @override
  List<Object> get props => [users];
}
```

Listing 9.10: States in BLoC

Ähnlich wie bei den States werden auch die Events mit einer abstrakten Klasse sowie den notwendigen Implementierungen abgebildet. Im Falle der Benutzerliste gibt es nur ein Event, welches das Abrufen der Liste von der API auslöst (siehe Listing 9.11).

```
abstract class UsersEvent extends Equatable {
  const UsersEvent();

  @override
  List<Object> get props => [];
}

class LoadUserList extends UsersEvent {}
```

Listing 9.11: Events in BLoC

Im Bloc werden die States mit den Events verknüpft. Dabei wird von der Klasse *Bloc* des *flutter_bloc* Packages abgeleitet und anhand von Generics die verwendeten Klassen sowie über den Konstruktor ein initialer Zustand definiert. Die Klasse *Bloc* verlangt dabei, dass die Methode *mapEventToState* überschrieben wird, die anhand eines Events sich den angepassten State erwartet. Im Falle des User-Blocs wird dabei beim *LoadUserList*-Event zuerst der *UsersLoadInProgress* State geschickt, danach der API-Call durchgeführt und nach Vollendung der *UsersLoadSuccess* State mit der Benutzerliste gesendet (durch die Verwendung von Streams können beliebig viele States gesendet werden). Die Implementierung ist im Listing 9.12 ersichtlich:

```
class UsersBloc extends Bloc<UsersEvent, UsersState> {
  UsersBloc() : super(UsersInitial());

  @override
```

```
  Stream<UsersState> mapEventToState(UsersEvent event) async*
      {
    if (event is LoadUserList) {
      yield* _mapUsersLoading();
    }
  }

  Stream<UsersState> _mapUsersLoading() async* {
    yield UsersLoadInProgress();
    final result = await ApiClient.getUsers();
    yield UsersLoadSuccess(result.userList);
  }
}
```

Listing 9.12: Implementierung eines Blocs

Um die Werte eines Blocs im Widget-Tree verwenden und auf Änderungen reagieren zu können, wird das *BlocBuilder* Widget des Package verwendet. Das Widget bekommt über Generics den zu verwendeten Bloc und State mitgeteilt, anhand dessen der aktuelle State zur Verfügung gestellt wird. Das Widget funktioniert dabei gleich wie ein *StreamBuilder*. Sobald der Stream innerhalb des Blocs einen neuen Wert bekommt, wird die *builder* Funktion des Widgets ausgeführt. Innerhalb dieser Funktion hat man Zugriff auf den aktuellen Zustand und kann auf dessen Werte zugreifen. Anhand des Beispiels für die Benutzerliste wird ein Loading-Indicator angezeigt, solange noch keine Daten vorhanden sind. Sollte sich der Bloc im initialen Zustand befinden, wird ein LoadUserList Event gesendet, welches den API-Call auslöst. Sobald die Daten vorhanden sind, wird das Dropdown Menü mit den Benutzern angezeigt (siehe Listing 9.13).

```
BlocBuilder<UsersBloc, UsersState>(
      builder: (context, state) {
        if (state is! UsersLoadSuccess) {
          if (state is UsersInitial) {
            BlocProvider.of<UsersBloc>(context).add(
                LoadUserList());
          }
          return CircularProgressIndicator();
        }

        final userList = (state as UsersLoadSuccess).users;
        return DropdownButton<User>(
          value: selectedUser,
          hint: Text('User ausw hlen'),
          items: userList
              .map((user) => DropdownMenuItem(
```

```
16                      child: Text(user.name),
17                      value: user,
18                    ))
19                .toList(growable: false),
20            onChanged: onChanged,
21          );
22        },
23      )
```

Listing 9.13: Verwendung des BlocBuilders

Da das Package für die Dependency Injection auf Provider setzt, ist die Erstellung und der Zugriff auf die Blocs ident zum Provider Package. Für die Instanziierung der Blocs gibt es z.B. den *BlocProvider* oder *MultiBlocProvider* und zugegriffen werden kann über *BlocProvider.of<T>(context)* oder Widgets wie dem *BlocBuilder* (es handelt sich hierbei um Wrapper für das Provider Package, die Verwendung ist ident zum Provider Package). Um das Testen der Blocs und der darunter liegenden Streams zu vereinfachen, gibt es das Package *bloc_test*. Damit lassen sich die erstellten Blocs mit Berücksichtigung des zeitlichen Ablaufs der Streams abstrahiert testen (Beispiel ist in Listing 9.14 ersichtlich).

```
1 blocTest(
2   'Delete Todo',
3   build: _givenTodoBlocWithOneEntry,
4   act: (bloc) => bloc.add(TodoDeleted(_getOpenTodo().id)),
5   expect: [isA<TodosList>()],
6   verify: (TodosBloc block) => expect((block.state as
      TodosList).todos.length, 0),
7 );
```

Listing 9.14: Testen eines Blocs

Objektive Kennzahlen	
Kennzahl	**Wert**
Zyklomatische Komplexität	72
Anzahl ausführbarer Codezeilen	237
Anzahl der Methoden	39
Maximale Anzahl an Argumenten	2
Wartbarkeitsindex	82

Tabelle 9.5: Objektive Kennzahlen - BLoC

Kriterium	Wert	Begründung
Testbarkeit	5	Durch die strikte Trennung von Funktionen in Blocs können diese sehr gut abgekapselt getestet werden. Durch das zusätzliche *bloc_test* Package können die Abläufe in einem Stream sehr abstrahiert getestet werden. Die klare Trennung zwischen Businesslogik und Benutzeroberfläche ermöglicht dabei die Erstellung von Unit Tests.
Integration im Framework	5	Der BLoC Ansatz bzw. das *flutter_bloc* Package baut stark auf den reaktiven Ansatz von Flutter und dessen Streams auf. Durch den Einsatz von Provider sind die Blocs zusätzlich zum Widget-Tree gebunden, was das Schließen von nicht mehr benötigten Streams und Ressourcen vereinfacht.
Verbreitung	3	Der Ansatz ist in der Community zwar stark vertreten (1336 Likes auf `pub.dev` mit Stand 03.11.2020), jedoch ist es ein spezifischer Lösungsansatz für das Flutter Framework. Für Einsteiger ist es zusätzlich eine Hürde, dass für eine korrekte Implementierung ein vertieftes Wissen über das Flutter Framework und die Stream API notwendig ist.
Dokumentation	5	Sowohl für den BLoC Ansatz als auch zur Implementierung mittels *flutter_bloc* Package gibt es ausführliche Dokumentationen, Beispielimplementierungen und Tutorials.

Widget Rendering	4	Laut Definition sollten Blocs so klein wie sinnvoll gehalten werden, was sich positiv auf die Anzahl an Rebuilds auswirkt. Zusätzlich zum automatischen Check, ob sich der State verändert hat, kann man über die Methode *buildWhen* des *BlocBuilders* bestimmen, wann ein Widget neu gezeichnet werden soll.
Debugging	2	Abgesehen vom *BlocObserver*, mit welchem man alle Änderungen an einem Bloc protokollieren kann, bietet das Package wenige Hilfsmittel für das Debugging (Erweiterungen über Drittanbieter aber möglich). Da sich der State aus Werten in einem Stream ergibt, kann es schwierig sein den aktuellen Zustand der Applikation zu bestimmen.
Tooling	4	Das Package kann ohne externe Tools verwendet werden. Erweiterungen für das automatische Generieren der notwendigen Dateien oder zusätzliche Hilfsmittel für das Debugging können über Drittanbieter integriert werden.
Änderbarkeit und Erweiterbarkeit	5	Jede komplexere Komponente besitzt laut Definition ihren eigenen Bloc. Die klare Trennung von Verantwortlichkeiten und Schnittstellen bzw. Events ermöglichen es, nachträgliche Anpassungen und Erweiterungen besser durchführen zu können. Durch die Unabhängigkeit zwischen den Blocs können diese leichter ausgetauscht, entfernt oder angepasst werden.
Architektur und einheitliche Vorgehensweise	5	Der Aufbau mittels BLoC ermöglicht es, unabhängige Widgets zu erstellen, welche durch ihren explizit zugewiesenen Bloc gut wiederverwendbar sind. Die klaren Schnittstellen in Form der Events und States geben dabei ein einheitliches Vorgehen vor, welches auch auf lange Sicht eine einheitliche Codebasis ermöglicht.

Tabelle 9.6: Bewertungen für BLoC

9.4 Variante - MobX

Für die Umsetzung der Beispielapplikation mittels MobX wurde auf vier Packages zurückgegriffen:

- **mobx** (Version 1.2.1+4) für die MobX Basis.
- **flutter_mobx** (Version 1.1.0+2) um die Stores von MobX über das *Observer* Widget nutzen zu können.
- **mobx_codegen** (Version 1.1.2) um die Annotations *@observable*, *@computed* und *@action* nutzen zu können.
- **get_it** (Version 5.0.1) als Dependency Injection, um die MobX Stores zur Verfügung zu stellen.
- **build_runner** (Version 1.10.1) um die notwendigen Dateien anhand der *mobx_codegen* Annotations zu generieren.

In MobX werden die benötigten Zustände in sogenannten Stores gruppiert. Ein solch ein Store beinhaltet dabei alle Observables, Computed Values und Actions (wie in Kapitel 6.2.6 beschrieben). Anhand des Package wird ein Store dabei als abstrakte Klasse definiert, welche das Store Mixin beinhaltet. Innerhalb dieser Klasse können anhand der Annotations die benötigten Eigenschaften definiert werden. Um von der abstrakten Klasse eine konkrete Implementierung zu bekommen, wird eine zusätzliche Klasse erstellt, welche von der abstrakten ableitet und ein weiteres Mixin mit den Implementierungen für die Observables enthält (ein Auszug aus dem Store für die Todo-Liste ist in Listing 9.15 ersichtlich). Das Mixin mit den Implementierungen wird dabei über den *build_runner* erstellt, welcher auf die Annotations zurückgreift. Da es sich hier um Code Generierung handelt, müssen bei jeder Änderung die Dateien neu erstellt werden (über den Befehl `flutter packages pub run build_runner build`).

```
1 class TodoListState extends _TodoListState with
     _$TodoListState {}
2
3 abstract class _TodoListState with Store {
4   @observable
5   ObservableList<Todo> todos = ObservableList<Todo>();
6
7   @computed
8   int get todoCount => todos.length;
9
```

```
10   @action
11   void setAllDone() {
12     todos.forEach((todo) => todo.isDone = true);
13   }
14 }
```

Listing 9.15: Definition eines MobX Stores

Um den MobX Store mit einem Widget zu verbinden, wird das *Observer* Widget verwendet. Dabei erkennt das Widget automatisch, welche Observables in seiner *builder* Methode verwendet werden und zeichnet somit das Widget bei jeder Änderung automatisch neu. Die automatische Erkennung erfolgt nur für die aktuelle Hierarchie, verschachtelte Widgets müssen explizit mit einem *Observer* Widget versehen werden. Am Beispiel der Todo Liste erkennt man, dass sowohl die ListView (wenn Todos hinzukommen oder entfernt werden) als auch das *TodoListTile* (wenn ein Todo als erledigt oder offen markiert wird) separat in ein *Observer* Widget gewrappt werden muss (siehe Listing 9.16).

```
1  Observer(
2    builder: (_) {
3      return ListView.builder(
4        itemCount: todos.length,
5        itemBuilder: (_, index) {
6          return Observer(
7            builder: (context) {
8              return TodoListTile(
9                id: todos[index].id,
10               title: todos[index].title,
11               user: todos[index].user?.name,
12               selected: todos[index].isDone,
13               onChanged: (isDone) => todos[index].isDone =
                    isDone,
14             );
15           },
16         );
17       },
18     );
19   },
20 )
```

Listing 9.16: Verwendung eines MobX Observer Widgets

Um die Stores zentral zur Verfügung zu stellen, wurde das Package *get_it* als Dependency Injection Lösung gewählt (MobX schränkt hier auf keinen Ansatz ein, man könnte z.B. auch Provider verwenden). Man registriert hierbei initial ein Singelton

mittels *registerSingleton<T>()* der benötigten Klasse und kann danach in der ganzen Applikation durch *get<T>()* darauf zugreifen. Im Zuge des Projektes wurden Hilfsmethoden erstellt, welche den Zugriff erleichtern (siehe Listing 9.17):

```
final _di = GetIt.instance;

void initDi() {
  _di.registerSingleton<TodoListState>(TodoListState());
  _di.registerSingleton<UserListState>(UserListState());
}

TodoListState get todoList => _di.get<TodoListState>();
UserListState get userList => _di.get<UserListState>();
```

Listing 9.17: Verwendung des get_it Package

Da auf die Werte der Observables in einem Store direkt zugegriffen werden kann, besteht die Möglichkeit, einfache Unit Tests durchzuführen. Durch die Abstraktion der Streams bzw. Observables durch das MobX Package können die Tests anhand synchroner Abläufe getestet werden. Der Test für das Löschen eines Todos ist in Listing 9.18 ersichtlich.

```
test('Delete Todo', () {
  final todoState = _givenTodoListStateWithOpenTodo();
  expect(todoState.todoCount, 1);
  todoState.deleteTodo(id: todoState.todos.first.id);
  expect(todoState.todoCount, 0);
});
```

Listing 9.18: Test eines MobX Stores

Objektive Kennzahlen	
Kennzahl	**Wert**
Zyklomatische Komplexität	34
Anzahl ausführbarer Codezeilen	171
Anzahl der Methoden	23
Maximale Anzahl an Argumenten	1
Wartbarkeitsindex	77

Tabelle 9.7: Objektive Kennzahlen - MobX

Kriterium	Wert	Begründung
Testbarkeit	5	Durch die Abstraktion der Streams bzw. Observables wird das Testen vereinfacht. Die Trennung von Funktionen und Verantwortungen in eigene Stores und die damit verbundene Trennung zwischen Benutzeroberfläche und Businesslogik ermöglicht es, einzelne Teile unabhängig testen zu können.
Integration im Framework	3	MobX bietet eine Abstraktion von Observables für die Implementierung eines reaktiven State Managements. Über die bereitgestellten Widgets sind diese Observables leicht zu verwenden, jedoch können diese nicht mit anderen Packages oder Funktionen des Frameworks kombiniert werden.
Verbreitung	3	Mit 444 Likes für *mobx* und 233 für *flutter_mobx* auf `pub.dev` (Stand 05.11.2020) ist die Variante in der Flutter Community im Vergleich zu den anderen nicht stark vertreten. Da der Ansatz aber aus der Webentwicklung kommt, können Erfahrungen und das Wissen über MobX in Flutter Projekten genutzt werden, da die Grundkonzepte technologieunabhängig sind.
Dokumentation	5	Zusätzlich zur allgemeinen MobX Dokumentation stellt das Package für Dart bzw. Flutter eine zusätzliche und sehr ausführliche Dokumentation über `mobx.netlify.app` zur Verfügung (Beispielapplikationen, Best Practices, API-Dokumentationen, etc.).
Widget Rendering	5	Anhand des *Observer* Widgets wird genau festgelegt, welcher Teil des Widget-Trees bei Änderungen aktualisiert werden soll. Der Vorteil dabei ist, dass dieses selektive neu Zeichnen implizit durch die Verwendung der Observables geschieht. Dabei wird das Widget nicht bei jeder Änderung des Stores neu gezeichnet, sondern nur bei jenen Daten, welche es auch verwendet.

Debugging	3	Durch die vorhandene Abstraktion der Observables durch das Package sind manche Abläufe oder Ergebnisse schwer nachvollziehbar. Da die Werte eines Stores nicht zwingend über eine Action angepasst werden müssen, gibt es keine zentrale Stelle, an jener der Store seinen Zustand ändert. Für ein besseres Verständnis des Ablaufs in der Applikation gibt es die Möglichkeit, über die *autorun* Funktion Änderungen am Store zu protokollieren oder ein zusätzliches Tool wie den *Spy* von MobX zu nutzen.
Tooling	3	Das Package baut stark auf Code Generierung auf und benötigt dazu sowohl das *build_runner* als auch das *mobx_codegen* Package. Bei jeder Änderung am Store müssen die dazugehörigen Dateien neu generiert werden.
Änderbarkeit und Erweiterbarkeit	4	Durch die Möglichkeit anhand eines vorhandenen States einen Derived State ableiten und die Stores hierarchisch aufbauen zu können, ist die Trennung von einzelnen Funktionen in separate Stores möglich. Da Änderungen am Store nicht an einer zentralen Stelle durchgeführt werden müssen, können bei größeren Projekten oder Stores Anpassungen und Erweiterungen komplexer werden.
Architektur und einheitliche Vorgehensweise	4	MobX erleichtert die Verwendung von Streams bzw. Observables, lässt aber einige Architekturentscheidungen dem Entwickler offen. Durch die Verwendung von Observables und dem *Observer* Widget wird dabei eine einheitliche Vorgehensweise definiert, wie ein Zustand in der Applikation abgebildet werden soll.

Tabelle 9.8: Bewertungen für MobX

10 Analyse und Interpretation der Ergebnisse

Die Ergebnisse der einzelnen Varianten werden in diesem Kapitel zusammengefasst und gegenübergestellt. Anhand der Auswertung werden gewonnene Erkenntnisse beschrieben und die Forschungsfrage beantwortet. Die Rohdaten zu den Kennzahlen können dabei auf der beigelegten CD eingesehen werden, ein HTML Report steht unter *Ergebnisse/MA_Project/metrics* zur Verfügung.

10.1 Auswertung

Anhand der objektiven Kennzahlen, welche über die statische Codeanalyse generiert wurden, sind sowohl Gemeinsamkeiten als auch grobe Unterschiede in den Varianten ersichtlich. Die nachfolgend genannten Werte können in der Tabelle 10.1 eingesehen werden. Hinsichtlich der Zyklomatischen Komplexität haben Provider und MobX mit 36 und 34 die geringsten Werte und Variante BLoC mit 72 und Redux mit 50 die höchsten. Die erhöhte Komplexität lässt sich darauf rückschließen, dass mittels BLoC oder Redux eine Architektur vorgegeben wird, welche zu einem gewissen Mehraufwand in der Entwicklung und Abbildung von Funktionen führt. Dieser Mehraufwand spiegelt sich aber positiv beim Wartbarkeitsindex wider, da durch die klarere Struktur die Abhängigkeiten besser definiert werden und somit die Wartbarkeit erleichtert wird. Mit einem Ergebnis von 82 hat BLoC beim Wartbarkeitsindex den höchsten Wert, gefolgt von Redux(78), MobX(77) und Provider(66). Die Anzahl an ausführbaren Codezeilen deckt sich dabei mit der Komplexität, da BLoC mit 237 den höchsten Wert besitzt und Provider bzw. MobX mit 186 und 171 den geringsten. Auch die Anzahl an Methoden ist beim BLoC Ansatz mit 39 am höchsten, die restlichen Varianten befinden sich zwischen 23 und 26. Die Kennzahl über die maximale Anzahl an Argumenten ist bei dieser Erhebung nicht aussagekräftig. Zwar hat Redux mit drei Argumenten den höchsten Wert, jedoch handelt es sich um vereinzelte Ausreißer. Alle Varianten liegen bei der Anzahl an Argumenten eng beieinander, mit Werten zwischen 1 und 3.

Kennzahl	Provider	Redux	BLoC	MobX
Zyklomatische Komplexität	36	50	72	34
Anzahl ausführbarer Codezeilen	186	224	237	171
Anzahl der Methoden	25	26	39	23
Maximale Anzahl an Argumenten	2	3	2	1
Wartbarkeitsindex	66	78	82	77

Tabelle 10.1: Übersicht über die Objektive Kennzahlen

Die subjektive Bewertung der Varianten anhand der Umsetzung hat Aufschluss über die Stärken und Schwächen der einzelnen Ansätze gebracht. Die Werte für die nachfolgend genannten Begründungen können in der Tabelle 10.2 eingesehen werden. Durch eine klare Trennung von Businesslogik und Benutzeroberfläche sind bei allen Varianten Unit Tests möglich. Dabei hat jede Variante eigene Vorteile, wie z.B. das unabhängige Testen von Funktionen mittels Provider oder MobX, die vereinfachte Erstellung von Testkonstellationen durch einen zentralen State in Redux oder das abstrahierte Testen eines kompletten asynchronen Ablaufs mittels *bloc_test* Package in BLoC. Hinsichtlich der Integration in das Framework haben sowohl Provider als auch BLoC die beste Bewertung, da sie auf die Funktionen und Best Practices von Flutter aufbauen. Der Aufbau von Redux schränkt in einigen Gebieten die Verwendung von Funktionen der Sprache bzw. Technologie ein (z.B. Streams). MobX setzt dabei auf eine eigene Implementierung von Observables, welche nicht mit der Standard-API von Dart kompatibel ist. Betrachtet man die Verbreitung der Ansätze, teilen sich Provider und Redux den ersten Platz. Provider ist laut Likes auf `pub.dev` und durch die Empfehlung auf der offiziellen Website das am meist genutzte Package, jedoch handelt es sich um eine spezielle Lösung für Flutter. Redux hingegen hat bereits eine große Community durch die Webentwicklung, wodurch es Einsteigern leichter fallen kann auf Flutter umzusteigen. Für jede betrachtete Variante gibt es dabei ausführliche Dokumentationen, wobei Provider als einzige Variante nicht die komplette Punktezahl bekommen hat. Ein sehr essentieller Punkt bezüglich Performance ist die Tatsache, wann und wie bzw. wie oft das Widget Rendering stattfindet (Details in Kapitel 3.5). Hier schneiden alle Varianten bis auf Redux gut ab, wobei MobX die beste Bewertung bekommen hat. In Redux wird standardmäßig bei jeder Änderung des ViewModels das komplette Widget neu gezeichnet. Wenn ein ViewModel mehrere Daten beinhaltet, kann das zu unnötigen Rebuilds des UI führen, was sich negativ auf die Performance auswirkt. Es gibt zwar die Möglichkeit an einzelnen Stellen diesen Ablauf zu optimieren, jedoch muss das explizit für jedes Widget durchgeführt werden. MobX auf der anderen Seite erkennt in seinem *Observer* Widget automatisch,

welche Eigenschaften des Stores verwendet werden und zeichnet das Widget auch nur bei dessen Änderungen neu (der Entwickler muss sich um nichts kümmern, es geschieht implizit). Für die Fehlerfindung ist es essentiell, die Applikation debuggen zu können, wobei Provider und Redux die beste Bewertung dahingehend bekommen haben. Provider ermöglicht, dadurch dass es ein Widget ist, dass man die Werte jederzeit im Widget-Tree einsehen kann, auch ohne Debug-Modus. Redux hingegen stellt durch den zentralen Store, einheitlichen Schnittstellen und einer Time-Travel Funktion Tools zur Verfügung um die Applikation gezielt debuggen zu können. Für die Verwendung der einzelnen Varianten ist bei allen bis auf MobX kein externes Tool oder ähnliches notwendig. MobX hingegen baut auf Code Generierung auf, wodurch zusätzliche Tools und Workflows notwendig sind. Ein zentraler Punkt in Bezug auf State-Management ist die Änder- bzw. Erweiterbarkeit des States und die allgemeine Architektur und dessen Verwendung. Da alle betrachteten Varianten eine klare Trennung von Verantwortlichkeiten haben, ist die Änder- bzw. Erweiterbarkeit grundsätzlich gegeben, jedoch haben Redux und BLoC eine bessere Bewertung durch ihre einheitlichen Schnittstellen bekommen. Bei der Architektur hingegen gab es einen Ausreißer. Da das Provider Package lediglich das Bereitstellen von Eigenschaften und Methoden im Widget-Tree ermöglicht und keine Architektur vorgibt, wurde es in diesem Bereich schlechter bewertet als die restlichen Varianten. Betrachtet man die Gesamtbewertungen, hat die BLoC Variante mit knappem Vorsprung die beste Bewertung.

Kriterium	**Provider**	**Redux**	**BLoC**	**MobX**
Testbarkeit	5	4	5	5
Integration im Framework	5	2	5	3
Verbreitung	4	4	3	3
Dokumentation	4	5	5	5
Widget Rendering	4	2	4	5
Debugging	5	5	2	3
Tooling	5	4	4	3
Änderbarkeit und Erweiterbarkeit	4	5	5	4
Architektur und einheitliche Vorgehensweise	1	5	5	4
Gesamt	37	36	38	35

Tabelle 10.2: Übersicht über die Bewertungen

10.2 Erkenntnisse

Die Auswahl eines State-Management Ansatzes ist essentiell und kann den weiteren Verlauf eines Projekts oder einer Applikation stark beeinflussen. Aus diesem Grund muss die Auswahl zum Projekt und zum Team passen. Unabhängig der Kennzahlen und technischen Vor- und Nachteile gibt es Präferenzen, Wissen und Erfahrungen im Team, welche berücksichtigt werden müssen. Die Hinzunahme von technischen oder allgemeinen Aspekten hilft dabei objektive Punkte zu berücksichtigen, um den optimalen Ansatz für die Problemstellung zu finden. Diese können dabei grob in zwei Kategorien unterteilt werden, den wirtschaftlichen und technologischen Aspekten. Bei den wirtschaftlichen Entscheidungsgründen handelt es sich um all jene, welche direkt den wirtschaftlichen Erfolg beeinflussen können. Diese beinhalten z.B. die Verbreitung der Variante oder die Qualität der Dokumentation, wodurch die Schwierigkeit neue Mitarbeiter zu finden oder zu schulen beeinträchtigt wird. Time-to-Market und der geplante Umfang der Software sollten zusätzlich bei der Auswahl betrachtet werden, da die Architektur bzw. Variante für die jeweiligen Anforderungen Vor- und Nachteile besitzen kann. Bei den technologischen Entscheidungsgründen spielen Performance, Testbarkeit sowie das Tooling und Debugging eine entscheidende Rolle um die gewünschte Qualität auch über einen längeren Zeitraum sicherstellen zu können. Die gewählte Variante sollte dabei möglichst kompatibel mit den restlichen Funktionalitäten des Frameworks sein und die Best Practices der Technologie unterstützen. Betrachtet man die Ergebnisse der objektiven Kennzahlen in Kombination mit den bewerteten Kriterien, so lassen sich zwei mögliche Empfehlungen daraus ableiten. Mit einer Gesamtbewertung der Kriterien von 38 liegt die BLoC Variante auf dem ersten Platz und hat dabei auch den höchsten Wartbarkeitsindex aller Varianten. Diesen Werten gegenübergestellt ist die Variante aber jene mit der höchsten Zyklomatischen Komplexität, Anzahl an ausführbarer Codezeilen und Anzahl an Methoden. Die hohen Einstiegshürden von BLoC und die durch die Architektur resultierenden Mehraufwände macht diese Variante vor allem für größere bzw. langfristige Projekte relevant, in welchem die Teammitglieder bereits Erfahrung mit der Technologie haben und eine klare sowie einheitliche Architektur zwingend notwendig ist. Auf dem zweiten Platz, anhand der Gesamtbewertung der Kriterien von 37, liegt das Provider Package, welches bei den Kennzahlen der Zyklomatischen Komplexität, Anzahl ausführbarer Codezeilen und Anzahl an Methoden jeweils den zweitbesten Wert besitzt. Aus diesen Gründen bietet sich der Ansatz für Einsteiger sowie für kleine bis mittlere Projekte an. Die fehlende Architektur und einheitliche Vorgehensweise der Variante können dabei durch interne Richtlinien und Design Patterns abgedeckt werden, wodurch der Ansatz auch für größere Projekte und erfahrenere Entwickler relevant bleibt. Sowohl BLoC als auch Provider bauen dabei auf die Funktionalitäten des Frameworks auf und unterstützen die genannten Best Practices.

11 Zusammenfassung

Flutter stellt durch seine Performance und der steilen Lernkurve sowohl für Einsteiger als auch Erfahrene eine relevante Alternative zu nativen Apps dar. Die Frage nach einer geeigneten State-Management Variante kann dabei auf den ersten Blick nicht immer schnell beantwortet werden, da eine Vielzahl an möglichen Lösungen zur Verfügung stehen. Im Zuge dieser Arbeit wurden die gängigsten Varianten genauer betrachtet (Provider, Redux, BLoC und MobX) und anhand einer Beispielapplikation umgesetzt. Dabei wurden sowohl technologische als auch wirtschaftliches Entscheidungsgründe in die Bewertung mit aufgenommen, wobei Best Practices des Frameworks berücksichtigt wurden. Eine objektive Bewertung der einzelnen Varianten wurde dabei über die Berechnung von Kennzahlen aus einer statischen Codeanalyse durchgeführt. Da durch solche Kennzahlen alleine keine Aussage getroffen werden kann, wurden zusätzlich Kriterien definiert, anhand dieser die Umsetzung bewertet wurde. Durch die Kombination von den Kennzahlen aus der statischen Codeanalyse mit den Kriterien für die Bewertung konnten zwei Empfehlungen abgeleitet werden. Die BLoC Variante hat dabei sowohl bei den Bewertungen der Kriterien, als auch bei der Kennzahl für den Wartbarkeitsindex am besten abgeschnitten. Durch den erhöhten Aufwand und die aus der Architektur resultierende Komplexität (spiegelt sich in den restlichen Kennzahlen wider) eignet sich dieser Ansatz vor allem bei größeren Projekten mit erfahrenen Flutter Entwicklern. Als Alternative für diesen Ansatz empfiehlt sich das Provider Package, welches bei der Bewertung der Kriterien nur einen Punkt hinter BLoC liegt und dabei bei den restlichen Kennzahlen rund um Komplexität und Aufwand bei der Entwicklung besser abschneidet.

Akronyme

AOT	Ahead Of Time (compiler)
API	Application Programming Interface
BLoC	Business Logic Components
CSS	Cascading Style Sheets
FPS	Frames Per Second
GPS	Global Positioning System
HTML	Hypertext Markup Language
JIT	Just In Time (compiler)
JSON	JavaScript Object Notation
JSX	JavaScript XML
OEM	Original Equipment Manufacturer
REST	Representational State Transfer
SDK	Software Development Kit
UI	User Interface
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

Abbildungsverzeichnis

3.1 Aufbau Web und OEM-Widgets (Quelle: vgl. Biessek, 2020) 8
3.2 Aufbau Flutter (Quelle: vgl. Biessek, 2020) 9
3.3 Widget Lifecycle (Quelle: vgl. Windmill, 2019) 11

4.1 Ansatz des deklarativen UI (Quelle: vgl. flutter.dev, 2020d) 17

5.1 Reactive Manifesto (Quelle: vgl. Bonér, Farley, Kuhn Thompson, 2014) 19

6.1 Beispiel für Lift state up (Quelle: vgl. flutter.dev, 2020e) 25
6.2 Ablaufbeschreibung von Redux (Quelle: vgl. Garreau, 2018) 33
6.3 Datenfluss in einem BLoC (Quelle: vgl. Boelens, 2018) 35

Tabellenverzeichnis

3.1 Lifecycle Methoden von Stateful-Widgets (Napoli, 2020) 10
3.2 Sprachenunterschiede zwischen vergleichbaren Frameworks (Payne, 2019) 12
3.3 Kompromisse bei unterschiedlichen Testtypen (flutter.dev, 2020d) . . . 15

8.1 Funktionsumfang der Beispielapplikation 43
8.2 Subjektive Bewertungskriterien 44

9.1 Objektive Kennzahlen - Provider 48
9.2 Bewertungen für Provider 50
9.3 Objektive Kennzahlen - Redux 54
9.4 Bewertungen für Redux 55
9.5 Objektive Kennzahlen - BLoC 60
9.6 Bewertungen für BLoC 61
9.7 Objektive Kennzahlen - MobX 64
9.8 Bewertungen für MobX 66

10.1 Übersicht über die Objektive Kennzahlen 68
10.2 Übersicht über die Bewertungen 69

Listings

3.1 Beispiel für ein Anti-Pattern bei der Erstellung von Widgets 13
3.2 Beispiel für die Korrektur des Anti-Patterns bei der Erstellung von Widgets 14

4.1 Imperative vs deklarative Programmierung 16

5.1 Anwendung von Transformationen auf Streams 20
5.2 Anwendung von StreamTransformer auf Streams 20
5.3 Erstellung eines Streams mittels async* Funktion 21
5.4 Verwendung von StreamController 21
5.5 Verwendung des StreamBuilders 22

6.1 Ändern des Zustandes mittels *setState()* 24
6.2 Verwendung von Callback Funktionen - Lifting State up 26
6.3 Verwendung eines InheritedWidgets 28
6.4 Verwendung eines InheritedModels 29
6.5 Erstellung eines ChangenNotifiers 30
6.6 Verwendung eines ChangeNotifierProviders bzw. MultiProviders . . . 31
6.7 Zugriff auf Models mittels Consumer-Widget 31
6.8 Verwendung der MobX Annotations 36

9.1 Implementierung des UserProvider 45
9.2 Auszug aus der Implementierung des TodoProvider 46
9.3 Provider im Widget-Tree zur Verfügung stellen 47
9.4 Verwendung des Consumer Widgets 47
9.5 Testen eines Providers 48
9.6 Einsatz von Actions und Reducern 51
9.7 Asynchrone Funktionen durch Middleware 52
9.8 Container Widget in Redux 52
9.9 Testen von Redux 53
9.10 States in BLoC 56
9.11 Events in BLoC 57
9.12 Implementierung eines Blocs 57
9.13 Verwendung des BlocBuilders 58
9.14 Testen eines Blocs 59
9.15 Definition eines MobX Stores 62
9.16 Verwendung eines MobX Observer Widgets 63

9.17 Verwendung des get_it Package 64
9.18 Test eines MobX Stores 64

Literaturverzeichnis

Alessandria, S. (2020). *Flutter projects*. Packt.

Andrade, P. R. M., Albuquerque, A., Frota, O. F., Silveira, R. V. & da Silva, F. A. (2015). Cross platform app: a comparative study. *CoRR*, *abs/1503.03511*. Zugriff auf `http://arxiv.org/abs/1503.03511`

Barker, C. (2020). *Learn swiftui*. Packt Publishing.

Bennett, J. (2018). *Xamarin in action: Creating native cross-platform mobile apps* (1. Aufl.). Manning Publications.

Biessek, A. (2020). *Flutter for beginners*. Packt.

Biørn-Hansen, A. & Ghinea, G. (2018). Bridging the gap: Investigating device-feature exposure in cross-platform development. In *Proceedings of the 51st hawaii international conference on system sciences.*

Bird, C., Menzies, T. & Zimmermann, T. (2015). *The art and science of analyzing software data* (1. Aufl.). Morgan Kaufmann.

Boelens, D. (2018). *Reactive programming - streams - bloc.* Zugriff am 14.09.2020 auf `https://www.didierboelens.com/2018/08/reactive-programming-streams-bloc/`

Bonér, J., Farley, D., Kuhn, R. & Thompson, M. (2014). *Reactive manifesto.* Zugriff am 28.06.2020 auf `https://www.reactivemanifesto.org/pdf/the-reactive-manifesto-2.0.pdfe`

Cheng, F. (2018). *Build mobile apps with ionic 4 and firebase: Hybrid mobile app development* (2nd ed. Aufl.). Apress.

Cheng, F. (2019). *Flutter recipes: Mobile development solutions for ios and android* (1st ed. Aufl.). Apress.

dart.dev. (2018). *Creating streams.* Zugriff am 02.05.2020 auf `https://dart.dev/articles/libraries/creating-streams`

dart.dev. (2020a). *Customizing static analysis.* Zugriff am 06.10.2020 auf `https://dart.dev/guides/language/analysis-options`

dart.dev. (2020b). *Using constructors.* Zugriff am 28.06.2020 auf `https://dart.dev/guides/language/language-tour#using-constructors`

dash-overflow.net. (2020). *Provider dokumentation.* Zugriff am 14.09.2020 auf `https://pub.dev/documentation/provider/4.3.2/`

developer.android. (2020). *Jetpack compose.* Zugriff am 28.06.2020 auf `https://developer.android.com/jetpack/compose`

Dinkevich, B. & Gelman, I. (2017). *The complete redux book* (1st Edition Aufl.). Leanpub.

flutter.dev. (2020a). *Desktop support for Flutter.* Zugriff am 02.05.2020 auf https://flutter.dev/desktop

flutter.dev. (2020b). *Differentiate between ephemeral state and app state.* Zugriff am 28.08.2020 auf https://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app

flutter.dev. (2020c). *Performance best practices.* Zugriff am 28.06.2020 auf https://flutter.dev/docs/perf/rendering/best-practices

flutter.dev. (2020d). *Showcase.* Zugriff am 28.06.2020 auf https://flutter.dev/showcase

flutter.dev. (2020e). *Simple app state management.* Zugriff am 28.08.2020 auf https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple

flutter.dev. (2020f). *Start thinking declaratively.* Zugriff am 28.06.2020 auf https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative

flutter.dev. (2020g). *Technical overview.* Zugriff am 22.05.2020 auf https://flutter.dev/docs/resources/technical-overview#everythings-a-widget

flutter.dev. (2020h). *Testing.* Zugriff am 28.06.2020 auf https://flutter.dev/docs/testing

flutter.dev. (2020i). *Web support for Flutter.* Zugriff am 02.05.2020 auf https://flutter.dev/web

Garreau, M. & Faurot, W. (2018). *Redux in action*. Manning Publ.

Kifer, M. & Liu, Y. A. (2018). *Declarative logic programming: Theory, systems, and applications*. Association for Computing Machinery and Morgan Claypool Publishers.

Mainkar, P. & Giordano, S. (2019). *Google flutter mobile development quick start guide*. Packt.

Maintainability index range and meaning. (2007). Zugriff am 11.10.2020 auf https://docs.microsoft.com/en-us/archive/blogs/codeanalysis/maintainability-index-range-and-meaning

Mistrik, I., Bahsoon, R., Kazman, R. & Zhang, Y. (2014). *Economics-driven software architecture* (1. Aufl.). Elsevier Inc, Morgan Kaufmann.

mobx.netlify.app. (2020). *Mobx - core concepts.* Zugriff am 14.09.2020 auf https://mobx.netlify.app/concepts

Napoli, M. L. (2020). *Beginning flutter : a hands on guide to app development*. John Wiley Sons.

Paul, A. & Nalwaya, A. (2019). *React native for mobile development: Harness the power of react native to create stunning ios and android applications* (2nd ed. Aufl.). Apress.

Payne, R. (2019). *Beginning app development with flutter*. Apress.

Podila, P. & Weststrate, M. (2018). *Mobx quick start guide: Supercharge the client state in your react apps with mobx* (Paperback Aufl.). Packt Publishing.

pub.dev. (2020). *Dart code metrics.* Zugriff am 06.10.2020 auf https://pub.dev/documentation/dart_code_metrics/2.0.0/

Richard Bellairs. (2019). *What is code quality? and how to improve code quality.* Zugriff am 06.10.2020 auf https://www.perforce.com/blog/sca/what

-code-quality-and-how-improve-code-quality

Rieger, C. & Majchrzak, T. A. (2016). Weighted evaluation framework for cross-platform app development approaches. In S. Wrycza (Hrsg.), *Information systems: Development, research, applications, education* (S. 18–39). Cham: Springer International Publishing.

Rieger, C. & Majchrzak, T. A. (2019). Towards the definitive evaluation framework for cross-platform app development approaches. *Journal of Systems and Software*, *153*, 175 - 199. Zugriff auf http://www.sciencedirect.com/science/article/pii/S0164121219300743 doi: https://doi.org/10.1016/j.jss.2019.04.001

Savkin, V. (2017). *Managing state in angular applications using ngrx.* Zugriff am 28.08.2020 auf https://blog.nrwl.io/using-ngrx-4-to-manage-state-in-angular-applications-64e7a1f84b7b

Sebesta, R. W. (2016). *Concepts of programming languages 11th global ed.* Pearson.

Urma, R.-G., Fusco, M. & Mycroft, A. (2018). *Modern java in action: Lambda, streams, functional and reactive programming* (2. Aufl.). Manning.

Wieruch, R. (2017). *Taming the state in react. your journey to master redux and mobx*. LeanPub.

Windmill, E. (2019). *Flutter in action* (1. Aufl.). Manning Publications.

Xanthopoulos, S. & Xinogalos, S. (2013). A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th balkan conference in informatics* (S. 213–220). New York, NY, USA: Association for Computing Machinery. Zugriff auf https://doi.org/10.1145/2490257.2490292 doi: 10.1145/2490257.2490292

Zammetti, F. (2019). *Practical flutter: Improve your mobile development with google's latest open-source sdk* (1st ed. Aufl.). Apress.