h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik –

Evaluation von Zustandsverwaltungssystemen für das mobile Cross-Plattform-Framework Flutter

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Jonas Franz

Matrikelnummer: 760233

Referent	:	Prof. Dr. Kai Renz
Korreferent	:	Prof. Dr. Hans-Peter Wiedling

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 03. März 2022

Jonas Franz

ABSTRACT

Flutter has multiple approaches in managing the state of an application. Flutter is a popular cross-platform framework for the development of applications for iOS, Android, Web, Windows, Linux and macOS. It offers the advantage to develop one codebase for multiple platforms at once resulting in reduced costs and complexity. Flutter uses a declarative UI that renders widgets based on a given state. The state management of a Flutter application is one of the most important parts of an application influencing the architecture and structure of an application.

This thesis describes multiple already existing approaches for state management in Flutter. The objective is to determine which approach is the most suitable approach for managing state in Flutter. In order to meet this objective, an evaluation will be carried out by evaluating the state management approaches setState, InheritedWidget, BLoC, Provider, Riverpod, Redux and MobX. An example application will be implemented for every approach. Those applications will be assessed by qualitative and quantitative criteria based on the requirements of state management systems for Flutter.

The result, in addition of the evaluation assessments, is also a recommendation which approach to use for a specific use case.

ZUSAMMENFASSUNG

In Flutter gibt es diverse Ansätze und Lösungsmöglichkeiten, den Zustand einer mobilen Anwendung zu verwalten. Flutter ist ein beliebtes Cross-Plattform-Framework, mit dem sich Anwendungen für iOS, Android, Web, Windows, Linux und macOS erstellen lassen, die den gleichen Quelltext verwenden. Damit sollen in der Entwicklung Aufwände gespart werden können und die Komplexität reduziert werden. Flutter baut auf eine deklarative Benutzeroberfläche, welche anhand eines Zustands erstellt wird. Die Verwaltung dieses Zustands ist entscheidend für die Architektur und die Funktion einer Anwendung.

In dieser Ausarbeitung werden verschieden bereits etablierte Ansätze zur Verwaltung des Zustands einer Flutter-Anwendung dargestellt und untersucht. Dabei ist das Ziel, herauszufinden, welcher Ansatz am besten zum Verwalten des Zustands einer Flutter-Anwendung geeignet ist. Dafür wird eine Evaluation für die Zustandsverwaltungssysteme setState, InheritedWidget, BLoC, Provider, Riverpod, Redux und MobX durchgeführt. Grundlage dieser Evaluation ist die Entwicklung einer Beispielanwendung für jedes Zustandsverwaltungssystem und die Bewertung dieser anhand von qualitative und quantitativen Bewertungskriterien, die anhand der Anforderungen an Zustandsverwaltungssysteme definiert werden.

Das Ergebnis der Arbeit stellt neben den Resultaten der Evaluation auch eine Empfehlung dar, welches Zustandsverwaltungssystem für welchen Anwendungsfall am besten genutzt werden sollte.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

BLoC	Business Logic Components
MI	Maintainability Index
CI	Continous-Integration-System

Teil I

THESIS

1

EINLEITUNG

In der mobilen Software-Entwicklung dominieren seit Jahren die beiden mobilen Betriebssysteme iOS und Android. |[Osd] Dies macht es für die Entwicklung von mobilen Anwendungen erforderlich, ein Programm für beide Plattformen zu entwickeln. Da durch die Entwicklung von zwei ähnlichen Anwendungen aus rein technischen Gründen größere Aufwände bei der Entwicklung entsteht, sind seit mehreren Jahren sogenannte Cross-Plattform Technologien in Verwendung. Diese ermöglichen es, im Optimalfall mit der Entwicklung einer Anwendung mehrere Plattformen auf einmal abdecken zu können.

In dieser Ausarbeitung wird dabei die Cross-Plattform Technologie Flutter betrachtet. Die erste Version von Flutter wurde am 12. Mai 2017 [Bra17] veröffentlicht und unterstützt mit der Version 1.20 die Möglichkeit eine Anwendung sowohl als App auf Android und iOS, als Desktop-Anwendung für macOS, Linux und Windows sowie als Website auszuspielen. [Goo22a] Weitere Details zur Flutter-Technologie werden im entsprechenden Grundlagen-Kapitel behandelt.

Dabei wird mit dieser Ausarbeitung eine konkrete Problemstellung, die sich aus der Architektur von Flutter und ähnlichen Werkzeugen wie React behandelt. Die Problemstellung umgreift die Frage, welche Software-Architektur und Werkzeuge gewählt werden müssen, um den Zustand der Anwendung oder einzelnen Bestandteilen der Anwendung effizient und architekturell sinnvoll zu verwalten.

Anhand von entsprechend der Problemstellung definierten Bewertungskriterien wird durch eine Evaluation verschiedener bestehender Lösungsansätze untersucht, welcher sich am besten zur Lösung der Problemstellung eignet.

Im Folgenden wird in diesem Einleitungskapitel näher erläutert, welche Motivation zur Wahl dieses Thema führt, welche konkrete Zielsetzung verfolgt wird, welche Gliederung gewählt wird und anhand welcher Methodik vorgegangen wird.

1.1 MOTIVATION

Bei der Entwicklung von mobilen Anwendungen wie beispielsweise im Agenturgeschäft stellen sich immer wieder Architektur-Fragen, welche den Erfolg eines Projektes maßgeblich mitentscheiden können. Bei Flutter stellt sich hier beispielsweise die Frage, wie man am besten den Zustand der Anwendung verwalten kann. Auf diese Frage eröffnet sich den Entwickler*innen verschiedene Lösungsmöglichkeiten. Dabei wird aktuell weder seitens der offiziellen Entwicklungsdokumentation von Flutter noch in den Online-Publikationen eine eindeutige Empfehlung getroffen. Erschwerend bei der Entscheidungsfindung kommt hinzu, dass es im Moment eine große Anzahl an verschiedenen Ansätzen zur Lösung des Problems gibt. Alleine die offizielle Flutter Dokumentation umfasst nach aktuellen Stand eine Aufzählung von 13 verschiedenen Ansätzen, das Problem der Zustandsverwaltung (engl. state management) zu lösen. [Goo21d]

Die Entscheidung für einen bestimmten Architekturentwurf für die Zustandsverwaltung ist dabei eine schwierig umkehrbare Entscheidung, da sie Auswirkungen auf diverse Komponenten einer Anwendung hat. Dies kann unter anderem dazu führen, dass bei einer späteren Änderung der Zustandsverwaltung größere Änderungen an dem Quelltext der Anwendung nötig sind.

Dies zeigt auf, dass die Entscheidung für die dem Anwendungsfall am besten entsprechende Zustandsverwaltung eine signifikant wichtige Entscheidung im Entwicklungsablauf einer Flutter-Anwendung ist. Daher soll diese Ausarbeitung dabei unterstützen, auf Grundlage einer wissenschaftlichen Evaluation verschiedener Lösungsansätze die für die Entwicklung beste Option wählen zu können.

1.2 ZIELSETZUNG

Das übergeordnete Ziel dieser Ausarbeitung ist es, den für die Entwicklung von Flutter-Anwendungen oder bestimmten Anwendungsfällen am besten geeignete Ansatz für die Zustandsverwaltung zu finden. Im Detail soll dabei gezeigt werden, welche Lösungsansätze für welchen Anwendungsfall am besten, begründet geeignet sind.

Zur Erreichung dieses übergeordneten Zieles ist es erforderlich, einige Zwischenziele zu definieren. Das Erste Ziele ist dabei, zu analysieren, welche Anforderungen aktuell an die Zustandsverwaltung in Flutter gestellt werden, um im nächsten Schritt geeignete Kriterien für die Bewertung von Lösungsansätzen zu finden.

Dabei ist es auch ein Ziel, diese Bewertungsansätze möglichst objektiv definieren zu können, da die Auswahl von geeigneten Kriterien entscheidend ist für die Aussagekraft der Evaluation.

1.3 GLIEDERUNG

Für die Strukturierung der Arbeit, wurde die Arbeit in verschiedene Kapitel und Unterkapitel gegliedert. Zur Erreichung der Ziele und Unterziele bietet sich folgende Struktur an.

Zu Beginn wird mit der Einleitung ein grober Überblick über die Ausarbeitung sowie Informationen zur Methodik vermittelt.

Im Anschluss wird im Grundlagenkapitel anhand von Literaturrecherchen und Quelltext-Beispielen die für die Arbeit grundlegenden Konzepte dargestellt. Dabei insbesondere die Cross-Plattform-Technologie Flutter näher dargestellt und Architekturentwürfe für diese vertieft. Außerdem wird erläutert, welche Probleme die Entwicklung mit einem deklarativen Komponentensystem wie bei Flutter mit sich bringen und mögliche Lösungsansätze aus der Literatur diskutiert.

Darauf aufbauend werden im Analyse-Kapitel die Bewertungskriterien und Richtlinien für die spätere Evaluation erarbeitet.

Für die Durchführung der Evaluation wird im Entwurf- und Realisierungs-Kapitel ein möglicher Versuchsaufbau konstruiert, welcher geeignet ist, die beschriebenen Lösungsansätze anhand der im Analyse-Kapitel erarbeiteten Kriterien zu evaluieren. Dabei werden auch Aspekte der Realisierung protokolliert.

Im Evaluationskapitel werden die Ergebnisse aus dem Entwurfs- und Realisierung-Kapitel anhand der im Analyse-Kapitel gewonnen Kriterien bewertet und die Ergebnisse analysiert. Zum Schluss findet eine Gesamtbewertung der Lösungsansätze statt.

Abschließend wird die Bewertung in einem Gesamtzusammenhang gesetzt und eine Empfehlung abgegeben, sowie mögliche weitere Forschungsfragen diskutiert.

1.4 METHODIK

Für die Beschreibung der möglichen Lösungsansätze für die Zustandsverwaltung, wird per Literaturrecherche auf verschiedene Ansätze eingegangen und diese näher erläutert.

Im Anschluss wird mittels Literaturrecherche mögliche Anforderungen an Zustandsverwaltungssysteme zusammengefasst. Darauf aufbauend, werden Bewertungskriterien konstruiert, die auf die einzelnen Anforderungen einzahlen sollen.

Für die Umsetzung und Realisierung werden aus diesen Bewertungskriterien und Anforderungen neue implizite und explizite Anforderungen an eine Beispielanwendung konstruiert. Die Beispielanwendung soll später als Untersuchungsobjekt für die Evaluation dienen. Zusätzlich wird ein Versuchsaufbau konstruiert und beschrieben.

Für die Evaluation wird für jedes untersuchte Zustandsverwaltungssystem eine eigene Anwendung implementiert und anhand der definierten Kriterien untersucht. Dabei kommen hauptsächlich qualitative Bewertungsmethoden zum Einsatz. Zusätzlich dazu werden als quantitative Bewertungsmethoden der Maintainability Index eingesetzt und eine automatisierte Teststrecke zur Durchführung von Leistungstests eingerichtet. Alle Messungen und Bewertungen erfolgen in einer nachvollziehbaren, reproduzierbaren Docker-Umgebung.

2

GRUNDLAGEN

Dieses Kapitel befasst sich mit den Grundlagen für die nachfolgende Evaluation. Dabei wird sowohl Literatur zum Thema ausgewertet, als auch Konzepte an Hand von Beispielen erläutert.

2.1 FLUTTER

Um die konkrete Problemstellung im Detail verstehen zu können, ist es erforderlich, die grundlegende Plattform näher zu betrachten. Dabei wird zu Beginn erst ein Überblick über das Cross-Plattform-Werkzeug Flutter gegeben und im Anschluss detaillierter auf einzelne Aspekte, die für die Erfassung der Problemstellung von Relevanz sind, eingegangen.

2.1.1 *Einordnung der Rolle für die Entwicklung von Apps*

In diesem Abschnitt wird ein grober Überblick über die aktuelle Situation bei der Entwicklung von Apps gegeben. Flutter ist ein von dem US-amerikanischen Digitalunternehmen Google entwickeltes Cross-Plattform-Werkzeug, welches es ermöglichen soll, mobile Anwendungen (Apps) für die Smartphone-Betriebssysteme iOS und Android mit einer gemeinsamen Code-Basis zu entwickeln. [Ama18] In der etablierten App-Entwicklung ist es weit verbreitet, zwei separate Anwendungen für die beiden dominierenden Betriebssysteme iOS und Android zu entwickeln. Eine Analyse von Appfigures zeigt dabei, dass im Apple App Store 55 % der analysierten Apps auf Swift basieren, welches für die sogenannte native iOS-Entwicklung genutzt wird, und im Google Play Store 38 % der analysierten Apps auf Kotlin basieren, welches dem Pendant zu Swift für Android entspricht.[App22] Das Entwickeln von zwei getrennten Anwendungen bringt dabei das Erfordernis mit sich, Quelltext zu duplizieren, da für Android entwickelter Software nicht mit iOS kompatibel ist und umgekehrt.

Neben Flutter existieren auch andere Cross-Plattform Werkzeuge, die das gleiche Problem lösen möchten. Diese werden jedoch nicht in dieser Ausarbeitung näher betrachtet. Erwähnenswert jedoch sollte sein, dass das von

Meta entwickelte React Native SDK vom Design ähnlich ist und Flutter dieses "well preserved" ([Wu18]) hat, sodass auch dieses SDK von den hier in der Ausarbeitung beschriebenen Problemen betroffen sein kann und die Ergebnisse somit als Basis für eine ähnliche Evaluation verwendet werden könnten.

2.1.2 *Technischer Überblick*

Dieser Abschnitt vermittelt die technischen Grundlagen für die Flutter Technologie. Flutter und die damit entwickelten Anwendungen werden mit der Dart-Programmiersprache entwickelt. Dart lässt sich dabei mit den bereits etablierten Programmiersprachen wie Java oder JavaScript vergleichen wie in Listing 2.1 zu sehen ist und ist objektorientierte, optional statisch typisierte [Goo21f].

Listing 2.1: Aufbau eines einfachen Flutter-Widgets in Dart

```
import 'package:flutter/material.dart';

class ExampleWidget extends StatelessWidget {
    const ExampleWidget({Key? key}) : super(key: key);

    @override
    Widget build(BuildContext context) {
        return const Row(children: [Text('Hello World'), Text('123')];
    }
}
```

Von der Architektur her ist Flutter in einem Schichtentwurf aufgebaut, wie in Abbildung 2.1 zu sehen ist.

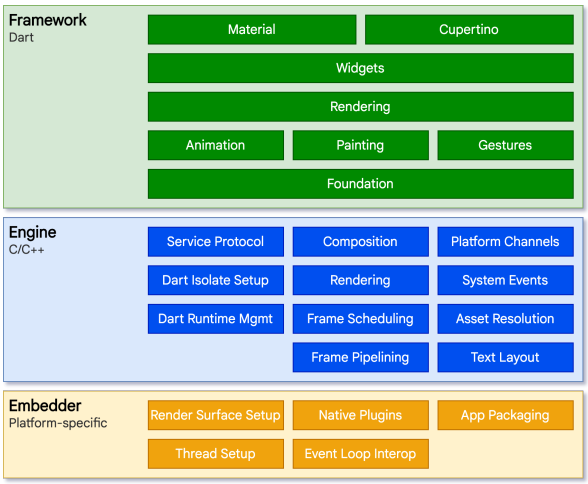


Abbildung 2.1: Architekturdiagramm von Flutter [Goo21a]

Die unterste `Embedder`-Schicht ist dabei für jede der unterstützten Plattformen einzeln implementiert und stellt den oberen Schichten Ressourcen zur Verfügung, wie beispielsweise ein Zeichnungsbereich, indem die Benutzeroberfläche gerendert werden kann. Zudem können hier native Erweiterungen eingebunden werden, die es der App später ermöglicht, von Flutter / Dart aus auf native Betriebssystemfunktionen wie die Kamera zuzugreifen.

Die mittlere `Engine`-Schicht beinhaltet die meisten Performance-kritischen Komponenten und stellt eine universelle Plattform für das Framework zur Verfügung so beinhaltet sie beispielsweise die Rendering-Engine. Anders als bei anderen Cross-Plattform-Frameworks wird in Flutter die komplette Benutzeroberfläche nicht mit nativen Design-Elementen dargestellt, sondern mit dieser eigenen Rendering-Engine gezeichnet. Dies bietet dabei den Vorteil, dass Design-Konzeptionen auf allen Plattformen gleich aussehen.

Die obere `Framework`-Schicht setzt auf den beiden unteren Schichten auf und bietet die Schnittstellen, die zur Entwicklung von Apps benötigt werden. Zudem beinhaltet sie einen umfangreichen Katalog an Standard-Widgets, die sich am Design der beiden Betriebssysteme iOS und Android orientieren. Diese Widgets sind in den Cupertino-Katalog für das iOS-ähnliche Design und in den Material-Katalog für das besonders auf Android oft genutzte Material-Design eingeordnet.

Das Konzept des Widgets, welches bereits öfters bisher erwähnt worden ist, wird in dem folgenden Kapitel näher beleuchtet.

2.1.3 *Widgets*

Eines der wichtigsten Bestandteile des Flutter-Frameworks sind die sogenannten Widgets. Der Satz “Everything is a widget” [Win20, Kap.1.9] wird in der Literatur oft verwendet, und bringt zum Ausdruck, dass Flutter das Konzept des Widgets für viele Anwendungsfälle nutzt. So kann ein Widget diverser Aufgaben übernehmen wie beispielsweise das Rendern einer UI-Komponente, Animationen oder das Anordnen von anderen Widgets. Zur Darstellung der Benutzeroberfläche benutzen also Widgets Kompositionen von diversen Widgets. So lassen sich beispielsweise mit einer `Row`, wie im Listing 2.1 zu sehen ist, mehrere Widgets nebeneinander anzeigen.

Listing 2.2: Vereinfachte Darstellung eines Widgets als Methode [Win20]

```
UI Widget(state)
```

Ein wichtiger Unterschied zu klassischen deklarativen UI-Frameworks ist, dass Widgets nur die Anleitung zum Bauen einer Benutzeroberfläche beinhalten, jedoch nicht den aktuellen Zustand des Widgets. Im Detail bedeutet

dies, dass die `build()`-Methode eines Widgets keinerlei Nebeneffekte haben soll und daher eine schnelle Ausführungszeit zu erwarten ist. Bildlich lässt sich ein Widget als Funktion darstellen, welche den aktuellen Zustand (engl. State) erhält und daraus die entsprechende Benutzeroberfläche generiert wie in Listing 2.2 vereinfacht dargestellt. Ein Widget erhält die darzustellenden Daten und generiert daraus die entsprechende Benutzeroberfläche.

In Flutter wird bei Widgets grundsätzlich zwischen `StatelessWidget` und `StatefulWidget` unterschieden.

Ein `StatelessWidget` hat grundsätzlich keinen veränderbaren Zustand. Dies heißt, dass alle Klassenvariablen unveränderlich sein sollen. In Dart wird dies mit dem Modifikator `final` gekennzeichnet. Daraus wird impliziert, dass alle Informationen zum Zustand des Widgets aus den im Konstruktor übergebenen werten stammen müssen. Somit wird das Widget nur neu gebaut, wenn sich Änderungen an den Werten des Konstruktors durch Änderungen in der Hierarchie darüber liegenden Widgets ergeben.

Ein `StatefulWidget` auf der anderen Hand besteht aus zwei Klassen. Zum einen dem `StatefulWidget` zum anderen dem `State` dieses Widgets. Diese Widgets vereinen die Möglichkeiten eines `StatelessWidget` mit der Möglichkeit, den Zustand des Widgets selbstständig zu verändern. Dies wäre beispielsweise bei einem Zähler relevant, wenn der Zähler erhöhrt werden soll. Zustandsänderungen werden dabei über die, wie in Listing 2.3 gezeigte, `setState` Methode vorgenommen, damit auch die Änderung an das Framework kommuniziert wird. Intern wird dabei das Widget als `dirty` markiert, welches dazu führt, dass die Build-Methode des States erneut aufgerufen wird durch das Framework. Entscheidend ist dabei, dass der State persistent bleibt, auch wenn sich beispielsweise Konstruktor-Parameter der `StatefulWidget`-Klasse ändern.

Listing 2.3: StatefulWidget

```
class Counter extends StatefulWidget {
    const Counter({Key? key}) : super(key: key);

    @override
    State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
    int _counter = 0;

    void _incrementCounter() {
        setState(() {
            _counter++;
        });
    }
```

```
    @override
    Widget build(BuildContext context) {
        return Column(
            children: [
                Text("Count: $_counter"),
                IconButton(icon: Icon(Icons.add),
                  onPressed: _incrementCounter),
            ],
        );
    }
}
```

Durch die Kombination von diversen Widgets entsteht so ein Widget-Baum, wie in Abbildung 2.2 zu sehen ist. Dieser Baum lässt sich auch mittels Hilfswerkzeugen traversieren, ist aber von der Grundstruktur für einen unidirektionalen Datenfluss ausgelegt. Dies bedeutete, dass Widgets nur untergeordnete Widgets durch Änderung derer Zustände verändern können sollen. Übergeordnete Widgets können somit - jedenfalls nicht direkt - verändert werden.

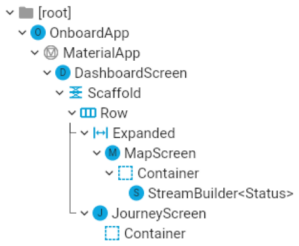


Abbildung 2.2: Flutter Widget Tree

Anders als bei der nativen Entwicklung wird hier also die Benutzeroberfläche deklarativ programmiert. Bei der Entwicklung wird also festgelegt, wie welches Element bei welchem Zustand auszusehen hat, ohne dabei den Kontrollfluss beschreiben zu müssen.

Zusammenfassend wird festgehalten, dass der Zustand und die Verwaltung des Zustands von Widgets einen wichtigen Teil im Lebenszyklus einer Flutter-Anwendung ist. Dies gibt schon einen Vorgriff darauf, dass hier eine Lösung gefunden werden muss, welche auf diverse Szenarien anwendbar sein muss.

2.2 AUTOMATISIERTES TESTEN

Um Flutter-Anwendungen automatisiert testen zu können, bestehen in Flutter die drei Testkategorien “unit testing, widget testing [...][und] integration testing” [AA21, S. 21].

Unit-Tests werden zum automatisierten Testen einzelner Funktionen oder Klassen eingesetzt. Tests werden dabei als Lambda-Funktion konstruiert. Mit mitgelieferten Werkzeugen, lassen sich Werte darauf überprüfen, ob sie den erwarteten Wert entsprechen.

Widget-Tests testen einzelne oder mehrere Widgets darauf, ob sie dem gewünschten Verhalten entsprechen. Die Widget-Tests können dabei ohne die Verwendung eines iOS- oder Android-Simulator innerhalb von Flutter getestet werden. [Goo21e] Dafür existieren diverse Werkzeuge in der Flutter-Testing-Bibliothek, womit sich Widgets erstellt werden können oder bestimmte Eigenschaften von Widgets überprüft werden können.

Integration-Tests testen die Anwendung als ganzes und prüfen somit, ob die jeweiligen Komponenten auch korrekt untereinander funktionieren. [AA21, S. 21] Diese Art von Tests sind allerdings für die Evaluation nicht relevant und werden daher nicht näher behandelt.

2.3 ZUSTANDSVERWALTUNG

Nachdem nun die Grundlagen des Flutter-Frameworks und die Details der Zustandsverwaltung der Widgets im letzten Kapitel erläutert wurden, kann jetzt die Zustandsverwaltung im Detail betrachtet werden.

Windmill fasst den Komplex der Zustandsverwaltung in Flutter in seinem Standardwerk Flutter in Action wie folgt zusammen:

"State management is a combination of passing data around the app, but also re-rendering pieces of your app at the right time. All the re-rendering in Flutter is dependent on the State object and its lifecycle." [Win20, Kap.8.1.2]

Daraus ergibt sich, dass die State-Klasse in der Zustandsverwaltung von Flutter eine wichtige Rolle spielt, und alle Ansätze dieses Konzept benutzen müssen, um in das Flutter-Framework integrierbar zu sein. Windmill beschreibt dabei die Aufgabe der Zustandsverwaltung eher auf einer Ebene des Datenflusses und des Ablaufs der Neu-Erstellung der Benutzeroberfläche.

Arshad sieht die Zustandsverwaltung dabei eher auf einer eher ablaufszentrierten Sichtweise indem er die Aufgabe der Zustandsverwaltung wie folgt analysiert:

"State management is simply a technique, or multiple techniques, used to take care of the changes that occur in your application." [Ars21, Kap.1]

Als Beispiele nennt er dabei, das Reagieren auf Interaktionen mit der Anwendung oder die Beibehaltung des Datenflusses über mehrere Screens hinweg.

Beide Sichtweisen haben gemein, dass die Grundaufgabe der Zustandsverwaltung die Sicherstellung eines korrekten Zustands der Anwendung, einzelner Screens oder einzelner Widgets sein muss, sowie die mögliche Überführung dieses Zustands in einen neuen Zustand als Reaktion auf Veränderungen.

Nachdem nun eingeführt wurde, was unter einer Zustandsverwaltung in Flutter zu verstehen ist, werden nun mögliche bestehende Ansätze für eine Zustandsverwaltung skizziert, um im weiteren Verlauf der Ausarbeitung im Analyse- und Evaluationskapitel diese eingehender zu untersuchen.

2.3.1 *Auswahl*

Zur Auswahl der zu evaluierenden Lösungsansätze wird die Aufzählung von Zustandsverwaltungs-Ansätzen aus der Flutter-Dokumentation [Goo21d] als Grundlage verwendet. In dieser Aufzählung werden die Zustandsverwaltungssysteme Provider, Riverpod, setState, `InheritedWidget` & -Model, Redux, Fish-Redux, BLoC / RX, GetIt, MobX, Flutter Commands, Binder, GetX, states_rebuilder und Triple Pattern genannt.

Da eine Evaluation aller aufgezählten Zustandsverwaltungssysteme, den Umfang dieser Ausarbeitung überschreiten würde, wurde die Auswahl eingeschränkt.

Die Ansätze setState und `InheritedWidget` werden in die Evaluation mit aufgenommen, da sie zu der Grundausstattung der Flutter-Standardbibliothek gehören, und somit einen Basiswert für Zustandsverwaltungssysteme bilden und somit relevant sind. Das BLoC-Pattern wird aufgenommen, da es aufgrund der großen Verbreitung in der Literatur relevant ist. Die Bibliothek Provider wird ebenfalls aufgenommen, da es sich laut der Dokumentation um den empfohlenen Ansatz für Zustandsverwaltung in Flutter handelt, und die Bibliothek mit 6132 Like-Angaben [Goo22b] zu einer der beliebtesten Flutter-Pakete [Goo] auf der Plattform handelt. Riverpod wird ebenfalls aufgenommen, da dies den Ansatz von Provider weiterentwickelt und somit geprüft werden kann, ob diese Bibliothek tatsächlich besser abschneidet als die Original-Bibliothek. Zuletzt werden die Bibliotheken MobX und Redux aufgenommen, da sie besonders aufgrund ihrer Herkunft aus dem React-Ökosystem eine besondere Relevanz für die Evaluation haben, um ebenfalls feststellen zu können, ob bereits in React verbreitete Ansätze auch in Flutter sinnvoll einsetzbar sind.

2.3.2 *Mitgelieferte Werkzeuge*

Die erste Kategorie der Zustandsverwaltungssysteme umfasst jene, welche ohne eine zusätzliche Bibliothek auskommen und somit de facto im Flutter Framework mitgeliefert werden. Zunächst werden einfacheren Konzepten und Werkzeugen vorgestellt, anschließend folgen die komplexeren Konzepte und Werkzeuge vorgestellt.

2.3.2.1 *setState*

Die wohl grundlegendste Möglichkeit, den Zustand in einer Flutter Anwendung zu verwalten stellt das ausschließliche Benutzen der `setState`-Methode dar. Ein Beispiel zur Verwendung wurde bereits in Listing 2.3 in der `incrementCounter`-Methode eingeführt. Hier findet die Speicherung des Zustands also durch die direkte Manipulation des States von `StatefulWidgets` statt.

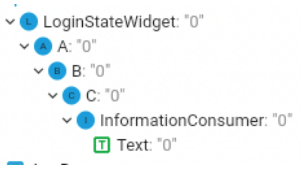


Abbildung 2.3: Flutter Widget Tree bei setState

Wie vorausgehend beschrieben, muss ein Zustandsverwaltungssystem aber nicht nur den Zustand einzelner Widgets verwalten können, sondern auch von größeren Ordnungen wie beispielsweise von Screens oder der ganzen Anwendung. Um dies bei diesem Ansatz erreichen zu können, wir der Zustand oder Teile des Zustands über die Konstruktor innerhalb des Widget-Trees weiter nach unten gegeben.

Anschaulich lässt sich dies durch das Beispiel Abbildung 2.3 darstellen, welches eine Anwendung, die global speichern muss, welche Person aktuell angemeldet ist, zeigt. Da diese Information in diesem Beispiel an diversen Stellen innerhalb der Anwendung benötigt wird, ergibt es Sinn, diese Information weit oben im Baum in Form eines `StatefulWidget` namens `LoginStateWidget` zu speichern, da der Datenfluss innerhalb des Baums ausschließlich unidirektional von oben nach unten stattfindet. Um diese Information nun an die Widgets zu kommunizieren, die es benötigen - in diesem Fall `InformationConsumer` - muss `LoginStateWidget` die Information per Konstruktor an das nachgelagerte Widget weitergeben. Diese nachgelagerten Widgets (`A, B, C`) müssen dies ebenfalls tun, bis die Information am Ziel `InformationConsumer` angekommen ist. Dieses Anwendungsmuster wird in der Literatur als “lifting state up” [Win20, Kap.8.2] bezeichnet.

2.3.2.2 *InheritedWidget*

Neben der Möglichkeit, den Zustand über den Widget-Baum nach unten weiter zu propagieren, bietet Flutter noch das Konzept `InheritedWidget` an. Diese Widgets bilden eine eigene Widget-Gruppe und sind weder den `StatefulWidgets` noch den `StatelessWidgets` zuzuordnen. [Win20, Kap.8.2.1] `InheritedWidgets` ermöglichen es nachgeordneten Widgets, auf den Zustand des Widgets direkt zuzugreifen. Hier muss allerdings beachtet werden, dass das `InheritedWidget` immer unveränderlich ist. Dies bedeutet, dass andere Widgets über die Veränderung von Konstruktor-Parametern neue Instanzen des Widgets erstellen müssen, um eine Zustandsänderung zu bewirken. Daher lassen sich diese Widgets oft in Kombination mit `StatefulWidgets`, welche für die Manipulation des Zustands zuständig sind, vorfinden.

Im Beispiel Listing 2.4 kann man erkennen, dass das Widget lediglich die Daten lagert, welche zur Verfügung gestellt werden sollen. In diesem Fall ist dies der aktuelle Benutzende `currentUser`. Diese*r kann nicht vom `InheritedWidget` selbst geändert werden, sondern hängt von der Eingabe im Konstruktor ab. Das `InheritedWidget` wird dabei gemäß dem Motto "Everything is a widget" in den Widget-Baum integriert. Der Konstruktor-Parameter `child` gibt dabei das im Baum untergeordnete Widget an. Über die `updateShouldNotify`-Methode wird dem Framework kommuniziert, ob sich der Zustand im Vergleich zum vorherigen geändert hat, und somit ein Neubauen der Widgets, die dieses `InheritedWidget` referenzieren, notwendig ist.

Listing 2.4: Aufbau eines InheritedWidget

```
import 'package:flutter/widgets.dart';

class UserStore extends InheritedWidget {
  const UserStore({
    Key? key,
    required this.currentUser,
    required Widget child,
  }) : super(key: key, child: child);

  final User currentUser;

  static UserStore? of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<UserStore>();
  }

  @override
  bool updateShouldNotify(UserStore old) {
    return currentUser != old.currentUser;
  }
}
```

Anders als bei dem `setState`-Konzept kann hier direkt der Zustand durch andere nachgelagerte Widgets referenziert werden. Dafür werden oft Hilfsmethoden wie in diesem Fall die `of`-Methode verwendet. Diese greifen auf Werkzeuge des Frameworks zu, die das `InheritedWidget` des angebenden Typen zurückgibt und sicherstellt, dass bei einer Veränderung des das referenzierende Widget auch neu gebaut wird und so die Änderung beachtet wird.

Dieser Mechanismus wird auch von diversen anderen Zustandsverwaltungssystemen verwendet.

Als Erweiterung des `InheritedWidget` kann man `InheritedModel` sehen. Dabei ist die Funktionsweise äquivalent mit einer Besonderheit. Es besteht hier die Möglichkeit, Zugriffe und Änderungen nach sogenannten `aspects` zu kategorisieren. Somit kann bei komplexeren Zuständen, es ermöglicht werden, dass die Widgets nur dann neu gebaut werden, wenn der betreffende aspect sich ändert. [Goo21c]

2.3.3 *Business Logic Components (BLoC)*

Das 2018 auf der Entwicklerkonferenz DartConf vorgestellte BLoC-Pattern ist im Vergleich zu den bisher vorgestellten Ansätzen ein Design-Pattern zur Verwaltung von Zuständen und nicht nur ein Werkzeug des Frameworks. Das Ziel von BLoC ist es, die komplette Logik von der Benutzeroberfläche zu trennen. [Fau20, S. 17] Die Logik und der Zustand wird dabei in den namensgebenden Business Logic Components (BLoC) verwaltet. Die Komponenten haben die Aufgabe, Zustands-Ereignisse von Widgets zu empfange und Widgets zu aktualisieren, wenn sich der Zustand ändert. Diese Komponenten unterliegen grundlegenden, nicht-verhandelbaren Regeln, welche im Vortrag von Soares definiert worden sind:

“

1. Inputs and outputs are simple Streams/Sink only
2. Dependencies must be injectable and platform agnostic
3. No platform branching allowed [...]

” [Soa18, 24:04]

Die erste Regel bedeutet, dass BLoC weder Methoden noch Variablen nach außen freigeben dürfen, sondern nur über `Streams` und `Sinks` mit Widgets kommunizieren. Ein `Stream` ist dabei in Flutter ein asynchroner Fluss von

Daten oder Ereignissen. Widgets können diesen Ereignisfluss abonnieren und werden dann aktualisiert, wenn sich dieser ändert. Ein `Sink` ist intern auch eine Art von Stream, welcher aber die Besonderheit hat, dass man von außen neue Ereignisse hinzufügen kann. Über diesen `Sink` lassen sich also Daten und Ereignisse an das BLoC übergeben.

Die zweite Regel sagt aus, dass BLoC keine Abhängigkeiten zur Benutzeroberfläche haben dürfen. Selbst das Importieren von Flutter-Bibliotheken in diese Dateien ist verboten. Damit wird erreicht, dass BLoC komplett plattformunabhängig sind, und somit die komplette Benutzeroberfläche theoretisch ersetzt werden könnte, ohne die Logik ändern zu müssen.

Die dritte Regel legt fest, dass innerhalb von BLoCs keine Unterscheidungen zwischen Betriebssystemen oder Plattformen vorgenommen werden darf.

Der innere Aufbau der BLoC ist explizit nicht vorgeschrieben, dient aber dazu die über die `Sinks` eingehenden Daten und Ereignisse zu verarbeiten und anschließend den neuen Zustand über die `Streams` zurück an die Widgets zu propagieren. Technisch kommen hier oft Techniken und Werkzeuge aus der reaktiven Programmierung wie `RxDart` zum Einsatz.

Jede Seite (engl. Screen) sollte dabei exakt einem BLoC zugeordnet sein. Damit die Widgets auf diesen BLoC zugreifen können, müssen diese `injectable` sein - also zwischen mehreren Widgets geteilt. Dies kann unter anderem mit den bereits in Unterabschnitt 2.3.2 vorgestellten Ansätzen umgesetzt werden.

2.3.4 *Provider*

Provider ist eine Bibliothek, die auf dem `InheritedWidget`-Ansatz (siehe Unterunterabschnitt 2.3.2.2) beruht. Dabei vereinfacht die Bibliothek die Benutzung und Erstellung von Klassen, die Zustände verwalten, und kombiniert das `InheritedWidget`-Konzept unter anderem mit der `ChangeNotifer`-Klasse. [Sle20]

Die abstrakte Klasse `ChangeNotifier` bietet die Möglichkeit Event-Listener für die erbende Klasse zu registrieren und bei Veränderung über den Aufruf einer entsprechenden Methode diese zu benachrichtigen, wie in Listing 2.5 in der Methode `updateUser` gezeigt wird. [Goo21b] Diese Kombinationsmöglichkeit wird auch für andere Konzepte und Klassen angeboten. So lassen sich Provider mit `Listnables`, also Objekte, die Events emittieren können, `ValueListanble`, also `Listnables`, die nur über Änderungen einer Variable benachrichtigen, `Streams`, welche bereits in Unterabschnitt 2.3.3 vorgestellt worden sind oder `Futures` kombinieren.

Listing 2.5: Aufbau einer ChangeNotifier-Klasse für Provider

```
class UserStore extends ChangeNotifier {
  var user;
  UserStore(this.user);

  void updateUser(newUser) {
    this.user = newUser;
    notifyListeners();
  }
}
```

Es kann festgehalten werden, dass Provider im ersten Schritt eine Form von *Dependency Injection* zur Verfügung stellt und im zweiten Schritt mit einer Klasse verknüpft wird, welche über Änderungen informiert, und somit zu einer Zustandsverwaltungslösung ausgebaut werden kann.

Listing 2.6: Injektion und Abruf von Provider-Klassen

```
class ProvidingWidget extends StatelessWidget {
  final user;
  const ProvidingWidget({Key? key, this.user}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(create: (context) => UserStore(user),
        child: ConsumingWidget());
  }
}

class ConsumingWidget extends StatelessWidget {
  const ConsumingWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final user = Provider.of<UserStore>(context).user;
    return Text(user.firstName);
  }
}
```

Um eine `ChangeNotifier` als Zustandsverwaltungslösung mit Provider zu nutzen, ist es erforderlich, die jeweiligen Zustandsklassen (oft Store genannt) in den Widget-Baum zu integrieren. Dies erfolgt wie in der Klasse `ProvidingWidget` in Listing 2.6 gezeigt, durch das Erstellen eines Widgets, welche keine eigene Benutzeroberfläche darstellen, sondern nur Widgets "nach unten" durchreichen. Nun können die Daten aus der Zustandsklasse von allen Widgets abgerufen werden, die sich im Widget-Baum an einen der nachgelagerten Äste der injizierenden Widgets befindet.

Die nachgelagerten Widgets können dabei, die Zustandsklasse durch eine einfachen, generischen Methodenaufruf der Provider-Bibliothek abrufen. Durch die Verwendung von `InheritedWidget` innerhalb der Bibliothek wird damit sichergestellt, dass wenn sich Daten in den Zustandsklassen ändern, referenzierende Widgets über diese Änderung informiert werden und gegebenenfalls neu gebaut werden.

2.3.5 *Riverpod*

Riverpod baut auf dem bereits vorgestellten Provider-Konzept auf, ergänzt es aber mit weiteren Funktionalitäten. Ein großer Unterschied liegt auch darin, dass hier Provider nicht in den Widget-Baum integriert werden müssen. Zudem ist es hier anders als bei der Provider-Bibliothek möglich, mehrere Provider vom gleichen Klassentyp zu unterscheiden. Hier wird nämlich nicht der Klassentyp zum Abruf der Zustandsklasse im Widget verwendet, sondern eine Variable. Wie diese im Widget abgerufen wird, ist dabei von den Entwickler*innen zu entscheiden. Denkbar sind hier beispielsweise das Verwenden einer globalen Variable oder einer Dependency-Injection-Lösung wie `get_it`. Damit wird auch ermöglicht, dass Riverpod im Vergleich zu Provider keine Abhängigkeit zum Flutter-Framework hat und somit eine reine Dart-Bibliothek ist. [Gre+21, S. 8]

Durch diese Unabhängigkeit vom Widget-Tree und dem Flutter-Framework kann hier auf das sogenannte *Nesting* verzichtet werden. Dieses wird bei Provider benötigt, um Provider zu initialisieren, die von anderen Providern abhängen. Riverpod nutzt hier stattdessen eine Methode, mit der bei der Erstellung eines Providers andere Provider gelesen werden können.

Listing 2.7: Zustandsklasse in Riverpod

```
class UserStore extends StateNotifier<User> {
  UserStore(User initialState) : super(initialState);

  changeFirstName(String newName) {
    state = User(newName);
  }
}
```

Für die Zustandsklassen selber gibt Riverpod eine eigene Struktur vor. Die Zustandsklassen erben die generische Klasse `StateNotifier`. Dabei wird über das Generic festgelegt, welchen Datentyp der Zustand haben soll. Dieser Zustand wird dann der Klasse als vererbte Variable zur Verfügung gestellt. Bei Änderung des Zustands reicht es somit aus, den Inhalt dieser Variable neu zu setzen wie in Listing 2.7 gezeigt wird. Die Benutzeroberfläche greift dabei ausschließlich auf die Zustands-Variable direkt zu.

Die Zustandsklasse ist nur dafür zuständig, Methoden zur Änderung dieses Zustands zur Verfügung zu stellen.

Riverpod akzeptiert neben diesen Zustandsklassen auch die im Provider-Kapitel (siehe Unterabschnitt 2.3.4) vorgestellen Event-Emitter wie beispielsweise `ChangeNotifier`.

Da wie bereits erwähnt, Riverpod keine Flutter-Bibliothek ist und somit nicht wie Provider `InheritedWidget` zum Aktualisieren von referenzierenden Widgets benutzt, unterstützt sie von Haus aus noch nicht die Nutzung innerhalb eines Widgets. Um dies jedoch zu ermöglichen, besteht die Wahl zwischen zwei Ansätzen, die sich nur semantisch unterscheiden.

Listing 2.8: ConsumerWidget

```
final userStoreProvider = StateNotifierProvider((ref) => UserStore());

class ConsumingWidget extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final userStore = ref.watch(userStoreProvider);
    return Text(userStore?.state.firstName);
  }
}
```

Bei `flutter_riverpod` werden die bereits eingeführten Widget-Typen `StatefulWidget` und `StatelessWidget` durch neue Typen ergänzt beziehungsweise ersetzt. Einer dieser neuen Typen ist `ConsumerWidget`. Durch das Erben von dieser Klasse, wird der `build`-Methode des Widgets eine neue Variable ergänzt, über die der Zustand von Riverpod-Providern abgefragt werden kann, wie in Listing 2.8 zu sehen ist. Die hier verwendete `watch`-Methode bewirkt, dass das Widget neu gebaut wird, wenn sich der betreffende Provider ändert.

2.3.6 *Redux*

Redux ist ein Zustandsverwaltungssystem, welches ursprünglich für React entwickelt worden ist. React ist eine JavaScript-Bibliothek zum Bauen von Benutzeroberflächen. Da React und Flutter ähnliche Techniken wie Widgets oder das State-Prinzip verwenden, lässt sich dieser Ansatz auf Flutter und Dart übertragen.

Redux basiert auf drei grundlegenden Prinzipien:

> Single source of truth [...]
>
> State is read-only [...]
>
> Changes are made with pure functions [...] [GF18, Kap.1.3.2]

Das Prinzip der "Single source of truth" [GF18, Kap.1.3.2] wird dadurch umgesetzt, dass es für die komplette Anwendung nur einen zusammengefassten Zustand gibt. Es findet hier also keine Aufteilung in diverse Unterzustände beispielsweise für einzelne Seiten statt.

Abbildung 2.4: Datenfluss in Redux [GF18, Kap.1.3.3]

Dieser globale Zustand ist, wie im zweiten Prinzip dargestellt wird, unveränderlich. Um den Zustand zu ändern beziehungsweise in diesem Fall zu ersetzen, ist es erforderlich, sogenannte *actions* zu emittieren. *Actions* sind dabei in Flutter einfache Objekte, die auch Parameter in Form von Variablen beinhalten können. Diese *Action* wird dann wie in Abbildung 2.4 anschaulich zu sehen ist, über einen *Reducer* auf den globalen Zustand (engl. *store*) angewendet.

Damit wird dann auch das dritte Prinzip umgesetzt. Ein *Reducer* stellt eine einfache Funktion dar, welche als Parameter den aktuellen Zustand und die *Action* erhält und als Rückgabewert einen neuen Zustand angibt, welcher den globalen Zustand ersetzt.

Für Dart wurde dies mit der `redux` Bibliothek umgesetzt. Ähnlich wie bei Riverpod wird hier noch eine zusätzliche Bibliothek benötigt, die Redux für das Flutter-Widget-System anwendbar macht. Dafür wird `flutter_redux` verwendet. Diese adaptiert das Prinzip der Provider-Bibliothek für Redux-Stores. So wird der Store mittels eines Provider-Widgets über den Widget-Tree für nachgelagerte Widgets zur Verfügung gestellt. Auf den Store zugegriffen wird über Selektoren, welche den benötigten Teil des globalen Zustands eingrenzen und per Callback-Funktion zurückgeben. Somit wird über diese Selektoren auch die Veränderungen der Widgets ausgelöst, wenn sich der entsprechende Teil des globalen Zustands ändert.

2.3.7 *MobX*

MobX ist wie Redux ebenfalls ein Ansatz, der ursprünglich aus dem React-Ökosystem stammt. MobX stellt sich dabei die Aufgabe, es zu vereinfachen automatisch diverse Informationen aus dem Anwendungs-Zustand abzuleiten. Um dies zu erreichen, gibt es bei MobX fünf grundlegende Konzepte:

"

- Observables [...]
- Computed Values [...]
- Reactions [...]
- Actions [...]

" [Mez18, S. 130]

Observables bilden die Grundlage eines Zustands. Ein Observable stellt einen Wert dar, welcher beobachtet werden kann und somit auf Änderungen reagiert werden kann. Von Observables werden alle anderen Informationen abgeleitet.

Mit Computed Values lassen sich ein oder mehrere Observables kombinieren oder einer weiteren Verarbeitung unterwerfen. Wenn sich eines der zugrundeliegenden Observables ändert, wird auch dieser Wert neu berechnet.

Reactions reagieren auf Änderungen von Observables oder Computed Values und lösen Seiteneffekte aus. Ein Beispiel hierfür ist das Observer-Widget, welches es ermöglicht, Widgets neu zu bauen, wenn sich die referenzierten Observables ändern.

Actions werden wie bei Redux hier ebenfalls verwendet, um Änderungen am Zustand - hier also Observables - durchzuführen. Allerdings wird hier als Action eine Funktion verstanden, welche Observables abändert und nicht ein Objekt, welche mithilfe eines Reducers den Zustand mutiert.

Observables, Computed Values und Actions werden dabei oft in Klassen zu einem Store zusammengefasst. Die Instanzen dieser Stores müssen dabei über einen Service Locator ähnlich wie bei Riverpod in die Widgets injiziert werden, damit diese den Store mithilfe von Observern nutzen können.

3

ANALYSE

Nachdem nun die diversen existierenden Ansätze für die Zustandsverwaltung in Flutter vorgestellt worden sind, muss im nächsten Schritt untersucht werden, welche Anforderungen an solche Systeme gestellt werden. Zudem soll daraus abgeleitet werden, welche Kriterien für die spätere Evaluation entscheidend sind.

In "State management approaches in Flutter"[Sle20] wurden bereits Bewertungskriterien für eine Evaluation von Zustandsverwaltungssystemen aufgestellt, allerdings kann die Herleitung der Bewertungskriterien nicht nachvollzogen werden, da die verwendeten Quellen nicht die zitierten Bewertungskriterien enthalten. Daher werden nun neue Bewertungskriterien definiert.

3.1 ANFORDERUNGSANALYSE

Die grundlegende Anforderung für Zustandsverwaltungssysteme ist es, dass sie den Zustand einer Anwendung und bestimmter Teile einer Anwendung wie eines Widgets oder einer ganzen Seite verwalten können. Alle bereits vorgestellten Lösungsansätze können dies mit mehr oder weniger Aufwand auch erfüllen. Allerdings ist es neben der Aussage, ob ein System diese Anforderung umsetzen kann, auch wichtig wie und in welcher Qualität diese umgesetzt werden.

3.1.1 *Allgemeine Anforderungen*

Die Wahl eines Zustandsverwaltungssystems bestimmt die Architektur einer Anwendung signifikant mit. Daher sind die Anforderungen an eine gute Architektur oder ein gutes Software-Design auch in Teilen auf Zustandsverwaltungssysteme übertragbar. Um nun daraus Anforderungen zu konstruieren, ist es also erforderlich, sich anzuschauen, was ein gutes Software-Design überhaupt ausmacht.

Zur Bewertung von Software-Qualität wurden diverse Anforderungen und Kriterien entwickelt. Diese wurden mit der ISO-Norm 9126 standardisiert.

Eines der Qualitätsmerkmale ist die Wartbarkeit von Software, die wie folgt, beschrieben wird:

> Fähigkeit des Softwareprodukts änderungsfähig zu sein. Änderungen können Korrekturen, Verbesserungen oder Anpassungen der Software an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen einschließen [Bal09, S. 470]

Daraus ergeben sich auch Anforderungen an die Quelltext-Qualität. Darunter wird beispielsweise die Anforderungen der "Änderbarkeit[...] [und] Testbarkeit" [Bal09, S. 470] verstanden. Diese Anforderungen können auch für Zustandsverwaltungssysteme übernommen werden. Zustandsverwaltungssysteme und die Anwendungen, die diese verwenden, müssen effizient veränderbar und erweiterbar sein. Zusätzlich ist eine wichtige Anforderung, dass die resultierende Anwendung auch automatisiert testbar sein muss, um zu prüfen, ob sie den gewünschten Anforderungen entspricht.

Hinzukommend spielen auch noch weitere Anforderungen zur Sicherstellung von Software-Qualität eine Rolle wie in "Software quality metrics for object-oriented environments" beschrieben wird. So seien die Attribute der Effizienz, Komplexität, Verständlichkeit, Wiederverwendbarkeit und Testbarkeit/Wartbarkeit von Bedeutung. [vgl. RH97, S. 1] Diese Anforderungen lassen sich auf den beschriebenen Evaluationsfall übertragen, müssen allerdings noch auf diesen Anwendungsfall hin angewandt werden.

Angewendet auf Zustandsverwaltungssysteme könnte die Effizienz dadurch beschrieben werden, wie effizient das Aktualisieren der Widgets im Widget-Baum erfolgt. Beispielsweise könnte hier untersucht werden, ob nur die Widgets neu erstellt werden, die von einer Änderung wirklich betroffen sind.

Die Verständlichkeit kann angewendet so gedeutet werden, ob die durch die Verwendung des Systems neu geschaffenen Strukturen auch einfach zu verstehen - also verständlich sind.

Die Wiederverwendbarkeit lässt sich nicht gut auf Zustandsverwaltungssysteme übertragen, da die geschaffenen Strukturen immer auf den spezifischen Anwendungsfall ausgerichtet sind, und somit eine Wiederverwendbarkeit keine hohe Relevanz hat.

Die Testbarkeit/Wartbarkeit wurde bereits bei den Erläuterungen zum ISO-Standard berücksichtigt und lässt sich auch auf den Anwendungsfall wie beschrieben anwenden.

3.1.2 *Spezifische Anforderungen*

Neben diesen allgemeinen Anforderungen stellen sich noch spezifische Anforderungen an die Zustandsverwaltungssysteme.

Für die Integration und Einbindung der Systeme durch Entwickler*innen, ist es erforderlich, dass das System hinreichend dokumentiert ist, damit die Einbindung und Umsetzung erleichtert wird.

Gerade bei Anwendungen, die mit agilen Methoden entwickelt werden, wird oft iterativ vorgegangen und die Anwendung so schrittweise erweitert. Daraus ergibt sich die Anforderung, dass die Systeme mit dem Wachstum der Anwendung mithalten können - also skalierbar sind. Dies erweitert die bereits beschriebene Anforderung der Änderbarkeit.

Da wie bereits erwähnt, die Zustandsverwaltung ein integraler und Architekturbestimmender Teil der Anwendung ist, ist es bei der Entwicklung ebenfalls hilfreich, wenn durch dieses System bereits eine gewisse Struktur vorgegeben wird. Dies hilft dabei, auch bei größeren Projekte einheitliche Vorgehensweisen durchzusetzen und somit einen homogenen Quelltextstil zu forcieren.

3.1.3 *Zusammenfassung*

Wenn man nun die allgemeinen und spezifischen Anforderungen zusammenfasst, erhält man folgenden Anforderungskatalog:

- Änderbarkeit / Skalierbarkeit
- Testbarkeit
- Effizienz
- Komplexität / Wartbarkeit
- Verständlichkeit
- Dokumentierung
- Strukturbestimmung

3.2 BEWERTUNGSKRITERIEN

Da nun die Anforderungen an Zustandsverwaltungssysteme feststehen, können jetzt Kriterien für die Bewertung der Systeme beziehungsweise der Umsetzung mit diesen Systemen definiert werden. Ziel dabei ist es, möglichst aussagekräftige, objektive Bewertungskriterien zu definieren. Die Kriterien sollen auf die jeweiligen Anforderungen einzahlen.

3.2.1 *Änderbarkeit/Skalierbarkeit*

Um zu untersuchen, ob ein Zustandsverwaltungssystem skalierbar oder nicht aufwendig änderbar ist, muss untersucht werden, wie dieses mit einer wachsenden Größe des Zustands umgeht. Dabei ist ein entscheidendes Kriterium, wie effektiv sich diverse Zustände untereinander verknüpfen lassen. Es ist nämlich damit zu rechnen, dass bei einer größer werdenden Anwendung auch die Koppelung zwischen den einzelnen Zuständen zunimmt.

Für dieses Bewertungskriterium wird eine qualitative Bewertungsskala verwendet, welche die Änderbarkeit/Skalierbarkeit mit den Werten “nicht erfüllt”, “teilweise erfüllt” und “vollständig erfüllt” bewertet.

3.2.2 *Testbarkeit*

Für die Testbarkeit sind diverse Blickpunkte von Relevanz. Zuerst sollte geprüft werden, inwiefern sich die mit dem Zustandsverwaltungssystem abgebildete Geschäftslogik testen lässt. Hier kann beispielsweise geprüft werden, ob es ohne Veränderung am Ausgangsquelltext möglich ist, einzelne Funktionen zu testen, die den Zustand verändern. Grundlage dafür sollten automatisierte Unit-Tests sein.

Ein anderer Blickpunkt ist die Testbarkeit von Widgets, die auf geteilte Zustände zugreifen. Hier sollte geprüft werden, ob die geteilten Zustände zum Test von Widgets einfach durch Platzhalter (engl. mocks) ausgetauscht werden können.

Ausgehend von der Prüfung dieser Eigenschaften, wird eine Bewertung mit den Werten “nicht erfüllt”, “teilweise erfüllt” und “vollständig erfüllt” vorgenommen.

3.2.3 *Effizienz*

Die Effizienz eines Zustandsverwaltungssystemes lässt sich auch dadurch messen, wie effizient es das Neubauen von Widgets nach Veränderungen orchestriert. So sollte ein Widget nur dann neu gebaut werden, wenn dies durch eine das Widget betreffende Änderung des Zustands es verlangt. Andere Betrachtungen der Performance sind nur schwer zu messen, da sich andere Variablen wie die Architektur oder Verwendung innerhalb der Anwendung einen großen Einfluss darauf haben.

Geprüft werden soll dieses Verhalten an x verschiedenen Prüfstellen in der Anwendung, indem an diesen stellen Zähler eingesetzt werden, die zählen, wann das Widget neu gebaut wird. Dabei wird eine für alle Zustandsverwaltungssysteme gleiche automatische Teststrecke erstellt, welche sicherstellen soll, dass immer die exakt gleichen Bedienhandlungen vorgenommen werden. Damit lässt sich dann abschätzen, wie effizient die Zustandsverwaltungssysteme die Zustandsänderungen an die Widgets propagieren. Erstrebenswert hierbei ist es, ein möglichst geringen Wert bei den Zählern zu erreichen, wobei sichergestellt werden muss, dass dabei alle Systeme die Funktionalität richtig implementieren.

3.2.4 *Komplexität / Wartbarkeit*

Zur Messung der Komplexität ist es auch möglich, die Komplexität eine Anwendung quantitativ zu messen. Für diesen Anwendungsfall werden in der Literatur diverse Verfahren und Metriken beschrieben. Die Metrik Maintainability Index (MI) kombiniert dabei diverse Metriken und gewichtet sie. Im Detail wird wie in Gleichung 3.1 dargestellt, das durchschnittliche Halstead-Volumen ($aveV$), die durchschnittliche zyklomatische Komplexität ($aveVG2$) und die durchschnittliche Anzahl der Quelltextzeilen ($aveLOC$) verwendet. [WOA97, S. 133] Aufbauend auf dieser Kombination von Metriken soll eine ein-wertige Metrik geschaffen werden, die zum Ausdruck bringt, wie wartbar eine Software ist. [WOA97, S. 129]

$$\begin{aligned} \text{Maintainability index} = 171 &- 5,2 * ln(aveV) \\ &- 0,23 * aveVG2 \\ &- 16,2 * ln(aveLOC) \end{aligned} \tag{3.1}$$

Um diese Metrik für die einzelnen Systeme zu bestimmen, wird das Werkzeug *Dart Code Metrics* eingesetzt, welches diverse Metriken für Dart-Quell-

text bestimmen kann. Der von dem Werkzeug berechnete Wert wird leicht abweichend, wie in Gleichung 3.2 zu sehen, von der Original-Formel auf eine Skala von 0 bis 100 abgebildet, wobei 100 den besten erzielbaren Wert darstellt. [Kru21]

$$\begin{aligned}\text{Maintainability index} = max(0, (171 \\ -5,2 * log(HALVOL) \\ -0,23 * log(CYCLO) \\ -16,2 * log(SLOC)) * 100/171)\end{aligned} \tag{3.2}$$

Diese Metrik wird als Bewertungsmaßstab verwendet.

3.2.5 *Verständlichkeit / Lesbarkeit*

Die Lesbarkeit und Verständlichkeit eines Quelltextes lassen sich nur schwer quantifizieren. Daher erfolgt hier eine begründete qualitative Bewertung anhand von mehreren Fragestellungen.

So ist beispielsweise für die Verständlichkeit dienlich, wenn bereits in Flutter eingeführte Konzepte verwendet werden, sodass Entwickler*innen, die noch keine Erfahrungen mit dem Zustandsverwaltungssystem haben, ihre bereits existierenden Kenntnisse anwenden können.

Eine weitere Eigenschaft von lesbaren und verständlichen Quelltext ist es, dass die Struktur nachvollziehbar und klar ist. Bei der Einschätzung muss berücksichtigt werden, dass der Einfluss des Zustandsverwaltungssystems auf diese Eigenschaft unterschiedlich stark ausgeprägt ist.

In Flutter werden Widget-Bäume wie in Unterabschnitt 2.1.3 gezeigt durch das Instanziieren von Widgets durch Konstruktor-Aufrufe erzeugt. Dabei kann es zu einer tiefen Verschachtlung (engl. nesting) kommen, die die Lesbarkeit erschwert. Daher sollten die Zustandsverwaltungssysteme diesem Problem mit geeigneten Maßnahmen entgegenwirken.

Abschließend wird auch hier eine Skala von “nicht erfüllt”, “teilweise erfüllt” bis “vollständig erfüllt” verwendet.

3.2.6 *Dokumentierung*

Bei der Dokumentation ist es entscheidend, dass sie für die Entwicklung hilfreich ist. Dabei sollte zuerst geprüft werden, ob es überhaupt eine entsprechende Dokumentation gibt und falls ja, ob die Grundkonzepte des Zustandsverwaltungssystems beschrieben sind. Zusätzlich stellt ein weiteres Kriterium das Zurverfügungstellen von umfangreichen Beispielen dar.

Von diesen Kriterien ausgehend wird eine qualitative Bewertung anhand einer Skala von "nicht erfüllt", "teilweise erfüllt" bis "vollständig erfüllt" verwendet.

3.2.7 *Strukturbestimmung*

Die Strukturbestimmung bewertet, inwiefern ein System Vorgaben an die Struktur der Anwendung stellt. Zusätzlich sollte untersucht werden, ob diese auch technisch forciert werden.

Dies wird auch mit einer qualitativen Bewertung von "nicht erfüllt", "teilweise erfüllt" bis "vollständig erfüllt" bewertet.

3.3 ZUSAMMENFASSUNG

Zusammengefasst ergibt sich aus den Anforderungen und den Bewertungskriterien eine Bewertungsmatrix, wie in Tabelle 3.1 zu sehen ist, welche im folgenden Teil auf einen Entwurf für jedes Zustandsverwaltungssystem angewandt werden soll.

ANFORDERUNG	METRIK	SKALA
Änderbarkeit / Skalierbarkeit	qualitativ	nicht teilweise vollständig } erfüllt
Testbarkeit	qualitativ	nicht teilweise vollständig } erfüllt
Effizienz	Anzahl der Widget-Rebuilds	$\mathbb{N};\mathbb{N}$
Komplexität/Wartbarkeit	MI	0-100
Verständlichkeit / Lesbarkeit	qualitativ	nicht teilweise vollständig } erfüllt
Dokumentierung	qualitativ	nicht teilweise vollständig } erfüllt
Strukturbestimmung	qualitativ	nicht teilweise vollständig } erfüllt

Tabelle 3.1: Bewertungsmatrix

4

UMSETZUNG UND REALISIERUNG

Nachdem nun die Anforderungen an die Zustandsverwaltungssysteme definiert worden sind, bedarf es eines Versuchsaufbaus, um die Zustandsverwaltungssysteme einer Evaluation unterziehen zu können.

Um die Evaluation möglichst praxisnah zu gestalten, ist es sinnvoll, diese anhand einer Beispielanwendung durchzuführen. Dies bietet die Vorteile, dass die Zustandsverwaltungssysteme im Test realistischen Bedingungen unterzogen werden und somit die Aussagekraft der Evaluation höher ist. Dieses Vorgehen birgt allerdings das Risiko, dass andere Variablen, die nicht Teil der Evaluation sind, die Ergebnisse beeinflussen. Um dieses Risiko zu minimieren werden Maßnahmen getroffen, die sicherstellen sollen, dass ausschließlich die Auswirkungen der Implementierung mit einem Zustandsverwaltungssystem überprüft werden.

4.1 ANFORDERUNGEN AN DIE BEISPIELANWENDUNG

Bevor nun eine solche Beispielanwendung konstruiert wird, ist es erforderlich, Anforderungen an diese zu definieren und darauf aufbauend einen Entwurf zu planen.

4.1.1 *Implizierte Anforderungen*

Die wichtigste Anforderung an die Beispielanwendung ist es, dass sie es ermöglicht eine Evaluation anhand der in Tabelle 3.1 definierten Kriterien durchzuführen. Im Folgenden sollte daher erst die Bewertungskriterien darauf untersucht werden, ob sie spezielle Anforderungen an eine Beispielanwendung stellen. Kriterien, die keine speziellen Anforderungen stellen, werden im Folgenden nicht weiter aufgeführt.

Bei der Änderbarkeit/Skalierbarkeit ist es nötig, einen gewissen Grad von Kopplung von Zuständen zu erreichen, damit die entsprechende Funktion der Zustandsverwaltungssysteme getestet werden kann.

Zur Überprüfung der Testbarkeit ist es erforderlich, Widgets so zu erstellen, dass sie im Rahmen der Implementierung von automatisierten Tests diese Implementierung nicht behindern. Zudem sollte es Anwendungsfälle geben, die auch die Verarbeitung von Zuständen erfordert, um eine gewisse Art von Geschäftslogik mit den Zustandsverwaltungssystemen implementieren und im Anschluss testen zu können.

Die Messung der Effizienz erfolgt an verschiedenen Messpunkten. Daher ist es erforderlich, Widgets zur Verfügung zu stellen, die dafür geeignet sind. Besonders gut geeignet sind dabei Widgets, die außerhalb einer Listendarstellung sind, und somit unabhängig vom Rendering der Liste sind.

Um das Verhalten bei tiefer Verschachtlung (engl. nesting) für das Kriterium der Verständlichkeit / Lesbarkeit prüfen zu können, ist es erforderlich dieses zu provozieren. Am besten kann dies umgesetzt werden, indem es Seiten gibt, die mehrere voneinander unabhängige, seitenweite Zustände haben.

4.1.2 *Explizite Anforderungen*

Nachdem nun die impliziten Anforderungen definiert worden sind, sollten nun weitere Anforderungen an die Beispielanwendung gestellt werden, um andere Faktoren mitberücksichtigen zu können. In dieser Untersektion werden nun andere explizite Anforderungen aufgestellt, die dafür sorgen sollen, dass ein möglichst praxisnahe Evaluation entsteht und somit ein realistisches Szenario getestet werden kann.

Die Beispielanwendung sollte so mehrere Seiten haben, um das Verhalten der Zustandsverwaltung auch nach einer Navigation testen zu können. Hier ist es wichtig, dass auf allen Seiten in der App die Daten persistent sind, und sich nicht unterscheiden.

In einer mobilen Anwendung werden oft Eingabemöglichkeiten für Nutzer*innen bereitgestellt, die den Zustand der Anwendung verändern. Daher sollte auch diese Beispielanwendung eine Form von Zustands-verändernden Interaktion besitzen.

Auch das Laden von Daten für den Anwendungszustand aus anderen Quellen wie beispielsweise von einem Server sollte implementiert werden, damit im gleichen Zug auch getestet werden kann, was im Falle einer Zeitüberschreitung oder eines Verbindungsfehlers passiert und wie die Fehlerbehandlung der Zustandsverwaltung funktioniert.

4.2 KONZEPTION EINER BEISPIELANWENDUNG

Anhand der aufgestellten Kriterien wird im folgenden Abschnitt die Beispielanwendung für die Evaluation konzipiert.

Als Anwendungsfall für die Beispielanwendung wurde hier die Produktauswahl- und Warenkorbansicht eines Internetshops gewählt. Dieser Anwendungsfall bietet den Vorteil, das mit ihm alle gestellten Anforderungen umgesetzt werden können und gleichzeitig ein realistisches Szenario zum Einsatz gelangt.

Grundlage bildet eine Liste mit verschiedenen Produkten, die aus dem Internet geladen wird, aus der die Benutzer*innen eine gewisse Anzahl eines oder mehrerer Produkte in einen Warenkorb legen können. Zudem existiert eine Warenkorbansicht, in der alle Produkte und ein Gesamtpreis zu sehen sind, die sich aktuell im Warenkorb befinden. Hiermit werden bereits alle expliziten Anforderungen erfüllt.

4.2.1 *Anwendung der impliziten Anforderungen*

Aufbauend auf diesen Anwendungsfall werden nun einzelne Funktionen anhand der gestellten Anforderungen beschrieben.

Um die Koppelung verschiedener Zustände testen zu können, wird in der Beispielanwendung die Möglichkeit angeboten, sich als registrierte*r Benutzer*in an- und abzumelden. Dies geschieht durch einen Schalter, welcher den aktuellen Anmeldezustand darstellt. Die Kopplung besteht nun darin, dass registrierten Benutzer*innen ein Rabatt von 20 % gewährt wird, und somit eine Kopplung zwischen der Berechnung des Warenkorbgesamtpreises und des Anmeldezustands notwendig wird.

Gleichzeitig wird hier auch die Anforderung zum Testen der tiefen Verschachtlung (engl. nesting) erfüllt, da hier mehrere unabhängige Zustände auf einer Seite benötigt werden.

Durch die Berechnung des Warenkorbgesamtpreises wird zudem die Anforderung der Testbarkeit umgesetzt, da hier ein geforderter Anwendungsfall für die Implementierung von Geschäftslogik in einem Zustand umgesetzt wird.

Um die Messung der Effizienz umsetzen zu können, wird ein Button hinzugefügt, welcher anzeigt, wie viele Produkte sich im Warenkorb befinden. Dieser Button hat zusätzlich die Aufgabe nach Berührung zur Detailansicht

des Warenkorbs zu wechseln. Da dieser Button außerhalb der Listendarstellung angebracht werden soll, ist er auch unabhängig von Änderungen dieser Liste.

4.2.2 *Wireframes*

Nachdem nun der Funktionsumfang der Anwendung beschrieben wurde, wird daraus eine Benutzeroberfläche konzipiert. Dazu werden vor der Umsetzung in Flutter ein Entwurf anhand von Wireframes erstellt. Diese zeigen, wie in Abbildung 4.1 dargestellt, die einzelnen Seiten (engl. screens) der App mit einem konsistenten Beispielzustand sowie die Navigationsmöglichkeiten, welche hier mit einem roten Pfeil angedeutet werden.

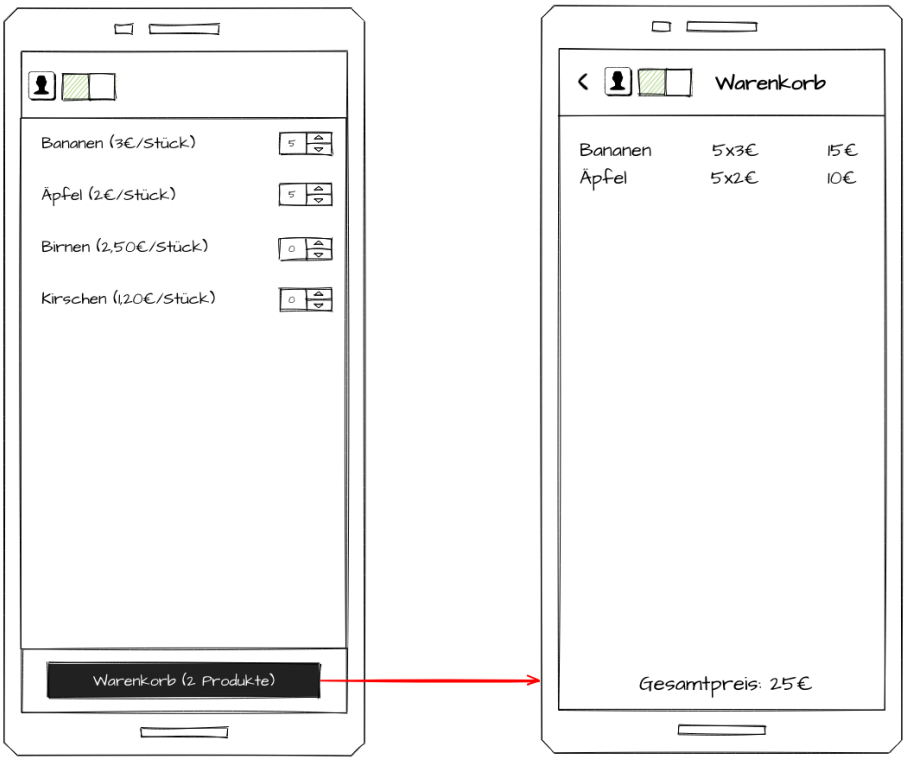


Abbildung 4.1: Wireframes der Beispielanwendung

4.3 VORGEHEN BEI DER IMPLEMENTIERUNG

Um diese Beispielanwendung für die Evaluation passend umsetzen zu können, ist eine nähere Beschreibung des Vorgehens bei der Implementierung notwendig, um den späteren Versuchsaufbau nachvollziehen zu können.

Für jedes zu evaluierende Zustandsverwaltungssystem muss eine eigene Anwendung entwickelt werden. Um sowohl den Aufwand der Implementierung zu verringern und die Vergleichbarkeit zu verbessern, bietet es sich an, eine gemeinsame Grundlage für alle Anwendungen zu schaffen. Diese sollte nur die Benutzeroberfläche mit Platzhaltern, sowie die Geschäftslogik zum Abrufen von Daten für die Produktliste beinhalteten.

Hierfür wird das Versionsverwaltungssystem Git eingesetzt. Die einzelnen Anwendungen für die Zustandsverwaltungssysteme zweigen dabei von einem gemeinsamen Arbeitszweig ab und setzen darin ihre eigenen Anpassungen um.

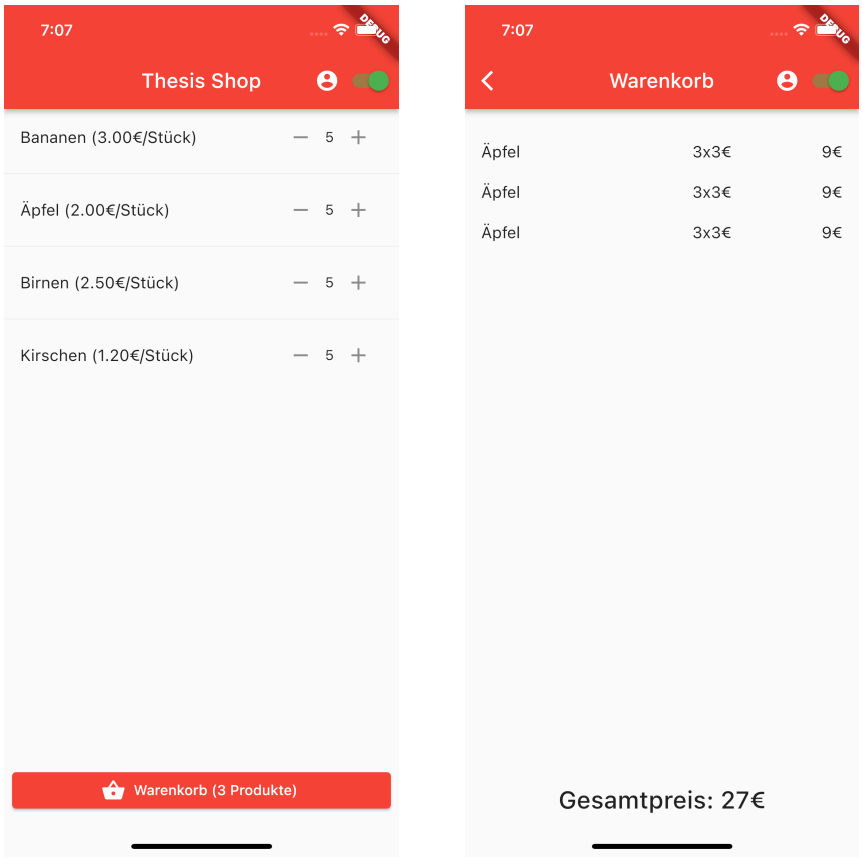
Außerdem wird in dem gemeinsamen Arbeitszweig die Versuchsvorrichtung für die Effizienz-Messung bereits vorbereitet, damit für alle Zustandsverwaltungssysteme das gleiche Messverfahren zum Einsatz kommen kann.

Der gesamte Quelltext wird in einem Git-Repository [Fra22] öffentlich zur Verfügung gestellt.

4.4 ERGEBNIS DER BEISPIELANWENDUNG

Die Beispielanwendung wurde im Material-Design mit Flutter in der Version 2.10.1 erstellt. Das Design weicht dabei nur leicht von dem in den Wireframes konzipierten Design ab, wie in Abbildung 4.2 zu sehen ist.

Die Funktionen sind abgesehen von der Navigation noch nicht implementiert, da dies, wie beschrieben, durch die Implementierung mit den Zustandsverwaltungssystemen erfolgt.



(a) Produktliste (b) Warenkorb

Abbildung 4.2: Umsetzung der Wireframes in Flutter unter iOS

4.4.1 *Versuchsaufbau*

Neben den sichtbaren Elementen der Anwendung wurde auch eine Infrastruktur zur Durchführung der Messung für die Effizienz und Komplexität / Wartbarkeit geschaffen.

Die Messung der Komplexität / Wartbarkeit erfolgt über die Metrik des Maintainability Index. Dieser Wert wird mit dem Tool `dart_code_metrics` ermittelt. Um die Berechnung unabhängig von der Laufzeitumgebung zu machen, wird die Berechnung in einem Docker-Container mit einem Continous-Integration-System (CI) durchgeführt. Die einzelnen Durchführungsschritte sind dabei in der CI-Konfigurationsdatei [Fra22, .drone.yml] im Unterschritt `generate_metrics` dokumentiert. Die vollständigen Testergebnisse sind dieser Ausarbeitung im Anhang beigefügt.

Für die Messung der Effizienz wird an zwei Messpunkten gemessen, wie oft ein Widget neu gebaut werden muss. Dazu wird ein Zähler erhöht, wenn die Build-Methode des entsprechenden Widgets aufgerufen worden ist. Zur Messung wird ein automatisierter Test [Fra22, test/efficiency_bechmark_test.dart] durchgeführt, welcher mehrere Produkte zum Warenkorb hinzufügt und wieder entfernt, und die An- und Abmeldung des Benutzenden auslöst. Im letzten Schritt wird zur Warenkorbansicht navigiert.

Die beiden Messpunkte wurden dabei an folgenden Stellen angebracht. Der erste Messpunkt `cartButton` wird ausgelöst beim Rendern des Warenkorb-Buttons und der zweite Messpunkt `userSwitch` wird ausgelöst beim Rendern des Schalters zum An- und Abmelden. Diese beiden Widgets wurden gewählt, da sie von Zustandsänderungen betroffen sind, allerdings nicht von allen Zustandsänderungen der Anwendung und außerhalb einer Listenansicht sind.

Beim Ausführen des Tests für die beschriebene Basisanwendung ohne Zustandsverwaltung wurde folgendes Ergebnis ermittelt:

Anzahl der Rendervorgänge:

Listing 4.1: Anzahl der Rendervorgänge

```
00:00 +0: /app/test/efficiency_bechmark_test.dart: Efficiency Test
cartButton: 1
userSwitch: 2
00:01 +1: All tests passed!
```

Diese Ergebnisse stellen die Minimalwerte für den Test dar.

5

EVALUATION

Nachdem nun das Grundgerüst der Beispielanwendung sowie deren Anforderungen definiert worden sind, können die Ansätze zur Zustandsverwaltung anhand der definierten Kriterien evaluiert werden.

Dafür wird für jedes Zustandsverwaltungssystem die Anwendung vollständig implementiert, und anschließend die Tests und Untersuchungen für die Bewertung durchgeführt.

Im darauf folgenden Kapitel werden dann die kumulierten Ergebnisse analysiert und ein Fazit gezogen.

5.1 SETSTATE

Das Erste zu evaluierende Zustandsverwaltungssystem stellt die in Unterunterabschnitt 2.3.2.1 beschriebe Vorgehensweise zur Verwaltung des Zustands dar.

Mit diesem Ansatz kann nicht die Mindestanforderungen an die Beispielanwendung umgesetzt werden, da es unmöglich ist, mit ihm einen konsistenten Zustand über mehre Seiten hinweg zu erzeugen. Dabei war es immer nur möglich, Zustandsänderungen auf einer Seite zu haben, sobald man auf eine andere Seite navigiert, wurden auf der alten Seite die Änderungen nicht übernommen.

5.2 INHERITEDWIDGET

Wie in Unterunterabschnitt 2.3.2.2 beschrieben, stellen `InheritedWidgets` eine Option für die Zustandsverwaltung dar, die ohne externe Bibliotheken auskommt und somit nur Bordmittel des Flutter-Frameworks benutzen.

5.2.1 *Implementierung*

Für die Implementierung dieses Ansatzes wurden mehrere Stores konstruiert, welche den Anmeldezustand, die geladenen Produkte und die im Warenkorb befindliche Anzahl der Produkte modellieren. Ein Store besteht dabei immer aus einem `InheritedWidget` und einem `StatefulWidget`. Das `InheritedWidget` erhält dabei vom `StatefulWidget` die Daten und Methoden, welche nach außen abrufbar sein sollen. Zustandsänderungen und Verknüpfungen mit anderen Stores finden ausschließlich im `StatefulWidget` statt.

Die Benutzeroberfläche greift auf die Stores ausschließlich über die `InheritedWidgets` zu, wie bereits im Unterunterabschnitt 2.3.2.2 beschrieben wurde.

Der Funktionsumfang konnte dabei ohne Einschränkungen vollständig implementiert werden.

5.2.2 *Bewertung*

Im folgenden Abschnitt wird die Implementierung mit `InheritedWidget` [Fra22, branch=inheritedwidget] anhand der definierten Bewertungskriterien bewertet.

ÄNDERBARKEIT/SKALIERBARKEIT Bei der Änderbarkeit und Skalierbarkeit wird dieses Zustandsverwaltungssystem mit "teilweise erfüllt" bewertet.

Eines der Merkmale von skalierbaren und änderbaren Zustandsverwaltungssystemen ist es, dass sich mehrere Zustände untereinander verknüpfen lassen. Dies ist hier eingeschränkt möglich, da um auf einen anderen Zustand zuzugreifen es erforderlich ist, dass dieser im Widget-Baum oberhalb des zugreifenden Zustandswidgets angeordnet ist. Falls dies der Fall ist, lässt sich der Zustand aber wie bei der Benutzeroberfläche (vgl. Unterunterabschnitt 2.3.2.2) über einen Methoden-Aufruf abrufen.

Negativ bewertet wird hier aber die Tatsache, dass ein Zustand beziehungsweise Store immer aus mindestens drei Klassen bestehen muss. Dies macht es erforderlich, alle nach außen zugängliche Funktionen oder Variablen per Konstruktoraufruf von dem `StatefulWidget` zum `InheritedWidget` zu übergeben. Bei wachsender Größe eines Zustands wird dies unübersichtlich.

Ein weiter Nachteil ist es, dass es erforderlich ist, im `InheritedWidget` eine Methode zu implementieren, die feststellt, ob sich der Zustand im Vergleich zu einem anderen Zustand verändert hat. Dies wird bei wachsender Größe oder Datenstrukturkomplexität eines Zustands unübersichtlich und auch komplex.

TESTBARKEIT Zur Bewertung der Testbarkeit wurden mehrere Tests entsprechend der Anforderungen implementiert.

Das Testen der Geschäftslogik der Stores stellte sich als komplex heraus, da hier keine Unit-Tests, sondern Widget-Tests vonnöten waren. Dies hat auch zur Folge, dass man sich während der Tests auch um den Lebenszyklus der Widgets kümmern muss. So muss man im Test die Hilfs-Funktion `tester.pump()`, wie in Listing 5.1 zu sehen ist, zum richtigen Moment aufrufen, damit das Framework die Zustandsänderungen umsetzt.

Listing 5.1: Aufbau eines einfachen Flutter-Widgets in Dart

```
testWidgets('test cart store', (tester) async {
    final widgetTree = ProductStore(
      products: RemoteResource.finished(demoProducts),
      child: CartStoreImplementation(child: Container()),
    );
    await tester.pumpWidget(widgetTree);
    final cartStoreFinder = find.byType(CartStore);
    expect(cartStoreFinder, findsOneWidget);
    var cartStore = tester.widget<CartStore>(cartStoreFinder);
    cartStore.increaseAmount(demoProducts.first);
    await tester.pump();
    cartStore = tester.widget<CartStore>(cartStoreFinder);
    expect(cartStore.amountOfProduct(demoProducts.first), 1);
});
```

Die Tests für Widgets, die den Zustand konsumieren auf der anderen Hand sind einfach umzusetzen, da die Geschäftslogik im besten Fall komplett von der Speicherung der emittierten Daten getrennt wird. So ist es möglich ein `InheritedWidget` ohne das dazugehörige `StatefulWidget` zu initialiseren und die benötigten Mock-Daten über den Konstruktor zu übergeben.

Abschließend erfolgt hier die Bewertung mit “teilweise erfüllt”.

EFFIZIENZ Nach der Ausführung der Teststrecke ergaben die Zähler folgendes Ergebnis:

Listing 5.2: Anzahl der Rendervorgänge bei InheritedWidget

```
00:00 +0: /app/test/efficiency_bechmark_test.dart: Efficiency Test
cartButton: 8
userSwitch: 4
00:02 +1: All tests passed!
```

KOMPLEXITÄT / WARTBARKEIT Die Auswertung der Metriken (vgl. Abschnitt A.2) ergab ein MI von 83 für das gesamte Projekt.

VERSTÄNDLICHKEIT / LESBARKEIT Bei der Bewertung der Lesbarkeit sind mehrere Faktoren wichtig.

Der erste Faktor ist die Frage, ob neue Konzepte eingeführt werden. Diese Frage kann mit Nein beantwortet werden, da hier ausschließlich Komponenten aus dem Flutter-Framework in Form verschiedener Widgetes zum Einsatz kommen.

Der zweite Faktor hierbei ist die Frage, inwiefern die Struktur klar nachvollziehbar und verständlich ist. Dies ist bei `InheritedWidgets` nicht der Fall, da besonders die Konstruktion aus mehreren Widgets und Widget-Typen schwer nachvollziehbar ist. So gibt es keinen zentralen Ort, welcher die Geschäftslogik übersichtlich zusammenfasst. Im Gegenteil ist es hier notwendig, die Geschäftslogik als Teil eines `StatefulWidgets` zu implementieren, was die Lesbarkeit definitiv erschwert. Hinzu kommt der große Umfang an Boilerplate-Quelltext, da selbst für einfache Zustände wie das Speichern des Anmeldezustands drei Klassen benötigt werden.

Der letzte Faktor befasst sich mit der tiefen Verschachtlung von Widgets (engl. Nesting). Nesting wirkt sich negativ auf die Lesbarkeit von Widgets aus. Auf Nesting kann allerdings bei diesem Ansatz nicht verzichtet werden, da `InheritedWidgets` in die Baumstruktur eingebaut werden, wie in Listing 5.3 zu sehen ist, und somit immer ein Child-Widget übergeben bekommen.

Listing 5.3: Nesting bei `InheritedWidgets` in app.dart [Fra22]

```
return UserStoreImplementation(
    child: ProductStoreImplementation(
      productService: productService,
      child: CartStoreImplementation(child: child),
    ),
);
```

Zusammengefasst wird die Verständlichkeit/Lesbarkeit mit "nicht erfüllt" bewertet.

DOKUMENTIERUNG Die Dokumentation gibt ein zwiegespaltenes Bild ab, da zum einen die einzelnen Widget-Typen an sich ausführlich beschrieben werden, allerdings das Zusammenspiel als Zustandsverwaltungssystem als Ganzes in der offiziellen Entwicklerdokumentation gar nicht beleuchtet wird. Dieses Zusammenspiel wird lediglich in Ressourcen, außerhalb der Dokumentation beschrieben. Somit fehlen auch umfangreiche Beispiele, wie das Zustandsverwaltungssystem implementiert oder genutzt werden kann.

Aufgrund der großen Bandbreite an Dritt-Ressourcen zu `InheritedWidgets` erfolgt hier noch eine Bewertung mit "teilweise erfüllt".

STRUKTURBESTIMMUNG Das Konzept mit `InheritedWidgets` stellt keine starren Strukturvorgaben an die Anwendung. Lediglich das Zusammenspiel zwischen `InheritedWidget` und `StatefulWidget` wird dadurch forciert, dass `InheritedWidgets` von sich aus ihren Zustand nicht ändern können. Dies hat allerdings keinen Einfluss auf die Gesamtarchitektur der Anwendung, das `InheritedWidget` und `StatefulWidget` als gemeinsame Komponente betrachtet werden können.

Somit wird hier eine Bewertung mit "nicht erfüllt" vorgenommen.

5.3 BLOC

Business Logic Components sind ein weit verbreitetes Konzept zur Zustandsverwaltung. Die Grundlagen zu diesem Zustandsverwaltungssystem wurden bereits in Unterabschnitt 2.3.3 eruiert.

5.3.1 *Implementierung*

Für die Implementierung der App mit BLoC wurden drei Business Logic Components erstellt. Diese verwalten den Anmeldezustand, den Inhalt des Warenkorbs sowie die verfügbaren Produkte. Die BLoCs bestehen dabei aus zu teils mehreren `Streams` und `Sinks`, die Zustandsänderungen von außen entgegennehmen und nach außen kommunizieren. Zusätzlich dazu gibt es für die `Streams` öffentliche Variablen, welche den aktuellen Zustand wiedergeben. Dies dient, wie auch im Beispiel Listing 5.4 anhand des Parameters

`initialData` zu sehen ist, dazu, dass Widgets, die erst später zum Widget-Tree hinzugefügt werden den aktuellen Zustand zum Zeitpunkt des initialen Erstellens erhalten können, da die Streams immer nur Zustandsänderungen kommunizieren können.

Alle drei BLoCs werden über ein Widget an die Benutzeroberfläche weitergegeben, welches als Dependency Injection fungiert. Hier wären auch andere Implementierungsvarianten beispielsweise mit dem Service Locator `get_it` denkbar gewesen. Der gewählte Ansatz benötigt jedoch keine zusätzliche Bibliothek und verfälscht somit das Ergebnis am wenigsten. Die Widgets der Benutzeroberfläche verwenden dabei `StreamBuilder`, wie in Listing 5.4 zu sehen ist, um über aktualisierten Zuständen informiert zu werden.

Listing 5.4: Verwendung eines StreamBuilder in cart_button.dart [Fra22]

```
final cartBloc = AppState.of(context).blocProvider.cartBloc;
return StreamBuilder<int>(
  stream: cartBloc.numberOfProductsStream,
  initialData: cartBloc.numberOfProducts,
  builder: (context, snapshot) {
    BenchmarkCounters.cartButton++;
    return ElevatedButton.icon(
      onPressed: () => Navigator.of(context).pushRouteKey(RouteKey.cart)
          ,
      icon: const Icon(Icons.shopping_basket),
      label: Text('Warenkorb (${snapshot.requireData} Produkte)'),
    );
  },
);
```

5.3.2 *Bewertung*

Im folgenden Abschnitt wird die Implementierung mit BLoC [Fra22, branch=bloc] anhand der definierten Bewertungskriterien bewertet.

ÄNDERBARKEIT/SKALIERBARKEIT Zu BLoC lässt sich sagen, dass diese sowohl skalierbar als auch änderbar sind. Diese Aussage stützt sich darauf, dass durch die Kommunikation ausschließlich über `Streams` und `Sinks` sich eine einheitliche Schnittstelle schaffen lässt, an die beliege Komponenten sich andocken können. Anders als bei den Widget-basierten Ansätzen wie `InheritedWidget` lassen sich BLoC komplett plattformunabhängig einsetzen und sind somit für eine weitere Skalierung außerhalb des Flutter-Frameworks durchaus geeignet.

Die Koppelung verschiedener Zustände lässt sich somit auch einfach lösen, indem andere BLoC beispielsweise über den Konstruktor oder ein Dependency-Injection Tool übergeben werden.

Aufgrund dieser Eigenschaften wird die Bewertung "vollständig erfüllt" vergeben.

TESTBARKEIT Zur Testbarkeit von BLoC lässt sich Folgendes sagen. Die Geschäftslogik der einzelnen BLoC lässt sich gut testen, da das Flutter-Testing-Framework bereits Werkzeuge zum Überprüfen von Streams bereithält, wie sie im Test in Listing 5.5 verwendet werden. Hier lassen sich somit auch einfach Unit-Tests verwenden. Falls BLoC Abhängigkeiten zu anderen BLoC oder Komponenten haben, müssen diese allerdings gemockt werden. Ohne eine entsprechende Bibliothek entstehen hier zusätzliche Aufwände durch das Implementieren von Mock-Klassen.

Listing 5.5: Test des Cart BLoC in cart_bloc_test.dart [Fra22]

```
test('test cart bloc', () async {
  const productBlocMock = ProductBlocMock();
  final cartBloc = CartBloc(productBlocMock);
  final expectedCart = {demoProducts.first: 1};
  expectLater(cartBloc.cartStream, emits(expectedCart));
  cartBloc.quantityEventSink.add(IncreaseQuantityEvent(demoProducts.
      first));
});
```

Ein weiterer betrachteter Aspekt beim Testen ist die Testbarkeit von Widgets, die auf BLoC zugreifen. Hierzu lässt sich sagen, dass das Ersetzen durch Platzhalter ohne Probleme möglich war. Allerdings hängt dies auch von dem verwendeten Injection-System ab. In diesem Beispiel wurden `InheritedWidgets` verwendet. Daher kann dies äquivalent zu dem dort beschriebenen Beobachtungen (vgl. 5.2.2) gewertet werden. Zusätzlich dazu lässt sich sagen, dass für das Erzeugen eines Platzhalters (engl. mock) für ein BLoC die Klasse dieses BLoC vollständig gemockt werden muss. Hier lassen sich wie im vorhergehenden Absatz beschrieben, mehrere Strategien nutzen.

Aufgrund der einfachen Testbarkeit der Geschäftslogik sowie des möglichen Austausches mit Platzhaltern für Widget-Tests wird die Testbarkeit mit "vollständig erfüllt" bewertet.

Effizienz Nach der Ausführung der Teststrecke ergaben die Zähler folgendes Ergebnis:

Listing 5.6: Anzahl der Rendervorgänge bei BLoC

```
00:00 +0: /app/test/efficiency_bechmark_test.dart: Efficiency Test
cartButton: 8
userSwitch: 4
00:02 +1: All tests passed!
```

Komplexität / Wartbarkeit Die Auswertung der Metriken (vgl. Abschnitt A.3) ergab ein MI von 82 für das gesamte Projekt.

Verständlichkeit / Lesbarkeit Die Verständlichkeit und Lesbarkeit bei BLoC wird anhand der in Unterabschnitt 3.2.5 beschriebenen Fragestellungen bewertet.

Zur Verwendung von BLoC ist es erforderlich, dass man sich mit Konzepten der asynchronen Programmierung beschäftigt, um die Funktionsweise von `Streams` und `Sinks` verstehen zu können. Diese werden in der Flutter-Programmierung sonst selten benötigt. Daher ist davon auszugehen, dass Entwickler*innen diese erst lernen müssen.

Die Struktur der BLoC selber ist schwer nachvollziehbar, da die Kombination von Streams, Sinks und Variablen nicht auf den ersten Blick nachvollziehbar ist. Zudem sind die Schnittstellen zur Benutzeroberfläche nicht wie andere Schnittstellen den Sprachfluss angepasst, sondern werden bestimmt durch die jeweiligen Funktionen von `Streams` und `Sinks`. Außerdem benötigt dieser Ansatz eine Großzahl von Klassen, um beispielsweise Argumente von Ereignissen und Zustände abbilden zu können. Dies erschwert zusätzlich die Lesbarkeit, wie auch von Kuzmin, Ignatiev und Grafov festgestellt wurde. [KIG20, S. 573]

Die tiefe Verschachtlung von Widgets kann bei BLoC nicht beobachtet werden, da hier nicht das Widget-System als Grundlage zum Einsatz kommt. Allerdings verlangt die Verwendung von `Streams` den Einsatz von `StreamBuildern` in der Benutzeroberfläche, wie bereits in Listing 5.4 gezeigt wird. Dies gibt zwar auf der einen Seite große Kontrolle über die Verarbeitung von Zustandsänderungen, macht den Quelltext allerdings schwer lesbar, besonders bei der Verwendung von mehreren `StreamBuildern` ineinander.

Zusammenfassend wird die Verständlichkeit und Lesbarkeit mit der Bewertung “nicht erfüllt” bewertet.

DOKUMENTIERUNG Die Dokumentation von BLoC ist schwierig zu bewerten, da es sich bei BLoC lediglich um ein Konzept handelt. Allerdings gibt es Bibliotheken, welche das Konzept mit Hilfs-Konstrukten versehen und so den Einsatz von BLoC einfacher machen. Die Dokumentation dieser Bibliotheken umfasst umfangreiche Erklärungen und Beispiele [Ang21], die auch auf BLoC an sich übertragbar sind.

Zusätzlich bestehen eine große Anzahl an Artikeln und anderen Veröffentlichungen zum Thema BLoC.

Aufgrund der umfangreichen Dokumentation der beschriebenen Bibliothek und der Großzahl an Veröffentlichungen zu BLoC, wird die Dokumentation mit "vollständig erfüllt" bewertet.

STRUKTURBESTIMMUNG Zur Strukturbestimmung lässt sich sagen, dass BLoC strikte Vorgaben zur Konstruktion von BLoC macht, aber viele Aspekte wie die Injection in Widgets oder die Dependency Injection innerhalb der BLoCs unbeantwortet bleibt. Eine technische Forcierung der Struktur eines BLoC ist nur teilweise erkennbar, da die Verwendung von `StreamBuilder` die Existenz von `Streams` voraussetzt. Weitere technische Forcierungen der Struktur sind auch mangels des Einsatzes einer Bibliothek nicht zu erkennen.

Daher wird die Strukturbestimmung mit "teilweise erfüllt" bewertet.

5.4 PROVIDER

Provider stellt die erste externe Bibliothek zur Zustandsverwaltung dar, die in dieser Ausarbeitung evaluiert wird. Die Grundlagen dazu sind im Unterabschnitt 2.3.4 nachzuvollziehen.

5.4.1 *Implementierung*

Für die Implementierung der Zustandsverwaltung mit der Provider Bibliothek wurde die Version 6.0.2 eingesetzt. Die Zustände wurden dabei in 4 Stores beziehungsweise Providern umgesetzt. Für die Verwaltung des Anmeldezustands wurde ein ChangeNotifier mit gleichnamigen Provider verwendet (siehe dazu Listing 2.5). Zum Laden der Produkte wurde eine FutureProvider eingesetzt. Dieser stellt das Ergebnis eines Futures, also einer asynchronen Operation, zur Verfügung und kann auch auf Fehler beim Warten auf das Future reagieren. Für die Produktliste wurde eine ProxyProvider2 einge-

setzt. Dieser ermöglicht es, die Zustände zweier verschiedener Provider zu kombinieren. In diesem Fall wurde der Zustand der beiden bereits beschriebenen Provider kombiniert, um den Rabatt anzuwenden. Für die Abbildung des Warenkorbs wurde ein ChangeNotifierProxyProvider verwendet. Dies stellt eine Mischung aus einem ProxyProvider und einem ChangeNotifier-Provider dar, und ermöglicht es so ChangeNotifier-Klassen, über Änderungen von anderen Providern zu informieren.

In die Benutzeroberfläche eingebunden, werden die Provider über diverse Extension-Funktionen für den BuildContext.
Dabei lässt sich der Zustand abonnieren oder exakt einmal abrufen, falls keine Aktualisierungen erwünscht sind. Zudem lässt sich auch ein Teil des Zustands selektieren, was dazu führt, dass das Widget nur aktualisiert wird, wenn sich dieser Teil des Zustands tatsächlich ändert.

5.4.2 *Bewertung*

Im folgenden Abschnitt wird die Implementierung mit Provider [Fra22, branch=provider] anhand der definierten Bewertungskriterien bewertet.

ÄNDERBARKEIT/SKALIERBARKEIT Das Thema Skalierbarkeit und Änderbarkeit zeigt sich als ambivalent bei Providern. Bei der Änderbarkeit muss man sagen, dass hier Provider einen Vorteil bietet, da durch das Konzept der Provider eine einheitliche Abstraktion für Zustandsverwaltungen geschaffen werden. Somit ist es nicht von Belang, ob ein Zustand über einen Stream, Future oder einen ChangeNotifier abgebildet werden. Dies bietet damit einen Vorteil für Änderbarkeit, da sich Zustände zu einem späteren Zeitpunkt einfach auf ein anderes Zustandsverwaltungsmodell abändern lassen, ohne das Anpassungen außerhalb dieses Zustands vonnöten wären.

Eine Designentscheidung von Provider zum Nachteil der Änderbarkeit ist es, dass Provider alleine über Ihren Klassentyp per Generic in Widgets injiziert werden. Dabei kann zur Kompilierungszeit nicht geprüft werden, ob der referenzierte Zustand sich auch tatsächlich in der Widget-Hierarchie befindet. Somit können bei Änderungen in der Hierarchie von Widgets zu einem späteren Zeitpunkt Laufzeitfehler auftreten, die nicht durch den Compiler gefunden werden können.

Zur Skalierbarkeit lässt sich sagen, dass in der Dokumentation bereits Limitationen für den Einsatz von Providern gezeigt werden. So träten ab der Verwendung von mehr als 150 Providern StackOverflowErrors auf. [Rou22b]
Zudem ist die Koppelung von Zuständen stark begrenzt. Dies liegt daran, dass es für jede Anzahl der zu koppelnden Zustände eine entsprechende

Klasse in der Provider-Bibliothek geben muss. Aktuell ist dies mit der Klasse ProxyProvider6 auf sechs Klassen begrenzt. Ob diese Limitation in der Realität jedoch tatsächlich relevant ist, ist fraglich, da es wohl architekturell besser wäre in diesem Fall, die Zustände in mehrere Zustände zusammenzufassen.

Aufgrund der eingeschränkten Skalierbarkeit, wird die Änderbarkeit/Skalierbarkeit mit "teilweise erfüllt" bewertet.

TESTBARKEIT Zur Testbarkeit lässt sich sagen, dass sowohl die Geschäftslogik als auch verwendende Widgets sich ohne Änderungen am Quelltext der Anwendung testen lassen.

Bei der Geschäftslogik lassen sich die Zustandsklassen komplett ohne die Verwendung von Flutter oder von Provider an sich testen. Somit ist es auch nicht nötig, etwaige Abhängigkeiten zu mocken oder Widget-Tests zu benutzen.

Bei den Tests der Widgets, die auf Zustände zugreifen, lässt sich sagen, dass das Mocking hier unaufwändig ist, da Provider die jeweiligen Zustände abstrahiert und so es möglich ist, einfache "Value"-Provider zu injizieren, wie in Listing 5.7 zu sehen ist. Diese können dann wie im Beispiel zu sehen ist, einfache Klassen injizieren, die die Eigenschaften der ursprünglichen Zustandsklasse überschreiben.

Listing 5.7: Widget-Test bei Provider in total_price_test.dart [Fra22]

```
class CartStoreMock extends CartStore {
  @override
  double get totalPrice => 10.0;
}

void main() {
  testWidgets('test total price', (tester) async {
    final widgetTree = MaterialApp(
      home: ChangeNotifierProvider.value(
        value: CartStoreMock() as CartStore,
        child: const TotalPriceText(),
      ),
    );
    await tester.pumpWidget(widgetTree);
    final correctText = find.text("Gesamtpreis: 10.00");
    expect(correctText, findsOneWidget);
  });
}
```

Da sowohl Widget-Tests als Geschäftslogik-Tests implementierbar waren, wird die Testbarkeit mit "vollständig erfüllt" bewertet.

EFFIZIENZ Nach der Ausführung der Teststrecke ergaben die Zähler folgendes Ergebnis:

Listing 5.8: Anzahl der Rendervorgänge bei Provider

```
00:00 +0: /app/test/efficiency_bechmark_test.dart: Efficiency Test
cartButton: 3
userSwitch: 4
00:02 +1: All tests passed!
```

KOMPLEXITÄT / WARTBARKEIT Die Auswertung der Metriken (vgl. Abschnitt A.4) ergab ein MI von 83 für das gesamte Projekt.

VERSTÄNDLICHKEIT / LESBARKEIT Die Verständlichkeit und Lesbarkeit lässt sich hier anhand mehrerer Aspekte bewerten.

Bei Provider werden hauptsächlich bereits bestehende Konzepte von Flutter erweitert. Dies kann beispielsweise an der Injizierung der Zustandsklassen in Widgets beobachtet werden. Hier wird eine ähnliche Syntax wie bei anderen Flutter-Komponenten genutzt wie auch im Beispiel in Listing 5.9 zu sehen ist. Jedoch muss hier auch angemerkt werden, dass es empfehlenswert ist, immer eine Zustandsklasse zu erstellen, auch wenn die an die Widgets zu übergebenden Zustände einem trivialen Datentyp wie einem `String` entsprechen, da für das Injizieren immer der Klassen-Name der zu injizierenden Klasse verwendet wird. So ist es also nicht möglich, mehrere Zustände vom Type `String` gleichzeitig zu verwenden. Diese zusätzlichen Klassen können somit die Lesbarkeit erschweren.

Listing 5.9: Vergleich des Abrufs von Zuständen zwischen Flutter und Provider

```
// Abruf der Bildschirmausrichtung
// aus der Flutter-Standardbibliothek
MediaQuery.of(context).orientation;
// Abruf des Einkaufwagenzustands
Provider.of<CartStore>(context).cart;
```

Die Zustandsklassen an sich führen auch keine neuen Konzepte ein und Verwenden bereits bekannte Konzepte wie die ChangeNotifier. Diese Zustandsklassen bieten auch eine gute Lesbarkeit, da hier direkt über Variabeln auf den Zustand zugegriffen wird und über Funktionen dieser direkt geändert wird. Hier muss also keine Spezialbehandlung aufgrund der Zustandsverwaltung genutzt werden.

Auf der anderen Seite werden Konstrukte wie der ProxyProvider, welcher die Verknüpfung von mehreren Zuständen ermöglicht, schnell aufgrund der vielen Parameter unübersichtlich.

Der letzte untersuchte Aspekt, ist, ob das Zustandsverwaltungssystem, der tiefen Verschachtlung wirksam entgegentritt. Dazu lässt sich konstatieren, dass hier durch die Einführung eines sogenannten `MultiProvider` dieses Problem umgangen wird. Zwar muss bei Provider, alle Provider in den Widget-Baum eingesetzt werden, allerdings kann ein `MultiProvider` eine Liste von Providern entgegennehmen und in den Widget-Baum einsetzen. Somit kommt es nicht zu einer tiefen Verschachtlung.

Zusammenfassend wird dieser Bewertungsaspekt mit "teilweise erfüllt" bewertet.

DOKUMENTIERUNG Zur Dokumentation [siehe Rou22b] lässt sich sagen, das diese die Grundkonzepte verständlich erklärt und zusätzlich umfangreiche Beispiele beinhaltet. Jedoch ist die Struktur als einzelne Markdown-Datei nicht übersichtlich. Zusätzlich gibt es einen großen Umfang an Drittveröffentlichungen unter anderem in der offiziellen Flutter-Dokumentation.

Die Dokumentierung wird mit "vollständig erfüllt" bewertet.

STRUKTURBESTIMMUNG Zur Strukturbestimmung lässt sich sagen, dass Provider keinen wesentlichen Einfluss auf die Struktur der Anwendung hat. Zudem gibt sie kein einheitliches Vorgehensmodell bei der Implementierung einer Zustandsklasse vor, da hier wie bereits beschrieben, aus mehreren möglichen Modellen wie ChangeNotifier oder Stream gewählt werden kann.

Daher wird die Strukturbestimmung mit "nicht erfüllt" bewertet.

5.5 RIVERPOD

Riverpod stellt eine externe Bibliothek zur Zustandsverwaltung dar, welche das Konzept von Provider erweitert und laut eigenen Angaben verbessert. Die Grundlagen dazu sind im Unterabschnitt 2.3.5 nachzuvollziehen.

5.5.1 *Implementierung*

Für die Implementierung wurde die Bibliotheken `riverpod` und `flutter_riverpod` in der Version 1.0.3 verwendet.

Die Zustände wurden dabei über diverse Provider. Der Anmeldezustand beispielsweise wurde mittels eines StateNotifierProvider umgesetzt, welcher

eine StateNotifier-Klasse, wie im Grundlagenkapitel erläutert, injiziert. Zudem wurde ein FutureProvider genutzt, um die Produkte zu laden und ein einfacher Provider, um den Anmeldezustand mit den geladenen Produkten zu verknüpfen und somit den Rabatt anzuwenden. Der Zustand des Warenkorbs wird über mehrere Provider abgebildet. So stellt ein StateNotifier-Provider die Grundlage für den Warenkorb. Diese wird dann anschließend durch weitere Provider um zusätzliche Informationen ergänzt wie beispielsweise den Produktdaten. Alle Provider werden als globale finale Variablen der Benutzeroberfläche zur Verfügung gestellt.

Die Einbindung in die Benutzeroberfläche erfolgte mittels ConsumerWidget. Dafür werden die Basisklassen der Widgets, die auf Zustände zugreifen müssen auf ConsumerWidget geändert. Dadurch erhält man in der build-Methode einen zusätzlichen Parameter, der es erlaubt die Provider anhand ihrer globalen Variablen auszulesen.

5.5.2 *Bewertung*

Im folgenden Abschnitt wird die Implementierung mit Riverpod [Fra22, branch=
riverpod] anhand der definierten Bewertungskriterien bewertet.

ÄNDERBARKEIT/SKALIERBARKEIT Zur Änderbarkeit und Skalierbarkeit bei Riverpod lassen sich Parallelen zu Provider ziehen, da hier ähnliche Konzepte zum Einsatz kommen. So bietet Riverpod auch eine einheitliche Abstraktionsschicht für Zustände und kann somit ChangeNotifier, StateNotifier, Futures, Streams und einfache Werte als Zustand verwalten. Somit werden Änderungen in der Zukunft vereinfacht, da nur die jeweiligen Zustände und Provider angepasst werden müssen und nicht die davon abhängigen Zustände und andere Komponenten.

Im Gegensatz zu Provider wurde hier nicht der Weg der Injection via Generics gewählt, sondern auf Variablen gesetzt. Dies hat den Vorteil, dass mehrere Zustände vom gleichen Datentyp existieren können und es immer Sichergestellt ist, dass ein entsprechender Provider überhaupt im verwalteten Gesamtzustand existiert.

Da die Provider nicht über den Widget-Baum übergeben werden, ist anzunehmen, dass das Problem mit der Skalierbarkeit mit über 150 Providern, welches bei der Provider-Bibliothek auftritt, hier nicht Anwendung findet.

Auch bei der Koppelung verschiedener Zustände wird hier ein anderer Weg gegangen. So wir hier über einen Array angegeben, von welchen anderen

Provider ein Provider abhängt. Die Zustände können im Anschluss wie bei der Benutzung im ConsumerWidget über eine Referenzvariable `ref` abonniert, verändert oder gelesen werden. Somit skaliert dieser Ansatz der Kopplung mit der wachsenden Größe von zu koppelnden Zuständen und wird nicht durch Generics limitiert.

Aufgrund der Verbesserungen bei der Skalierbarkeit und Änderbarkeit gegenüber Provider wird diese mit "vollständig erfüllt" bewertet.

TESTBARKEIT Zur Bewertung der Testbarkeit wird zum einen geprüft, inwiefern sich die Geschäftslogik der einzelnen Zustandsklassen testen lässt, und inwiefern es möglich ist, verteilte Zustände bei Widget-Tests durch Platzhalter zu ersetzen.

Die Tests der Geschäftslogik sind ohne Probleme implementierbar. Hier kann komplett auf die Verwendung von Riverpod oder Flutter verzichtet werden, da die Zustandsklassen auch unabhängig von dem Zustandsverwaltungssystem ihre Geschäftslogik behalten. Somit wird hier ein einfacher Unit-Test, welcher die Funktion einer Klasse oder deren Funktionen testet implementiert.

Listing 5.10: Widget-Test bei Riverpod in total_price_test.dart [Fra22]

```
testWidgets('test total price widget', (tester) async {
    await tester.pumpWidget(
        MaterialApp(
        home: ProviderScope(
            overrides: [
            totalPriceProvider.overrideWithValue(10.0),
            ],
            child: const TotalPriceText(),
        ),
        ),
    );
    expect(find.text('Gesamtpreis: 10.00'), findsOneWidget);
});
```

Bei den Widget-Tests bietet Riverpod die Möglichkeit an, beliebig viele Provider entweder mit festen Inhalten oder mit anderen Providern zu überschreiben. Besonders die Möglichkeit mit dem Überschreiben durch feste Inhalte vereinfacht das Ersetzen durch Platzhalter, wie im Listing 5.10 zu sehen ist, sehr.

Die Testbarkeit wird daher mit "vollständig erfüllt" bewertet.

EFFIZIENZ Nach der Ausführung der Teststrecke ergaben die Zähler folgendes Ergebnis:

Listing 5.11: Anzahl der Rendervorgänge bei Riverpod

```
00:00 +0: /app/test/efficiency_bechmark_test.dart: Efficiency Test
cartButton: 3
userSwitch: 4
00:02 +1: All tests passed!
```

KOMPLEXITÄT / WARTBARKEIT Die Auswertung der Metriken (vgl. Abschnitt A.5) ergab ein MI von 80 für das gesamte Projekt.

VERSTÄNDLICHKEIT / LESBARKEIT Zur Verständlichkeit und Lesbarkeit lassen sich mehrere Aspekte in die Bewertung miteinbeziehen.

Im Gegensatz zu Provider wird bei Riverpod eine neue Syntax zur Injizierung von Providern verwendet. Diese weicht leicht von der in Flutter gebräuchlichen Methodik, die in Listing 5.9 gezeigt wurde, ab. So muss wie bereits beschrieben die Basisklasse von zugreifenden Widgets geändert werden und über die Referenzvariable `ref` auf die Zustände zugegriffen werden. Ob dies jedoch schon ausreichend ist, um zu sagen, dass hier von Flutter abweichende Konzepte eingeführt werden, ist fraglich. Allerdings wird durch die Verwendung der Referenzvariable die Lesbarkeit vereinfacht, da so der Aufruf von `Provider.of<XY>()` entfallen kann.

Die Verwendung von Variablen anstatt Generics zum Abruf der Provider, erhöht die Verständlichkeit, da somit direkt aus dem Variabel-Namen klar wird, um was es sich handelt. Zudem besteht somit die Möglichkeit, mehere Provider mit dem gleichen Datentyp zu haben. Damit kann dann auf Wrapper-Klassen, verzichtet werden, die die Komplexität und damit die Lesbarkeit behindern.

Das neu eingeführte Konzept des StateNotifier bietet aus der Perspektive der Lesbarkeit und Verständlichkeit keine Vorteile, da damit die Zustandsänderungs-Funktionen und der Inhalt des Zustands in getrennten Klassen behandelt werden, zudem muss zur Änderung des Zustands immer eine neue Instanz des Zustandsobjekts `state` gesetzt werden, was auf den ersten Blick verwirrend wirken kann. Die Vorteile des StateNotifier liegen eher in der Verbesserung der Performance, da hier bei jeder Zustandsänderung tatsächlich geprüft wird, ob die Zustandsvariable selbst sich tatsächlich geändert hat.

Die Möglichkeit andere Zustände über die Referenzvariable zu injizieren hingegen, verbessert die Lesbarkeit, da hier im Vergleich zu ProxyProvi-

der sofort klar wird, warum dieser Provider gelesen oder abonniert werden muss.

Das Problem der tiefen Verschachtlung umgeht Riverpod damit, dass die Provider nicht in den Widget-Baum eingesetzt werden müssen.

Da es neben dem StateNotifier auch die Möglichkeit gibt, wie bei Provider, ChangeNotifier einzusetzen, fällt dieser beschriebe Kritikpunkt nicht stark ins Gewicht, weshalb hier eine Bewertung mit "vollständig erfüllt" erfolgt.

DOKUMENTIERUNG Die Dokumentation [Rou22a] von Riverpod beschreibt alle Grundkonzepte und beinhaltet diverse Anwendungsbeispiele. Zusätzlich ist ebenfalls dokumentiert, wie automatisierte Tests implementiert werden können. Die Dokumentation ist auch in deutscher Sprache verfügbar.

Die Dokumentierung wird mit "vollständig erfüllt" bewertet.

STRUKTURBESTIMMUNG Ähnlich wie bei Provider lässt sich sagen, dass Riverpod keinen signifikanten Einfluss auf die Struktur der Anwendung hat. Daher kann dieser auch nicht technisch forciert werden.

Daher wird die Strukturbestimmung mit "nicht erfüllt" bewertet.

5.6 REDUX

Redux stellt eine externe Bibliothek zur Zustandsverwaltung dar, welche auf Ansätzen aus dem React-Umfeld basiert. Die Grundlagen dazu sind im Unterabschnitt 2.3.6 nachzuvollziehen.

5.6.1 *Implementierung*

Für die Implementierung mit Redux wurden die Bibliotheken redux in der Version 5.0.0 und flutter_redux in der Version 0.8.2 genutzt.

Anstelle von mehreren Zuständen wurde hier ein zentraler Zustand implementiert, der auch als Store bezeichnet wird. Um Änderungen an diesem Store durchzuführen wurden diverse Actions definiert wie beispielsweise eine Action für den Anmeldevorgang. Diese Actions werden von Reducern verarbeitet, welche in diesem Fall immer einen Teilzustand des zentralen Zustands verändern.

Für den Abruf der Daten für die Produktliste wurde eine Middleware implementiert. Diese erhält von der Bibliothek alle Actions, bevor diese von einem Reducer bearbeitet werden. Die Middleware führt im Falle, dass eine entsprechende Action ausgelöst wird, die Netzwerkzugriffe durch und emittiert danach entsprechende Actions, welche durch Reducer die Daten in den Store übermitteln. Middlewares sind notwendig, da Reducer keine asynchronen Operationen durchführen sollen, da diese keine Nebeneffekte haben dürfen und bei gleicher Eingabe immer das gleiche Ergebnis liefern müssen. [vgl. GF18, Kap1.3.3]

Für die Einbindung in die Benutzeroberfläche kommen zwei verschiedene Möglichkeiten zum Einsatz. Zum einen existiert der StoreConnector, welcher einen bestimmten Teil des zentralen Zustands selektiert und diesen an ein Widget weitergibt. Zum anderen gibt es den StoreBuilder, welcher den kompletten Store an ein Widget weitergibt. Der StoreConnector wird hauptsächlich bei trivialeren Einsatzgebieten verwendet, die nur das Auslesen einer Variable aus dem Store benötigen. Der StoreConnector hingegen bei komplexeren Fällen, die beispielsweise auch Actions depeschieren müssen.

5.6.2 *Bewertung*

Im folgenden Abschnitt wird die Implementierung mit Redux [Fra22, branch=redux] anhand der definierten Bewertungskriterien bewertet.

ÄNDERBARKEIT/SKALIERBARKEIT Die Änderbarkeit und Skalierbarkeit bei Redux lässt sich anhand verschiedener Aspekte beschreiben.

Ein Problem für die Skalierbarkeit kann der zentrale Zustand werden. Dieser besteht aus einer Klasse, die den gesamten Zustand der App abbilden soll. Bei großen Apps, könnte dieses Konzept an seine Grenzen kommen, besonders, wenn eine Anwendung aus mehreren Modulen besteht.

Die Untersuchung der Verknüpfung mehrere Zustände ist für Redux nicht anwendbar, da hier nur ein zentraler Zustand verwendet wird. Allerdings muss man sagen, dass dies auch zur Folge hat, dass Reducer viele Aufgaben übernehmen müssen, die nicht auf den ersten Blick offensichtlich sind, und aus der fehlenden Möglichkeit resultieren, mehrere Zustände zu haben, die voneinander abhängen.

Um die Reducer besser strukturieren zu können, bietet die Bibliothek, Reducer eines Teilaspektes des Zustands zusammenzufassen, womit diese nur ein Teilaspekt des Zustands tangiert.

Zusammenfassend wird die Skalierbarkeit und Änderbarkeit mit "nicht erfüllt" bewertet.

TESTBARKEIT Für die Testbarkeit werden zum einen Tests der Geschäftslogik und zum anderen Tests von Widgets, die auf den Store zugreifen bewertet.

Für die Tests der Geschäftslogik ist es ausreichend, den Store manuell zu erstellen und die gewünschte Action zu depeschieren. Anschließend kann anhand des zentralen Zustands geprüft werden, ob das gewünschte Ergebnis eingetreten ist.

Die Tests der Middlewares hingegen stellen sich als schwierig heraus, da hier asynchrone nicht-wartende Aktionen durchgeführt werden, womit es erforderlich wäre, auf die gewünschte Action zu warten, ohne zu wissen, ob diese tatsächlich jemals depeschiert wird.

Der zentrale Zustand sollte selbst keine Geschäftslogik beinhalten und lässt sich somit bei Widget-Tests ohne Probleme durch einen Platzhalter ersetzen.

Aufgrund der Probleme beim Testen der Middlewares, wird die Testbarkeit mit "teilweise erfüllt" bewertet.

EFFIZIENZ Nach der Ausführung der Teststrecke ergaben die Zähler folgendes Ergebnis:

Listing 5.12: Anzahl der Rendervorgänge bei Redux

```
00:00 +0: /app/test/efficiency_bechmark_test.dart: Efficiency Test
cartButton: 8
userSwitch: 10
00:02 +1: All tests passed!
```

KOMPLEXITÄT / WARTBARKEIT Die Auswertung der Metriken (vgl. Abschnitt A.6) ergab ein MI von 82 für das gesamte Projekt.

VERSTÄNDLICHKEIT / LESBARKEIT Die Verständlichkeit und Lesbarkeit wird anhand der definierten Aspekte bewertet.

Redux bringt mit dem aus dem React-Ökosystem stammenden Konzept ein völlig neues Konzept in das Flutter-Ökosystem. Begriffe wie Reducer, Action und Middlewares spielen außerhalb von Redux in Flutter sonst nur eine un-

tergeordnete Rolle. Somit müssen Entwickler*innen hier sich vor der Benutzung zuerst mit diesem Konzept beschäftigen, was für die Verständlichkeit nicht dienlich ist.

Auf der anderen Seite bietet Redux mit der klaren Aufteilung in Action, Reducers, Middlewares und State eine klare Struktur und macht den Quelltext aufgrund der relativ kleinen Methoden lesbar. Es mag allerdings bezweifelt werden, ob diese Lesbarkeit bei einem in der Theorie immer weiter wachsenden zentralen Store auf Dauer gewährleistet werden kann, da diese Klasse mit der Anwendung mitwachsen würde.

Das Problem der tiefen Verschachtlung findet bei Redux keine Anwendung, da hier auf einen Zustandsverwaltungsmechanismus außerhalb des Widget-Baums gesetzt wird und der Store lediglich an einer zentralen Stelle in den Widget-Baum injiziert wird.

Die Verständlich und Lesbarkeit wird abschließend mit "teilweise erfüllt" bewertet.

DOKUMENTIERUNG Die Dokumentation [Ega18] von Redux beschreibt die grundlegenden Konzepte, sowie verweist für detaillierte Erklärungen auf die Dokumentation von Redux für React. Zudem existieren in der Dokumentation detaillierte Anwendungsbeispiele. Zusätzlich zur offiziellen Dokumentation, gibt es eine große Zahl an Publikationen für Redux für React, die in großen Teilen anwendbar auf die Flutter-Umsetzung von Redux sind.

Die Dokumentierung wird daher mit "vollständig erfüllt" bewertet.

STRUKTURBESTIMMUNG Redux bestimmt die Struktur der Anwendung klar mit, da durch die feste Aufteilung in die bereits bestehenden Elemente wie Reducer oder Action, eine Struktur sich impliziert.

Technisch wird diese zusätzlich forciert, indem über die Bibliothek geprüft wird, ob beispielsweise die Reducer der geforderten Struktur entsprechen. Zusätzlich kann damit sichergestellt werden, dass Reducer immer ausschließlich synchrone Prozeduren durchführen.

Daher wird die Strukturbestimmung mit "vollständig erfüllt" bewertet.

5.7 MOBX

MobX ist ähnlich wie Redux eine Bibliothek, welche auf Ansätzen aus dem React-Umfeld basiert. Die Grundlagen dazu werden in Unterabschnitt 2.3.7 behandelt.

5.7.1 *Implementierung*

Für die Implementierung mit MobX werden die Bibliotheken mobx in der Version 2.0.6+1 und flutter_mobx in der Version 2.0.4 genutzt.

Für die Zustandsverwaltung wurden vier sogenannte Stores implementiert. Diese beinhalteten einen Teilzustand, sowie Computed-Values und Actions zum Ändern des Zustands. Dabei wurde jeweils ein Store für den Anmeldezustand, die Produktliste, den Warenkorb und die Anzahl eines Produktes im Warenkorb implementiert. Die Stores werden ergänzt durch automatisch generierten Quelltext von der MobX Bibliothek.

Die Stores werden über ein `InheritedWidget` in die Widget-Tree übergeben. Hierfür können auch beliebige andere Dependency-Injection-Lösungen genutzt werden. Auf Zustände wird innerhalb eines Observer-Widgets zugegriffen.

5.7.2 *Bewertung*

Im folgenden Abschnitt wird die Implementierung mit MobX [Fra22, branch=mobx] anhand der definierten Bewertungskriterien bewertet.

ÄNDERBARKEIT/SKALIERBARKEIT Zur Skalierbarkeit lässt sich sagen, dass hier keine Hindernisse bei der Skalierbarkeit festgestellt werden konnten. So lassen sich beliebig viele Zustände miteinander koppeln, da hier ein einfacher Zugriff auf Observable-Variablen genügt, um Zustandsaktualisierungen zu erhalten.

Zur Änderbarkeit lässt sich sagen, dass hier zwar keine Abstraktionsschicht wie bei Provider oder Riverpod existiert, aber spätere Änderungen sich durch die Verwendung von beispielsweise Computed-Values ausgleichen lässt und somit eine Rückwärtskompatibilität gewährleistet werden kann.

Aufgrund der einfachen Kopplung von Zuständen und der Änderbarkeit wird MobX hier mit "vollständig erfüllt" bewertet.

TESTBARKEIT Bei der Testbarkeit wird zum einen geprüft, inwiefern sich die Geschäftslogik testen lässt, und wie sich die Zustände bei Widget-Tests durch Platzhalter ersetzen lassen.

Die Tests der Geschäftslogik konnten ohne zusätzliche Maßnahmen implementiert werden, da nach außen hin hier eine Klasse mit Variablen und Funktionen existiert, die sich mit den Mitteln eines Unit-Test testen lässt.

Bei den Widget-Tests ist es nötig, die Store-Klasse mit neuen Platzhalter-Klassen, die von den Store-Klassen erben, zu ersetzen. Damit lassen sich erfolgreich Stores durch triviale Platzhalter ersetzen.

Die Testbarkeit wird mit "vollständig erfüllt" bewertet.

EFFIZIENZ Nach der Ausführung der Teststrecke ergaben die Zähler folgendes Ergebnis:

Listing 5.13: Anzahl der Rendervorgänge bei MobX

```
00:00 +0: /app/test/efficiency_bechmark_test.dart: Efficiency Test
cartButton: 4
userSwitch: 4
00:02 +1: All tests passed!
```

KOMPLEXITÄT / WARTBARKEIT Die Auswertung der Metriken (vgl. Abschnitt A.7) ergab ein MI von 83 für das gesamte Projekt.

VERSTÄNDLICHKEIT / LESBARKEIT Zur Verständlichkeit und Lesbarkeit lässt sich sagen, dass mit MobX hier ein Ansatz verfolgt wird, der zur Lesbarkeit beiträgt, da hier mithilfe der Annotationen wie `computed` der Sprachfluss beim Lesen unterstützt wird. Zusätzlich beinhalten die Zustandsklassen nur die tatsächlich notwendige Geschäftslogik und Prozeduren für die Zustandsverwaltung werden mittels Quelltext-Generierung ausgelagert. Somit ist die Struktur nachvollziehbar und klar.

Die Quelltext-Generierung kann allerdings auch ein Risiko darstellen, da Fehler oder Probleme im generierten Quelltext schwer nachzuvollziehen sind und der generierte Quelltext schwer lesbar ist.

Mit MobX verwendet hauptsächlich bestehende Konzepte aus dem Flutter-Ökosystem und erweitert diese lediglich um die bereits erwähnten Annotationen, welche allerdings bereits durch ihre Benennung Aufschluss über ihre Funktion geben.

Die tiefe Verschachtlung wird bei MobX verhindert, indem hier zur Zustandsverwaltung kein Widget-basierter Ansatz verwendet wird, und somit keine Injizierung in den Widget-Baum nötig ist.

Die Verständlichkeit und Lesbarkeit wird mit "vollständig erfüllt" bewertet, allerdings sollte das Risko der Quelltext-Generierung bei einer Entscheidung miteinbezogen werden.

DOKUMENTIERUNG Die Dokumentation [Pod21] von MobX beschreibt die Grundkonzepte des Zustandsverwaltungssystems und ergänzt diese ebenfalls mit umfangreichen Anwendungsbeispielen. Zusätzlich zur offiziellen Dokumentation existieren noch diverse Drittveröffentlichungen zu MobX und die Dokumentation zur JavaScript-Variante von MobX kann ebenfalls in Teilen für Dart adaptiert werden.

Die Dokumentierung wird daher mit "vollständig erfüllt" bewertet.

STRUKTURBESTIMMUNG Zur Strukturbestimmung lässt sich sagen, dass MobX über die Annotationen und Quelltext-Generierung eine gewisse Struktur der Zustände vorgibt, allerdings leitet sich daraus keine Struktur für die gesamte Anwendung ab.

Technisch forciert, wird die Struktur der Zustände in Teilen durch die Quelltext-Generierung, wenn von Vorgaben abgewichen wird. So schlägt beispielsweise die Quelltext-Generierung fehl, wenn man versucht, Funktionen mit Parametern als `computed` zu markieren.

Die Strukturbestimmung wird daher mit "teilweise erfüllt" bewertet.

5.8 ÜBERSICHT

Die Ergebnisse der Evaluation sind in der Tabelle 5.1 übersichtlich zusammengestellt. Als Vergleichsgröße wurden zusätzlich die Ausgangsmesswerte ohne Zustandsverwaltungssystem in der ersten Zeile ergänzt. Die Ergebnisse werden im nachfolgenden Fazit-Kapitel näher gedeutet und analysiert.

Tabelle 5.1: Ergebnisse der Evaluation

Zustandsverwaltung	Änderbarkeit/Skalierbarkeit	Testbarkeit	Effizienz (in Rendervorgängen)	Komplexität/Wartbarkeit (in MI)	Verständlichkeit/Lesbarkeit	Dokumentierung	Strukturbestimmung
ohne Zustandsverwaltung	n. a.	n. a.	1;2	83	n. a.	n. a.	n.a.
setState	nicht umsetzbar						
InheritedWidget	/	/	8;6	83	✗	/	✗
BLoC	✓	✓	8;4	82	✗	✓	/
Provider	/	✓	3;4	83	/	✓	✗
Riverpod	✓	✓	3;4	80	✓	✓	✗
Redux	✗	/	8;10	82	/	✓	✓
MobX	✓	✓	4;4	83	✓	✓	/

Legende: ✓=vollständig erfüllt; /=teilweise erfüllt; ✗=nicht erfüllt;
n.a. = nicht anwendbar

6

FAZIT

Abschließend werden die Ergebnisse der Evaluation nochmals zusammengefasst und analysiert, inwiefern mit der Evaluation das Ziel der Ausarbeitung erreicht werden konnte. Darauf aufbauend wird eine Empfehlung für verschiedene Anwendungsfälle gegeben, gefolgt von einem Ausblick, welcher Aspekte der Ausarbeitung in Zukunft noch erweitert werden könnten.

Mit der Evaluation konnten bis auf das setState-Konzept alle Zustandsverwaltungssysteme evaluiert werden. Die Bewertungskriterien haben sich dabei größtenteils als aussagekräftig erwiesen. Lediglich der Maintainability Index konnte keine wirklich aussagekräftigen Ergebnisse bieten, da die Ergebnisse alle sehr nah aneinander liegen und so keine Differenzierung zwischen den einzelnen Systemen erfolgt ist. Die Metrik der Effizienz hingegen zeigt relativ aufschlussreich, welche Systeme effizient mit dem Neu-Erstellen von Widgets umgehen und welche nicht. Die qualitativen Bewertungen können zusätzlich einen detaillierten Einblick in die Umsetzung verschiedener Zustandssysteme bieten.

Bei den Zustandsverwaltungssystemen selbst haben sich in der Evaluation besonders die Bibliotheken Riverpod und MobX mit guten Ergebnissen in den qualitativen Bewertungen sowohl in der quantitativen Bewertung der Effizienz gezeigt. Riverpod bietet dabei mehr Freiräume bei der Strukturgestaltung der Zustände während MobX hier eine klare Struktur vorgibt.

Somit hat sich auch gezeigt, dass Riverpod eine sinnvolle Weiterentwicklung von Provider ist, welches in der Evaluation mehrere Schwachstellen wie beispielsweise in der Skalierbarkeit aufwies.

BLoC und `InheritedWidget` konnten zwar die Anforderungen an die Beispielanwendung umsetzen, erwiesen sich allerdings ohne besondere Optimierung als wenig effizient und hatten Einschränkungen in der Lesbarkeit und Verständlichkeit.

ReduX als besonders im React-Ökosystem oft verwendeter Ansatz zur Zustandsverwaltung, konnte auf Flutter nicht überzeugen, da hier besonders der zentrale Zustand zu einem Problem wird, da dieser für die in der Evaluation schlechtesten Effizienz-Werte und eine negative Bewertung bei der Skalierbarkeit verantwortlich ist.

Zusammenfassend lässt sich sagen, dass durch die Evaluation ein umfassender Überblick über die Funktionsweise der verschiedenen Zustandsverwaltungssysteme gegeben werden konnte, und eine in großen Teilen aussagekräftige Bewertung der Zustandssysteme vorgenommen werden konnte. Zudem konnte Anforderungen und Probleme bei der Zustandsverwaltung in Flutter entwickelt und dargestellt werden.

6.1 EMPFEHLUNGEN

Für große Anwendungen mit verschiedenen Komponenten und vielen Zuständen, wird MobX empfohlen, da hier besonders auch im Vergleich zu Riverpod eine bessere Strukturbestimmung der Anwendung gewährleistet werden kann und die Bibliothek auch die Verwaltung und Koppelung großer Mengen an Zuständen ermöglichen kann.

Für mittelgroße Anwendungen, empfiehlt sich die Nutzung von Riverpod, da hier besonders die Vielseitigkeit der Bibliothek die Integration von verschiedenen Ansätzen erleichtert und das Merkmal der Strukturbestimmung hier eine nicht so große Relevanz haben dürfte.

Für kleine Anwendungen empfiehlt sich entweder die Nutzung von InheritedWidgets, wenn beispielsweise auf die Einbindung weitere Bibliotheken verzichtet werden kann oder die Verwendung von Provider, da hier die Probleme mit der Skalierbarkeit und Lesbarkeit von ProxyProvidern nicht stark ins Gewicht fallen sollten.

6.2 AUSBLICK

Für weitergehende Untersuchungen bieten sich mehrere Aspekte an. Zum einen ist die Ersetzung der MI-Metrik durch eine aussagekräftigere Metrik empfehlenswert, um so auch die Anforderung der Komplexität / Wartbarkeit in der Evaluation aussagekräftig bewerten zu können. Zum anderen bietet sich die Erweiterung der Evaluation um andere, alternative Zustandsverwaltungssysteme, die in dieser Ausarbeitung nicht untersucht werden konnten (vgl. Unterabschnitt 2.3.1), um ein vollständigeres Bild über die Zustandsverwaltungssysteme für Flutter zu bekommen. Ein weiterer untersuchenswerter Aspekt könnte der Vergleich mit der Zustandsverwaltung von React sein, um mögliche Unterschiede darzustellen.

Teil II

APPENDIX

A

TESTERGEBNISSE

A.1 METRIKEN: MAIN

Directory	Cyclomatic complexity	Source lines of code	Maintainability index	Number of Arguments	Maximum Nesting	Technical Debt
lib	4	17	68	1	1	0.0
lib/models	5	3	98	0	0	0.0
lib/screens/cart	7	43	80	1	1	0.0
lib/screens/product_list	10	36	84	1	1	0.0
lib/service	2	8	83	0	1	0.0
lib/utils	1	2	84	1	0	0.0
lib/widgets	7	27	83	1	1	0.0

Cyclomatic complexity : 36

Source lines of code : 136

Maintainability index : 83

Number of Arguments : 1

Maximum Nesting : 1

Technical Debt : 0

A.2 METRIKEN: INHERITEDWIDGET

Directory	Cyclomatic complexity	Source lines of code	Maintainability index	Number of Arguments	Maximum Nesting	Technical Debt
lib	5	28	68	1	1	0.0
lib/models	16	12	97	0	1	0.0
lib/screens/cart	7	45	80	1	1	0.0
lib/screens/product_list	15	54	83	1	1	0.0
lib/service	2	8	83	0	1	0.0
lib/stores	35	100	85	1	1	0.0
lib/utils	1	2	84	1	0	0.0
lib/widgets	7	28	83	1	1	0.0

Cyclomatic complexity : 88

Source lines of code : 277

Maintainability index : 83

Number of Arguments : 1

Maximum Nesting : 1

Technical Debt : 0

A.3 METRIKEN: BLOC

Directory	Cyclomatic complexity	Source lines of code	Maintainability index	Number of Arguments	Maximum Nesting	Technical Debt
lib	13	42	69	1	1	0.0
lib/bloc	59	117	87	0	1	0.0
lib/models	13	12	94	0	0	0.0
lib/screens/cart	7	61	77	1	1	0.0
lib/screens/product_list	15	76	77	1	2	0.0
lib/service	2	8	83	0	1	0.0
lib/utils	2	2	92	1	0	0.0
lib/widgets	8	36	80	1	1	0.0

Cyclomatic complexity : 119

Source lines of code : 354

Maintainability index : 82

Number of Arguments : 1

Maximum Nesting : 1

Technical Debt : 0

A.4 METRIKEN: PROVIDER

Directory	Cyclomatic complexity	Source lines of code	Maintainability index	Number of Arguments	Maximum Nesting	Technical Debt
lib	4	25	68	1	1	0.0
lib/models	17	10	96	0	0	0.0
lib/screens/cart	7	47	79	1	1	0.0
lib/screens/product_list	14	53	82	1	1	0.0
lib/service	2	8	83	0	1	0.0
lib/store	27	69	89	1	1	0.0
lib/utils	1	2	84	1	0	0.0
lib/widgets	7	28	82	1	1	0.0

Cyclomatic complexity : 79

Source lines of code : 242

Maintainability index : 83

Number of Arguments : 1

Maximum Nesting : 1

Technical Debt : 0

A.5 METRIKEN: RIVERPOD

Directory	Cyclomatic complexity	Source lines of code	Maintainability index	Number of Arguments	Maximum Nesting	Technical Debt
lib	4	22	68	1	1	0.0
lib/models	6	4	97	0	0	0.0
lib/screens/cart	7	44	80	1	1	0.0
lib/screens/product_list	11	49	81	1	1	0.0
lib/service	2	8	83	0	1	0.0
lib/stores	11	16	66	1	1	0.0
lib/utils	1	2	84	1	0	0.0
lib/widgets	7	28	82	1	1	0.0

Cyclomatic complexity : 49

Source lines of code : 173

Maintainability index : 80

Number of Arguments : 1

Maximum Nesting : 1

Technical Debt : 0

A.6 METRIKEN: REDUX

Directory	Cyclomatic complexity	Source lines of code	Maintainability index	Number of Arguments	Maximum Nesting	Technical Debt
lib	4	20	68	1	1	0.0
lib/models	21	14	96	0	0	0.0
lib/redux	12	32	75	1	1	0.0
lib/redux/actions	8	1	100	0	0	0.0
lib/redux/middlewares	3	11	81	2	2	0.0
lib/redux/reducers	11	26	84	2	1	0.0
lib/screens/cart	7	52	78	1	1	0.0
lib/screens/product_list	11	54	79	1	1	0.0
lib/service	2	8	83	0	1	0.0
lib/utils	1	2	84	1	0	0.0
lib/widgets	8	33	81	1	1	0.0

Cyclomatic complexity : 88

Source lines of code : 253

Maintainability index : 82

Number of Arguments : 1

Maximum Nesting : 1

Technical Debt : 0

A.7 METRIKEN: MOBX

Directory	Cyclomatic complexity	Source lines of code	Maintainability index	Number of Arguments	Maximum Nesting	Technical Debt
lib	4	25	68	1	1	0.0
lib/models	6	4	97	0	0	0.0
lib/screens/cart	7	52	78	1	1	0.0
lib/screens/product_list	11	58	79	1	1	0.0
lib/service	2	8	83	0	1	0.0
lib/stores	26	48	94	1	1	0.0
lib/utils	1	2	84	1	0	0.0
lib/widgets	8	33	81	1	1	0.0

Cyclomatic complexity : 65

Source lines of code : 230

Maintainability index : 83

Number of Arguments : 1

Maximum Nesting : 1

Technical Debt : 0

LITERATUR

[Ama18] Ron Amadeo. "Google starts a push for cross-platform app development with Flutter SDK". In: *Ars Technica* (Feb. 2018). URL: https://arstechnica.com/gadgets/2018/02/google-starts-a-push-for-cross-platform-app-development-with-flutter-sdk/ (besucht am 12. 01. 2022).

[AA21] Dennis Andersson und Axel Axelsson. "Evaluation of The Software Development Process for A Multi-Platform Solution in Flutter". Bachelor Thesis. 2021. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:hj:diva-54302 (besucht am 25. 02. 2022).

[Ang21] Felix Angelov. *Core Concepts (package:bloc)*. 2021. URL: https://bloclibrary.dev/#/coreconcepts (besucht am 21. 02. 2022).

[App22] Appfigures. *The Most Popular Development SDKs*. 2022. URL: https://appfigures.com/top-sdks/development/all (besucht am 12. 01. 2022).

[Ars21] Waleed Arshad. *Managing State in Flutter Pragmatically*. Packt Publishing, Nov. 2021. 246 S. ISBN: 1801070776. URL: https://www.ebook.de/de/product/41926143/waleed_arshad_managing_state_in_flutter_pragmatically.html.

[Bal09] Helmut Balzert. "Anforderungen und Anforderungsarten". In: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Heidelberg: Spektrum Akademischer Verlag, 2009, S. 455–474. ISBN: 978-3-8274-2247-7. DOI: 10.1007/978-3-8274-2247-7_16. URL: https://doi.org/10.1007/978-3-8274-2247-7_16.

[Bra17] Chris Bracken. *v0.0.6: Rev alpha branch version to 0.0.6, flutter 0.0.26*. 2017. URL: https://github.com/flutter/flutter/releases/tag/v0.0.6 (besucht am 04. 01. 2022).

[Ega18] Brian Egan. *Redux.dart Basics*. 2018. URL: https://github.com/fluttercommunity/redux.dart/blob/master/doc/basics.mdhttps://github.com/fluttercommunity/redux.dart/blob/f7590336d1b564aa05f7bf7bf87e6ac7c0d30f75/doc/basics.md (besucht am 25. 02. 2022).

[Fau20] Sebastian Faust. "Using Google´s Flutter Framework for the Development of a Large-Scale Reference Application". en. Bachelor Thesis. Technische Hochschule Köln, 2020, S. 85. URL: https://nbn-resolving.org/urn:nbn:de:hbz:832-epub4-14989.

[Fra22] Jonas Franz. *thesis_shop: An example application to evaluate state management in Flutter*. 2022. URL: https://git.jonasfranz.software/KoSI/thesis_shop.

[GF18] Marc Garreau und Will Faurot. *Redux in action*. Shelter Island, NY: Manning Publications Co, 2018. ISBN: 9781617294976.

[Goo] Google LLC. *Liste der Flutter-Packete auf pub.dev sortiert nach Anzahl der Likes*. URL: https://pub.dev/packages?q=sdk%3Aflutter&sort=like (besucht am 25. 02. 2022).

[Goo21a] Google LLC. *Architectural overview*. 2021. URL: https://docs.flutter.dev/resources/architectural-overview (besucht am 12. 01. 2022).

[Goo21b] Google LLC. *ChangeNotifier class - foundation library - Dart API*. 2021. URL: https://api.flutter.dev/flutter/foundation/ChangeNotifier-class.html (besucht am 27. 01. 2022).

[Goo21c] Google LLC. *InheritedModel class - widgets library - Dart API*. 2021. URL: https://api.flutter.dev/flutter/widgets/InheritedModel-class.html (besucht am 23. 01. 2022).

[Goo21d] Google LLC. *List of state management approaches*. 2021. URL: https://docs.flutter.dev/development/data-and-backend/state-mgmt/options (besucht am 07. 01. 2022).

[Goo21e] Google LLC. *Testing Flutter apps*. 2021. URL: https://docs.flutter.dev/testing (besucht am 25. 02. 2022).

[Goo21f] Google LLC. *The Dart type system*. 2021. URL: https://dart.dev/guides/language/type-system (besucht am 12. 01. 2022).

[Goo22a] Google LLC. *Supported platforms*. 2022. URL: https://docs.flutter.dev/development/tools/sdk/release-notes/supported-platforms (besucht am 04. 01. 2022).

[Goo22b] Google LLC. *pub.dev Entry of Provider*. 2022. URL: https://pub.dev/packages/provider (besucht am 25. 02. 2022).

[Gre+21] Benjamin L Greenberg, Spencer L Howell, Tucker R Miles, Vicki Tang und Daniel N Troutman. "Attendio: Attendance Tracking Made Simple". In: *Chancellor's Honors Program Projects* (2021). URL: https://trace.tennessee.edu/utk_chanhonoproj/2432 (besucht am 30. 01. 2022).

[Kru21] Dmitry Krutskikh. *Maintainability Index - Dart Code Metrics*. 2021. URL: https://dartcodemetrics.dev/docs/metrics/maintainability-index (besucht am 11. 01. 2022).

[KIG20] Nikita Kuzmin, Konstantin Ignatiev und Denis Grafov. "Experience of Developing a Mobile Application Using Flutter". In: *Information Science and Applications*. Hrsg. von Kuinam J. Kim und Hye-Young Kim. Singapore: Springer Singapore, 2020, S. 571–575. ISBN: 978-981-15-1465-4.

[Mez18] Luca Mezzalira. "MobX: Simple State Management". In: *Front-End Reactive Architectures: Explore the Future of the Front-End using Reactive JavaScript Frameworks and Libraries*. Berkeley, CA: Apress, 2018, S. 129–158. ISBN: 978-1-4842-3180-7. DOI: 10.1007/978-1-4842-3180-7_5. URL: https://doi.org/10.1007/978-1-4842-3180-7_5.

[Osd] *Mobile Operating System Market Share Worldwide*. Dez. 2021. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (besucht am 25. 12. 2021).

[Pod21] Pavan Podila. *Core Concepts - mobx.dart*. 2021. URL: https://mobx.netlify.app/concepts (besucht am 25. 02. 2022).

[RH97] Linda H Rosenberg und Lawrence E Hyatt. "Software quality metrics for object-oriented environments". In: *Crosstalk journal* 10.4 (1997), S. 1–6. URL: http://people.ucalgary.ca/~far/Lectures/SENG421/PDF/oocross.pdf (besucht am 08. 02. 2022).

[Rou22a] Remi Rousselet. *Getting Started - Riverpod*. 2022. URL: https://riverpod.dev/de/docs/getting_started/ (besucht am 24. 02. 2022).

[Rou22b] Remi Rousselet. *Provider's README.md*. 2022. URL: https://github.com/rrousselGit/provider/blob/eac827630a5f330c86857e4e13113aacdca759bc/README.md (besucht am 22. 02. 2022).

[Sle20] Dmitrii Slepnev. "State management approaches in Flutter". en. Bachelor Thesis. South-Eastern Finland University of Applied Sciences, 2020. URL: https://nbn-resolving.org/urn:nbn:de:hbz:832-epub4-14989.

[Soa18] Paolo Soares. "Flutter / AngularDart – Code sharing, better together". In: *DartConf 2018*. 2018. URL: https://www.youtube.com/watch?v=PLHln7wHgPE (besucht am 23. 01. 2022).

[WOA97] Kurt D. Welker, Paul W. Oman und Gerald G. Atkinson. "Development and Application of an Automated Source Code Maintainability Index". In: *Journal of Software Maintenance: Research and Practice* 9.3 (1997), S. 127–159. DOI: https://doi.org/10.1002/(SICI)1096-908X(199705)9:3<127::AID-SMR149>3.0.CO;2-S. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291096-908X%28199705%299%3A3%3C127%3A%3AAID-SMR149%3E3.0.CO%3B2-S. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-908X%28199705%299%3A3%3C127%3A%3AAID-SMR149%3E3.0.CO%3B2-S.

[Win20] Eric Windmill. *Flutter in Action*. New York, Vereinigte Staaten von Amerika: Manning Publications, 2020. ISBN: 9781617296147.

[Wu18] Wenhao Wu. "React Native vs Flutter, cross-platform mobileapplication frameworks". Bachelorthesis. Metropolia University of Applied Sciences, März 2018.