HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Hannes Steiner

Konzeption und Entwicklung einer Software-Architektur für Augmented-Reality-Anwendungen in Kombination mit einer Smartwatch als Benutzeroberflächen-Schnittstelle

Ein Ausblick auf zukünftige Augmented-Reality-Brille-Anwendungen

2020

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Konzeption und Entwicklung einer Software-Architektur für Augmented-Reality-Anwendungen in Kombination mit einer Smartwatch als Benutzeroberflächen-Schnittstelle

Ein Ausblick auf zukünftige Augmented-Reality-Brille-Anwendungen

Autor:
Hannes Steiner

Studiengang:
Softwareentwicklung

Seminargruppe:
IF17wS-B

Matrikelnummer:
46540

Erstprüfer:
Prof. Dr. Marc Ritter

Zweitprüfer:
Dipl.-Inf. Holger Langner

Mittweida, September 2020

Bibliografische Angaben

Steiner, Hannes: Konzeption und Entwicklung einer Software-Architektur für Augmented-Reality-Anwendungen in Kombination mit einer Smartwatch als Benutzeroberflächen-Schnittstelle, Ein Ausblick auf zukünftige Augmented-Reality-Brille-Anwendungen, 91 Seiten, 21 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Bachelorarbeit, 2020

Englischer Titel

Conception and development of a software architecture for augmented reality applications in combination with a smartwatch-based user interface

Referat

Ziel dieser Arbeit ist es, ein Software-Architektur-Konzept zu entwickeln, umzusetzen und zu evaluieren, dessen Fokus auf die Kombination einer Smartwatch-Anwendung mit einer AR-Brille-Anwendung gerichtet ist. Dazu wurde ein Patent untersucht, dessen Schwerpunkt auf der Kombination von AR-Brille und Smartwatch liegt, um Anforderungen an solch eine Software-Architektur zu erstellen. Im Anschluss entstand unter Einbeziehung der recherchierten Informationen ein Architektur-Konzept. Für die darauffolgende Umsetzung wurden Anwendungsfälle entwickelt, die zeigen sollen, wie ausgewählte Software-Qualitätsmerkmale bei Verwendung der Architektur erreicht werden können. Bei der anschließenden Evaluation zeigte sich, dass die Architektur zur Erfüllung wichtiger Software-Qualitätsmerkmale beiträgt, aber auch Optimierungspotenzial besitzt.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Quellcodeverzeichnis

V. Abkürzungsverzeichnis

aGPS ..... Assisted Global Positioning System, Deutsch: Unterstütztes Globales Positionierungssystem, Seite 4

AR ..... Augmented Reality, auf Deutsch: Erweiterte Realität, Seite 1

AV ..... Augmented Virtuality, Deutsch: Erweiterte Virtualität, Seite 3

CPU ..... Central processing unit, Deutsch: Zentrale Verarbeitungseinheit oder Prozessor, Seite 13

CV ..... Computer vision, Deutsch: Computer Sicht oder computergestützte Bildverarbeitung, Seite 6

fps ..... Frames per second, Deutsch: Bilder pro Sekunde, Einheit für die Bildfrequenz, Seite 34

FRP ..... Funktionale reaktive Programmierung auch als Datenflussprogrammierung bekannt, Seite 18

GPU ..... Graphics processing unit, auf Deutsch: Grafikprozessor, Seite 33

HCI ..... Human-Computer Interaction, Deutsch: Mensch-Computer-Interaktion, Seite 1

HMDs ..... Head-Mounted Displays, Deutsch: Am Kopf montierte Displays und im deutschen Synonym für AR-Brillen, aber auch VR-Brillen, Seite 3

IETF ..... Internet Engineering Task Force, auf Deutsch: Internettechnik-Arbeitsgruppe, Seite 10

IoT ..... Internet of Things, Deutsch: Internet der Dinge, Seite 73

MB ..... Megabyte, Maßeinheit der Digitaltechnik, Seite 67

MVVM ..... Model-View-ViewModel, Seite 17

NFC ..... Near Field Communication, Deutsch: Nahfeldkommunikation, Seite 10

RAM ..... Random access memory, Deutsch: Speicher mit wahlfreiem/direktem Zugriff oder Direktzugriffsspeicher, Seite 33

SDK ..... Software development kit, Deutsch: Softwareentwicklungsbaukasten, Seite 34

TCA ..... The Composable Architecture, Deutsch: Die zusammensetzbare Architektur, Seite 42

VR ..... Virtual Reality, Deutsch: Virtuelle Realität, Seite 3

1 Einführung

1.1 Motivation

Augmented Reality (AR) erzeugt die Illusion, dass virtuelle Objekte in einer realen Welt platziert sind. Heute kann fast jede Smartphone-Linse als Tor in die virtuelle Welt dienen. Basierend auf dem, was die Kamera sieht, wird mit AR die reale Welt durch Informationen erweitert. Doch neben den Hosentaschenrechnern gibt es auch andere Möglichkeiten, in die Augmented Reality einzutauchen. Beispielsweise Augmented-Reality-Brillen - kurz AR-Brillen. Sie sind eine aufstrebende Technologie für Verbraucher, die so unterschiedliche und grundlegende Bereiche wie Bildung [24], Barrierefreiheit [15], Gesundheitswesen [37], Logistik [28] und Unterhaltung [34] zu beeinflussen verspricht. Verständlicherweise - da AR in der Forschung zur Mensch-Computer-Interaktion (HCI) seit Langem Aufmerksamkeit erregt [4]. Während Aspekte wie Tracking-Genauigkeit, Anzeigequalität und Rechnerleistung im Hinblick auf die heutige High-End-Produkte erheblich weiterentwickelt wurden, sind jedoch die meisten Eingabe- und Interaktionstechnologien weniger ausgereift. Aktuelle kommerzielle Systeme verfügen über On-Headset-Touch-Oberflächen (z. B. Google Glass) oder Handsteuerungen in Form von Touchpads (z. B. Epson BT-300) oder Handcontrollern (z.B. Microsoft HoloLens) als wichtige Interaktionskanäle. Diese Systeme können zwar effektiv sein, bieten aber begrenzte Eingabebereiche und sind im Falle von Handheld-Controllern lästige Zusatzgeräte, welche die Systemnutzung stören oder ausschließen.

In Anbetracht des Bedarfs an Eingabesystemen für Augmented-Reality-Brillen, welche die Hände frei lassen, hat sich eine beträchtliche Anzahl von wissenschaftlichen Forschungen mit diesen Themen befasst, die von tragbaren Geräten wie Gürteln [7] oder Ringen [8] bis hin zu gestischen Eingaben in der Luft [14] (z.B. Microsoft HoloLens 2) und Berührungen am Gesicht [19] oder vielmehr Körper [36] reichen. Allerdings können vor allem die derzeitigen Methoden der Gesteneingabe für AR-Brillen sozial irritierend oder nicht intuitiv sein. Beispielsweise kann ein Dritter die virtuellen Bilder, die von einem Benutzer mit AR-Brille beobachtet werden, in der Regel nicht sehen. Ein solcher Dritter kann sich irritiert oder gestört fühlen, wenn er einen AR-Brillenträger erblickt, der ohne ersichtlichen Grund in der Luft wischt, drückt oder andere Gesten macht. Deshalb hat der Erfinder Jonathan M. Rodriguez II. seine Idee einer armbanduhrbasierten Benutzeroberfläche für Augmented-Reality-Brillen patentieren lassen [26].

Das Patent beschreibt Systeme und Methoden zur Bereitstellung einer armbanduhrbasierten Benutzeroberfläche für Augmented Reality Brillen. Die Grundidee dabei ist es, Armbanduhr bzw. Smartwatch und eine AR-Brille zu kombinieren, um eine gesellschaftlich akzeptable und intuitive Methode der gestischen Eingabe für Augmented-Reality-Brillen zu realisieren. Beispielsweise wird der Bildschirm der Smartwatch durch

eine AR-Benutzeroberfläche erweitert, sodass ein fließender Übergang der grafischen Benutzeroberflächen entsteht. Auch umgekehrt soll die Benutzeroberfläche der Smartwatch die AR-Oberfläche durch zusätzliche Informationen anreichern. In dem Patent bleibt jedoch größtenteils offen, wie die Software-Architektur solch einer Anwendung aussehen könnte.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, ein Anwendungs-/ Software-Architektur-Konzept zu entwickeln, umzusetzen und zu evaluieren, dessen Fokus auf die Kombination einer Smartwatch-Anwendung mit einer AR-Brille gerichtet ist.

Speziell soll die Architektur einen fließenden Übergang der Benutzeroberflächen sowie diverse Eingabemöglichkeiten auf den Geräten unterstützen. Zu diesem Zweck wird das Patent auf Hinweise zur Funktionsweise und Struktur der Architektur untersucht. Aus den gesammelten Informationen soll ein passendes Architektur-Konzept entwickelt werden, welches im Anschluss mit verschiedenen Anwendungsfällen in einem lauffähigen Demonstrator umgesetzt wird. Dadurch soll aufgezeigt werden, wie die Architektur anwendungsspezifisch genutzt werden kann. Abschließend wird herausgearbeitet, auf welche Weise die Architektur eine Umsetzung spezifischer Softwarequalitätskriterien unterstützt. Damit soll diese Arbeit als Leitfaden für zukünftige Entwickler, welche die Architektur verwenden, dienen.

2 Grundlagen Augmented Reality

Im folgenden Kapitel werden ausschließlich technische Grundlagen zu Augmented Reality vermittelt, da diese im anschließenden Kapitel 3 benötigt werden. Softwaretechnische Grundlagen der Architektur sind erst im Kapitel 5 erläutert.

Unter einer komplett virtuellen Realität, besser bekannt unter dem Namen Virtual Reality (VR), ist die Darstellung und Wahrnehmung einer in Echtzeit computergenerierten interaktiven Umgebung mit eigenen physikalischen Eigenschaften zu verstehen. Im Gegensatz dazu bedeutet Augmented Reality (AR), auf Deutsch Erweiterte Realität, die digitale Anreicherung der Realität mit zusätzlichen sensorischen Informationen. Der Übergang von der Realität zur Virtual Reality ist im Mixed-Reality-Kontinuum [21] von Paul Milram und Fumio Kishino aus dem Jahr 1994 dargestellt. Wie in Abbildung 2.1 zu sehen, steht AR der realen Welt nahe, da die Realität lediglich mit Informationen angereichert wird. Ganz rechts dagegen steht die Virtual Reality. Dazwischen befindet sich die eher weniger bekannte Augmented Virtuality (AV), die überwiegend computergeneriert ist und durch Anteile der realen Welt ergänzt wird.

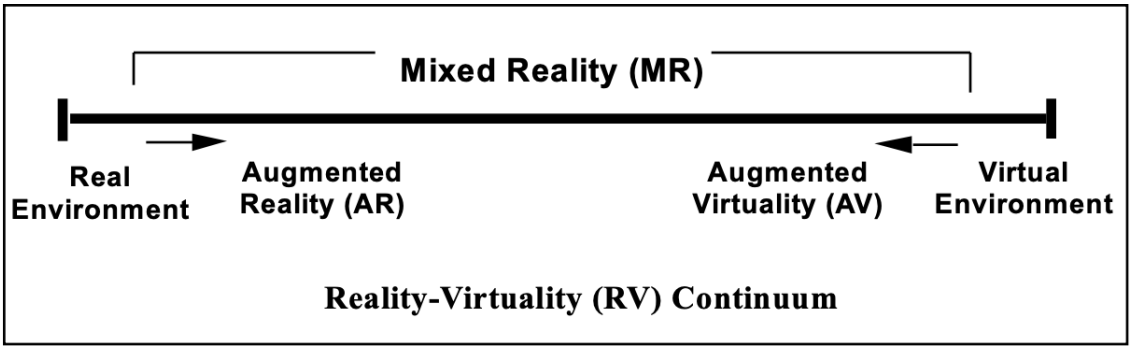


Abbildung 2.1: Mixed-Reality-Spektrum nach Milgram [21]

In der Literatur wird außerdem oft die Definition von Ronald Azuma [4] verwendet, die Augmented Reality Systeme durch die folgenden drei Charakteristika beschreibt:

1. AR ist eine Kombination von realen und virtuellen Elementen.
2. AR-Systeme sind in Echtzeit interaktiv.
3. Die Verbindung von virtueller und realer Information erfolgt dreidimensional.

Diese Definition erlaubt neben den sogenannten Head-Mounted Displays (HMDs), sprich AR-Brillen, auch andere Technologien unter Beibehaltung der wesentlichen Bestandteile von AR. Prinzipiell werden somit alle menschlichen Sinne angesprochen. Allgemein bezieht sich AR und auch diese Arbeit auf den Sehsinn oder vielmehr auf das Erweitern der Realität durch visuelle Informationen wie Texte, Bilder, Videos und 3D-Objekte. Die Darstellung von AR erfolgt dabei hauptsächlich über Displays von Smartphones und Tablets wie auch die vielversprechenden AR-Brillen.
Um Augmented Reality Anwendungen zu ermöglichen, ist es notwendig, zunächst die

reale Umgebung zu erfassen, um diese anschließend mit virtuellen Objekten zu ergänzen [20]. Damit die computergenerierten Bilder bei der Überlagerung mit einem realen Hintergrund echt aussehen, müssen diese Bilder exakt auf die reale Welt und auf die Perspektive des AR-Displays bzw. der Kamera des AR-Geräts ausgerichtet sein. Um die richtige Perspektive auf ein virtuelles Objekt zu gewährleisten, muss das System in der Lage sein, die Position und Ausrichtung des AR-Geräts in der realen Welt zu bestimmen. Ansonsten führt dies dazu, dass die virtuellen Objekte nicht an der gewünschten Stelle erscheinen oder die Orientierung und Blickrichtung der virtuellen Objekte bei Änderung des Blickwinkels nicht der realen Betrachtung entspricht. [17]

2.1 Tracking

Das Erfassen der realen Umgebung, aber auch das Verfolgen von Objekten der realen Umgebung, nennt man Tracking. Die Hard- und Software, die diese Aufgabe erfüllt, wird als Tracker bezeichnet. [20] Beispielsweise eignen sich Smartphones und Tablets durch ihre Vielzahl an eingebauten Sensoren für AR-Anwendungen. Im Allgemeinen wird zwischen zwei Tracking-Verfahren unterschieden, die in Kombination Augmented Reality ermöglichen: nichtvisuelles und visuelles Tracking. [20, 26, 27]

2.1.1 Nichtvisuelles Tracking

Bei nichtvisuellem Tracking wird die Position und/oder Orientierung eines Geräts mithilfe von Sensoren erfasst. [20, 27] Beispielsweise verfügen AR fähige Smartphones und Tablets über folgende Sensoren:

- **Magnetometer (Kompass):**
  Anhand des Magnetfelds der Erde kann die Orientierung des Smartphones relativ zur Erdachse bestimmt werden. [27]
- **Assisted Global Positioning System (aGPS)-Sensor:**
  Mithilfe des satellitenbasierten Ortungssystem GPS kann die Position (Längen- und Breitengrad) des Empfangsgerätes (z. B. Smartphone) errechnet werden. Im Vergleich dazu ermöglicht aGPS die Verwendung von zusätzlichen Daten, die von einem aGPS-Server über das Internet bereitgestellt werden, um die Satelliten bei schlechten Signalbedingungen - beispielsweise in Städten - zu orten und zu nutzen.
- **Beschleunigungsmesser:**
  Ein Beschleunigungsmesser ist ein elektromechanischer Sensor, der Beschleunigungskräfte messen kann. Die meisten Smartphones haben einen eingebauten 3-Achsen-Beschleunigungsmesser, um die Beschleunigung in allen drei Achsen gleichzeitig zu messen. Neben der Beschleunigungsmessung kann durch die statische Beschleunigung der Erde auch die Orientierung oder die Neigung des

Smartphones zur Erde ermittelt werden. In der Praxis wird das aber nicht angewendet. [22]

- **Gyroskop-Sensor:**
  Ein Gyroskop-Sensor misst die Geschwindigkeit, mit der sich das Smartphone um eine Raumachse dreht. Viele Smartphones verfügen über ein Drei-Achsen-Gyroskop, das Rotationswerte in jeder der drei in Abbildung 2.2 dargestellten Achsen liefert. Allerdings kann ein Gyroskop nur relative Orientierungsänderungen messen, sodass keine absolute Orientierung mittels eines Gyroskop-Sensors möglich ist.

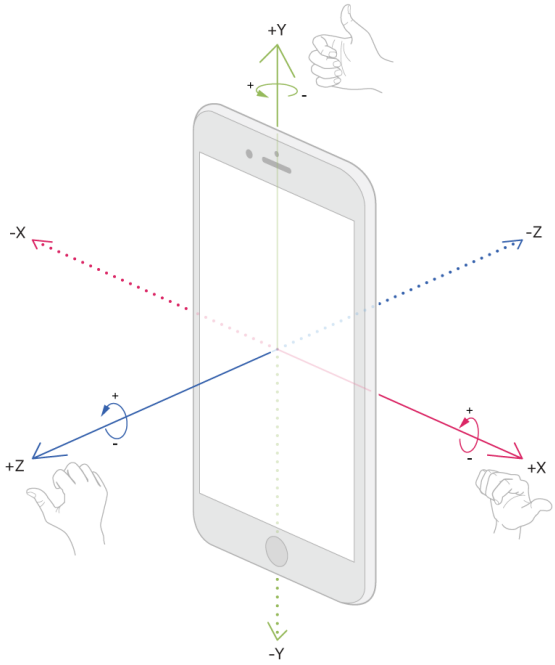


Abbildung 2.2: Rotationsachsen an einem Smartphone
Quelle: (07.09.2020, 14:20) `https://developer.apple.com/documentation/coremotion/getting_raw_gyroscope_events`

Eine Kombination aller ermittelten Messdaten, durch die genannten Sensoren eines Smartphones oder Tablets, ermöglicht die Bestimmung der Position und Orientierung des Gerätes. Insgesamt ergeben sich 6 Freiheitsgrade aus der Position entlang der x-, y- und z-Achse sowie den Rotationen um diese drei Achsen. Eine visuelle Darstellung davon ist in Abbildung 2.2 zu finden. So liefert das GPS durch Längen- und Breitengrad sowie der Höhenmeter, Werte für alle drei Koordinatenachsen. Um bereits eine Fehlerquelle zu reduzieren, ist es ratsam, nicht den Höhenwert des GPS zu verwenden, sondern die y-Koordinate auf einen festen Offset über dem Boden zu setzen, der in etwa der Höhe entspricht, in der ein Benutzer ein solches Gerät normalerweise halten würde. Je nach Anwendungsfall kann es aber auch sinnvoll sein, der x- und z-Achse eigene Werte zu geben. Die Rotation um die y-Achse bzw. die horizontale Orientierung des AR-Gerätes kann durch die Erfassung des Magnetfelds mithilfe des Kompasses dargestellt werden. Die übrigen zwei Drehungen um die x- und z-Achse liefert der

Beschleunigungsmesser und der Gyroskop-Sensor. Andere Kombinationsmöglichkeiten von Sensoren, sodass 6 Freiheitsgrade entstehen, sind ebenfalls möglich.

2.1.2 Visuelles Tracking

Visuelles Tracking ist eine Technik, bei der Algorithmen der Computer Vision (CV) verwendet werden, um bekannte Muster im aktuellen Kamerabild zu erkennen, damit die räumliche Beziehung zwischen Markern und der Kamera berechnet werden kann. Hierfür gibt es zwei grundlegende Konzepte:

- **Marker-basiertes Tracking**
  Beim Marker-basierten Tracking werden ein oder mehrere Marker oder sogenannte Tags, wie z. B. ein QR-Code verwendet, um Position und Orientierung eines virtuellen Objekts zu bestimmen. Um Marker zu tracken, müssen diese im Vorfeld definiert sein. Der Tracker kann dann gezielt nach diesen künstlichen Markern im Kamerabild Ausschau halten, um das virtuelle Objekt darauf zu platzieren. [16,25]
- **Markerloses Tracking**
  Eine komplexere Methode, die sich ebenfalls auf CV-Algorithmen stützt, wird markerloses Tracking genannt und ist nicht von künstlichen Markern abhängig [20]. Stattdessen sucht der Tracker nach natürlichen auffälligen Stellen, sogenannten Feature Points, im Kamerabild. Natürliche Merkmale sind Bereiche auf der Oberfläche eines Objekts, wie z. B. Ecken und Kanten, die einen hohen Grad an lokalem Kontrast aufweisen. Die Idee ist, dass ein zuvor bekanntes 2D- oder 3D-Modell eine Ansammlung von Feature Points ergibt [6]. Bei dieser Sammlung spricht man auch von einer Punktwolke. Anhand der Anordnung dieser Merkmale kann das vordefinierte Modell und somit das gesamte Objekt im Kamerabild identifiziert werden. Dadurch ist es möglich, fast jedes planare oder 3-dimensionale Objekt als Referenz zu verwenden, solange es eine gewisse Menge an visueller Information und Kontrast enthält. Durch die Bewegung der Kamera um das Objekt ist die Bestimmung der Kameralage in Bezug auf die Feature Points ermittelbar [18].

In dieser Arbeit werden beide Arten des Trackings noch einmal in Kapitel 7 in Bezug auf das Tracking einer Smartwatch untersucht.

2.2 Rendering

Unter Rendering versteht man die Darstellung einer 3D-Umgebung, die mittels Projektion auf einen zweidimensionalen Bildschirm übertragen wird [33]. Diese Umgebung, auch Szene genannt, kann auch zweidimensional sein. Jedoch sind im AR Kontext vorwiegend dreidimensionale Räume von Interesse, um perspektivische Projektionen zu

erzielen.
Solch eine Szene besitzt neben einem Koordinatensystem auch eine virtuelle Kamera. Vergleichbar mit einer Smartphone-Kamera, die die dreidimensionale Welt auf einen zweidimensionalen Bildschirm überträgt, übernimmt die virtuelle Kamera diese Aufgabe für die virtuelle 3D-Umgebung. Somit kann ein Objekt in der Szene durch Bewegen der Kamera oder des Objekts selbst aus allen Blickwinkeln und Entfernungen durch die virtuelle Kamera betrachtet werden.
Um die Illusion von AR auf dem Smartphone zu erlangen, ist es nötig, das Kamerabild des Smartphones und das Kamerabild einer virtuellen Kamera zu kombinieren. Dazu muss ein weiteres Koordinatensystem für die reale Umgebung definiert werden, um die Position der Smartphone-Kamera zu tracken. Danach werden die Kameras mit den jeweiligen Koordinatensystemen praktisch übereinandergelegt und synchronisiert. Das heißt, wenn man die Smartphone-Kamera in eine bestimmte Richtung bewegt, folgt die virtuelle Kamera in genau dieselbe Richtung. Stimmen die Systeme nicht überein, resultieren daraus Fehler in der Betrachtung. Daher ist das Tracking der Smartphone-Kamera besonders wichtig bei diesem Prozess.

3 Das Patent wristwatch based interface for augmented reality eyeware

Das Patent *wristwatch based interface for augmented reality eyeware* [26] von Jonathan M. Rodriguez II. hat das Ziel, eine gesellschaftlich akzeptable und intuitive Methode der gestischen Eingabe für Augmented-Reality-Brillen zu realisieren. Wie in Abbildung 3.1 zu sehen ist, werden dazu der Touchscreen einer Smartwatch **102** und die grafische Oberfläche **501** einer AR-Brille **101** zu einer Benutzeroberfläche kombiniert. Neben vielen interessanten Anwendungen, die durch das System möglich wären, gibt das Patent auch Einblicke in die Funktionsweise, wodurch das Architektur-Konzept dieser Arbeit geprägt werden soll.

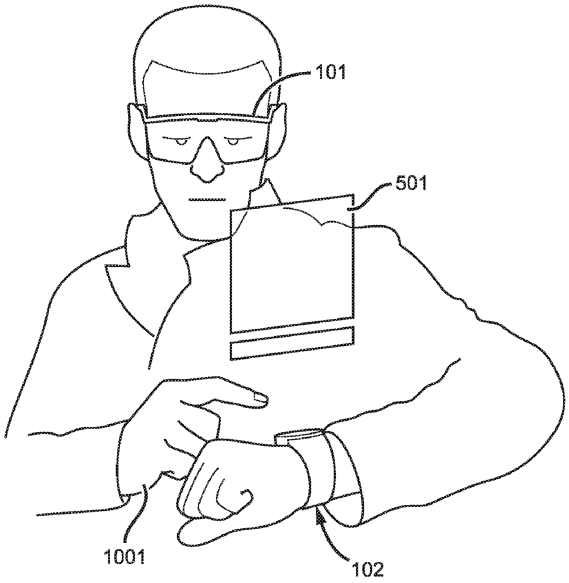


Abbildung 3.1: Illustration der Funktionsweise des Patents
Quelle: Originalbild aus dem Patent [26, Seite 1]

3.1 Funktionsweise des kombinierten Hardware-Systems

Wie schon erwähnt, sieht das Konzept vor, das Display der Smartwatch und die AR-Oberfläche zu einem kontinuierlichen Display zu kombinieren[1]. Ebenso könnte auch bei einigen Anwendungen eine reguläre Armbanduhr verwendet werden[2]. Der Funktionsumfang der herkömmlichen Uhr wird durch AR erweitert, sodass diese mit Hilfe von Augmented Reality zu einer simulierten Smartwatch wird[2]. In diesem Fall müsste die

[1] [26, Abs. 0098]
[2] [26, Abs. 0092]

Anwendung ausschließlich auf der AR-Brille laufen. Im Gegensatz dazu gibt es auch die Idee, dass eine mit der AR-Brille gekoppelte Smartwatch Rechenlast der AR-Brille abnimmt. Rechenaufwendigen Prozessen wie beispielsweise das Tracking der Uhr oder das Rendern von Views scheinen dabei sinnvoll. Demzufolge könnte die AR-Brille kleiner sowie preisgünstiger werden und weniger Strom verbrauchen[3]. Bei der Aufteilung des Systems spricht man häufig von einem Master-Slave-Prinzip[4]. Im folgenden wird dafür durchgängig das Synonym Primary-Secondary-Prinzip gebraucht, wie von der IETF, einer Organisation, die sich mit der Weiterentwicklung des Internets befasst, empfohlen wird[5]. Grundlage einer Rechenlastverteilung ist eine Kommunikationsverbindung zwischen Smartwatch und AR-Brille, um beispielsweise die Tracking- oder auch Renderdaten zu senden. Dabei könnten Bluetooth, W-LAN, NFC oder andere Funk-Protokolle zum Einsatz kommen[6]. Das Primary-Secondary-Prinzip zwischen Smartwatch und AR-Brille ist aufgrund der Rechenleistung aktueller Geräte vermutlich jedoch nicht geeignet. Stattdessen könnte aber ein leistungsfähigeres drittes Gerät, wie z. B. ein Smartphone, verwendet werden[7], welches die Arbeit der AR-Brille in Teilen abnimmt. So würde ein Gerät einen Großteil der Verarbeitungsleistung übernehmen, aber nicht die gesamte[8]. Vorzugsweise sollten alle Plattformen eine Vollversion der Anwendung besitzen[9], sodass auch bei einem Verbindungsabbruch die Anwendung weiterlaufen kann. Zudem könnten durch die Kopplung von Smartwatch und AR-Brille auch andere Daten, wie zum Beispiel Live-Daten eines Pulsmessers der Smartwatch[10]oder Benachrichtigungen und Hinweise an die Brille gesendet und sofort in AR angezeigt werden[10]. Gleichermaßen kann der Kommunikationskanal genutzt werden, um Daten von der AR-Brille an die Smartwatch zu senden. Beispielsweise könnte so der Zustand der Anwendung auf der Smartwatch durch eine von der Brille erkannte Geste geändert werden[11]. Der Touchscreen der Smartwatch könnte wiederum auch die AR-Oberfläche anpassen, wodurch ein fließender Übergang der Oberflächen und Bedienung möglich ist[12], wie zu Beginn des Abschnitts geschildert.

3.2 Eingabe- und Interaktionsmöglichkeiten des Systems

Im Patent ist eine Vielzahl von verschiedenen und vielseitigen Eingaben aufgeführt. Die Abbildung 14 des Patents und die Abbildung 11.1 zeigen ein Ablaufdiagramm mit

[3] [26, Abs. 0093]
[4] [26, Abs. 0101]
[5] Entwurf des IETF zum Alternativen Sprachgebrauch von Master-Slave in der IT (08.09.2020, 11:30) - https://tools.ietf.org/html/draft-knodel-terminology-01#section-1.1.1
[6] [26, Abs. 0048]
[7] [26, Abs. 0123]
[8] [26, Abs. 0101]
[9] [26, Abs. 0101, 0105]
[10] [26, Abs. 0094]
[11] [26, Abs. 0097]
[12] [26, Abs. 0097, 0099]

drei Eingabemöglichkeiten: der Touchscreen der Smartwatch, die Gestenerkennung über die AR-Brille und eine Gestenerkennung über die Smartwatch. Neben den üblichen Interaktionen mit einem Touchscreen, wie Drücken, Streichen und Zoomen sowie weitere Multitouch-Gesten, könnten diese auch für AR-Oberflächen als Geste in der Luft möglich sein[13]. Dazu könnte es eigene programmierbare Gesten geben, die beispielsweise Apps starten oder beenden[14]. Mittels Gestenerkennung könnte die reguläre Armbanduhr zu einer simulierten Smartwatch werden, indem das Drehen von Rädchen oder Drücken von Knöpfen vom Benutzer simuliert oder vorgetäuscht werden[15]. Zudem könnte eine Sprachsteuerung bei der Bedienung von Benutzeroberflächen helfen oder Aufgaben entgegennehmen, wie man es von aktuellen Sprachassistenten, wie Alexa[16], Google Assistent[17] und Siri[18] kennt.

3.3 Weitere Anwendungsbeispiele

Neben den genannten Eingabemöglichkeiten beschreibt und illustriert das Patent weitere konkrete Anwendungsbeispiele, die durch die Kombination von Smartwatch oder Armbanduhr und AR-Brille möglich wären.

Beispielsweise könnte es für den Bereich Sicherheit eine für AR angepasste Version des Entsperrmusters geben, welches bei Android-Smartphones verbreitet ist. In Abbildung 3.2 ist diese Anwendung zu sehen. Das Entsperren des Geräts würde durch Verbinden von virtuellen Punkten zu einem Muster erfolgen. Dabei ist das Entsperrmuster **601** an der Uhr **102** in AR verankert. Auch eine dreidimensionale Version in AR, die für mehr Sicherheit sorgen könnte, ist möglich.[19]

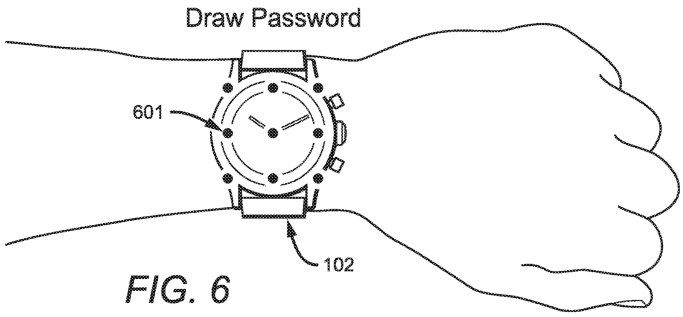


Abbildung 3.2: Entsperrmuster in AR für eine Smartwatch oder eine reguläre Armbanduhr
Quelle: Originalbild aus dem Patent [26, Seite 6 Abbildung 6]

[13] [26, Abs. 0056, 0057]
[14] [26, Abs. 0080, 0050]
[15] [26, Abs. 0072]
[16] Alexa (14.09.2020, 17:35) - https://www.amazon.de/b/?node=17084415031
[17] Google Assistent (14.09.2020, 17:35) - https://assistant.google.com/intl/de_de/
[18] Siri (14.09.2020, 17:35) - https://www.apple.com/de/siri/
[19] [26, Abs. 0061]

Ein weiteres, eher simpleres Beispiel, ist ein AR-Audioplayer, welcher an einer Uhr **102** verankert in Abbildung 3.3 zu sehen ist. Dabei gibt es Informationen über den Titel **404**, das Album **405** und den Künstler **406** des aktuell gespielten Songs sowie drei Steuerelemente. Neben einem Start- bzw. Stop-Knopf **402** befinden sich zwei Knöpfe, um den vorherigen **401** bzw. den nächsten **403** Titel abspielen zu können.[20]

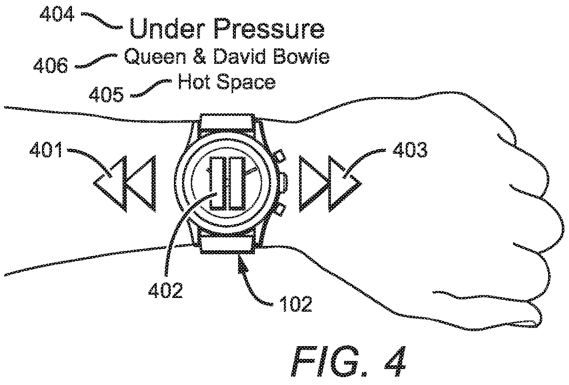


Abbildung 3.3: AR-Audioplayer für eine Smartwatch oder eine reguläre Armbanduhr
Quelle: Originalbild aus dem Patent [26, Seite 5 Abbildung 4]

Anders als bei den Beispielen zuvor wird bei der in Abbildung 3.4 dargestellten Karten-Anwendung eine Smartwatch **1201** benötigt. Bei der AR-Oberfläche **1301** handelt es sich hierbei um eine Mitfahrgelegenheit-App. Ebenso ist auch eine komplette Navigations-App in diesem Stil vorstellbar. Exemplarisch bilden hier die AR-Oberfläche **1301** und die Benutzeroberfläche der Smartwatch zusammen einen fließenden Übergang der Karte, sodass ein größerer Ausschnitt für den Benutzer sichtbar ist. Laut dem Patent könnte das Heran- und Herauszoomen der Karte zum einen über den Touchscreen als auch über Gesten nahe der Smartwatch erfolgen.[21]

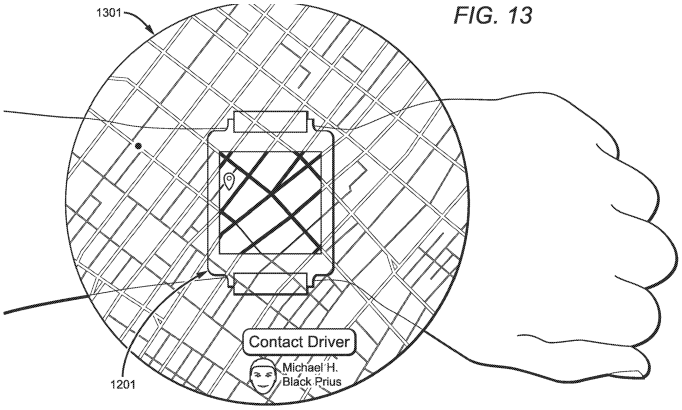


Abbildung 3.4: Karten-App der Smartwatch erweitert durch eine AR-Oberfläche
Quelle: Originalbild aus dem Patent [26, Seite 11 Abbildung 13]

[20] [26, Abs. 0059]
[21] [26, Abs. 0098, 0099]

4 Anforderungen an die Architektur

In diesem Kapitel ist beschrieben, welche Anforderungen an das Konzept gestellt werden und welche äußeren Einflüsse oder Faktoren berücksichtigt werden müssen. Weiterhin ist dargestellt, welche Hardware-Ansätze aus dem Abschnitt 3.1 übernommen werden, um Daten einer Anwendung in einem kabellos verbundenen Hardware-System zu verarbeiten.

4.1 Bewertung der Vorgaben des Patents

Im ersten Ansatz des Patents ist einzig und allein die AR-Brille für alle Funktionen der Anwendung verantwortlich. Diese umfassen die gesamte Datenverarbeitung von Tracking und Rendering bis hin zur Anwendungslogik und sonstige Funktionalität, die eine AR-Brille leisten würde. Grundlage des im Patent dargestellten Ansatzes ist eine reguläre Armbanduhr ohne CPU und Kommunikationsmöglichkeit. Diese Arbeit bezieht sich jedoch auf Smartwatches, deren Bildschirm durch AR erweitert wird oder als Erweiterung der AR-Oberfläche dient. Daher ist dieser Ansatz nicht geeignet. Jedoch könnte das fertige Architektur-Konzept so gestaltet sein, das auch reguläre Armbanduhren in Kombination mit der Brille verwendet werden können, wie in Abschnitt 3.1 beschrieben.

Das Primary-Secondary-Prinzip scheint durch die Arbeitsaufteilung daher passender zu sein. Das Patent beschreibt, wie beispielsweise ein Gerät die Datenverarbeitung und ein weiteres die Oberflächendarstellung einer Anwendung übernimmt[22]. Dabei kommt ein Kommunikationskanal zum Einsatz, um die Plattformen zu koppeln und zu synchronisieren. Ein großer Nachteil dieses Ansatzes ist allerdings, dass essenzielle Funktionen eines Geräts vom anderen abhängig sind. In dem Patent bleibt auch offen, was passiert, wenn der Benutzer nur eines der Geräte zur Hand hat. Daher scheint es am sinnvollsten, das dritte vorgeschlagene Konzept zu verwenden.

Im dritten Ansatz des Patents haben Smartwatch und AR-Brille jeweils eine eigene Vollversion der Anwendung. Beide Plattformen sind somit selbst für die Datenverarbeitung verantwortlich. Das bedeutet, dass sowohl die Smartwatch als auch die AR-Brille dem Nutzer beispielsweise eine Musik-Anwendung in vollem Umfang bereitstellt. Weiterhin lässt sich die Anwendung der Smartwatch und die der AR-Brille zu einer Anwendung kombinieren, sodass entweder die eine Anwendung die zweite ergänzt oder die Anwendungen einen fließenden Übergang von Benutzeroberfläche und Bedienung haben. Die Bereitstellung der Vollversion einer Anwendung kann jedoch abhängig von dieser selbst sein. Eine abgespeckte Version auf einer der beiden Plattformen schließt dieser Ansatz

[22] [26, 0101]

deshalb auch nicht aus.

Diese Aufteilung in unabhängige Versionen passt zudem zu aktuellen Smartwatch-Ökosystemen, wie das von Apple oder Samsung. In diesen Ökosystemen stehen beispielsweise die Smartwatch-Apps und deren Smartphone-Pendants im Datenaustausch. So ist es möglich, die Daten einer Smartphone-App auf eine Smartwatch-App, mit meist geringerem Umfang, zu überführen[23,24]. Exemplarisch dafür sind Nachrichten-Apps wie iMessage oder die Android Messages App, bei denen die Chat-Verläufe auf beiden Geräten synchron bleiben. Auch bei Fitness-Apps stehen Smartwatch und Smartphone im Datenaustausch[25]. Über die Smartwatch werden Aktivitätsdaten tageweise gesammelt und mit gesammelten Daten des Smartphones kombiniert und gespeichert. Seit watchOS 6 ist es möglich, eigenständige Smartwatch-Apps[26] komplett ohne ein iOS-Pendant zu entwickeln.

Derartige Anwendungen, die für verschiedene Plattformen wie Smartwatch und Smartphone, aber auch PC oder TV eines IT-Ökosystems entwickelt werden, sind in einer sogenannten App- oder Anwendungs-Gruppe zusammengefasst[27]. In solch einer Gruppe kann Quellcode plattformunabhängig oder -spezifisch erstellt werden. Beispielsweise müssen so Kernfunktionen der Anwendung nur einmalig entwickelt und können auf jeder Plattform wiederverwendet werden. Dadurch können auch Views und Anwendungslogik der Smartwatch-App genutzt werden, um mithilfe der AR-Brille eine reguläre Armbanduhr als Smartwatch zu simulieren.

4.2 Abgeleitete Anforderungen

Basierend auf der Bewertung der Vorgaben des Patents ergeben sich folgende Anforderungen an die Architektur:

Plattformunabhängigkeit
Die Architektur muss plattformunabhängig sein, da sie als Basis der Anwendungslogik dient und somit auch auf verschiedenen Plattformen wiederverwendet wird.

Interoperabilität auf Ebene des Kontrollflusses
Die Architektur muss eine Kopplung oder viel mehr eine Art Symbiose der Plattformen

[23] Datenaustausch zwischen iPhone und Apple Watch (27.08.2020, 11:40) - `https://www.apple.com/iphone-and-apple-watch/`
[24] Datenaustausch zwischen Android-Smartphone und der Galaxy Watch 3 (27.08.2020, 11:40) - `https://www.samsung.com/us/watches/galaxy-watch3/#experience`
[25] Fitness-Anwendung unter der Galaxy Watch3 (27.08.2020, 12:00) - `https://www.samsung.com/us/watches/galaxy-watch3/#health`
[26] Creating Independent watchOS Apps (11.09.2020, 12:25) - `https://developer.apple.com/documentation/watchkit/creating_independent_watchos_apps`
[27] App Groups Entitlement (27.08.2020, 12:05) - `https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_security_application-groups`

gewährleisten. Das bedeutet, dass die Architektur so konzipiert sein muss, dass zwei Anwendungslogiken interoperabel sind, sich gegenseitig beeinflussen können und über einen gemeinsamen Kontext verfügen, der Auswirkung auf den Kontrollfluss beider Anwendung haben kann. Zudem soll die Architektur jeder Zeit diese Symbiose eingehen können, die wiederum sofort getrennt werden kann.

Schnittstellenoffenheit für verschiedene Kommunikationsmöglichkeiten
Des Weiteren muss die Architektur Schnittstellen, für einen oder mehrere Kommunikationskanäle, zulassen, um eine Kopplung der Plattformen zu gewährleisten, unabhängig von Protokollen oder Low-Level-APIs.

Erhaltung des Anwendungszustandes bei der Synchronisation und nach Verbindungsende oder -abbruch
Wichtig für die Benutzung ist, dass die App-Zustände nach dem Verbinden synchronisiert oder beim Trennen erhalten bleiben, sodass an derselben Stelle vor oder nach der Kopplung fortgefahren werden kann.

Abstraktion oder Generalisierung von Benutzeraktionen
Bei der Kopplung müssen neben der Synchronisierung der Anwendungsdaten auch Benutzeraktionen unabhängig ihrer Eingabequelle registriert werden. Daher sollte die Architektur in der Lage sein, Benutzeraktionen so zu abstrahieren, dass verschiedene Eingaben, wie Tastaturbedienung oder Spracherkennung realisierbar sind.

Verarbeitung konkurrierender Benutzerprozesse
Während der Kopplung dürfen keine race conditions entstehen. Unter einer race condition versteht man „[e]ine Situation, in der mehrere [Prozesse] Zugriff auf gemeinsam genutzte Daten haben und das Ergebnis der Berechnung von der jeweiligen Ausführungsreihenfolge abhängt [...]“ [13, Seite 92]. Auf das System des Patents im gekoppelten Zustand bezogen bedeutet dies, dass eine Benutzerinteraktion auf beiden Geräten abhängig von ihrer Reihenfolge ebenfalls unterschiedliche Ergebnisse hervorrufen kann. Daher muss sicher gestellt werden, dass die Aktionen in der Reihenfolge ausgeführt werden müssen, in der sie auch auftreten.

Peer-to-Peer-Verbindung
Bei der Verbindung zwischen den Geräten muss es sich um eine Peer-to-Peer-Verbindung handeln. Das heißt, dass die Daten direkt zwischen den Geräten und nicht über einen Server ausgetauscht werden. So kann die Kommunikation auch an Orten ohne Internetzugang stattfinden und eine geringe Latenz sichergestellt werden.

5 Softwaretechnische Grundlagen

Dieses Kapitel beschreibt softwaretechnische Grundlagen für die Architekturentwicklung sowie für die darauffolgende Umsetzung. Speziell wird auf Model-View-ViewModel (MVVM) sowie auf die Frameworks SwiftUI und Combine eingegangen. Während MVVM zum Verständnis des Architekturkonzepts nötig ist, werden SwiftUI und Combine bei der Umsetzung verwendet, um MVVM auf Apple Plattformen zu unterstützen. Warum Plattformen von Apple verwendet werden, ist im Kapitel 7 beschrieben.

5.1 Model-View-ViewModel

Model-View-ViewModel ist ein Software-Architekturmuster. Unter einem Software-Architekturmuster oder auch Entwurfsmuster ist eine bewährte Lösungsvorlage für wiederkehrende Entwurfsprobleme zu verstehen. MVVM erleichtert eine Trennung zwischen der Umsetzung der grafischen Benutzeroberfläche (der View in Abbildung 5.1) und der Entwicklung der Geschäftslogik oder der Back-End-Logik (dem Model). So ist die View nicht von einer bestimmten Model-Plattform abhängig und das Model nicht von einer speziellen View.

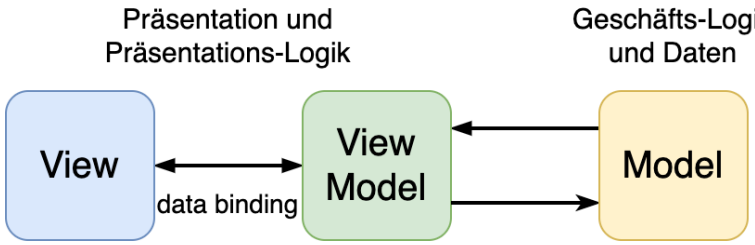


Abbildung 5.1: Schematische Darstellung des Model-View-ViewModel Entwurfmusters

Die View reflektiert die Daten im Model, weshalb es in einer Anwendung nur genau ein Model geben sollte. Reflektieren bedeutet, dass die View auf Änderungen im Model reagiert und das Model jederzeit darstellt. Dieses Verhalten der View nennt man *reactive*, auf Deutsch so viel wie rückwirkend. Die View selbst kann deshalb fast zustandslos und deklarativ sein. Das bedeutet, man deklariert einmal, **was** in der View zu sehen ist und wie sie sich je nach Zustand des Models ändern soll. Im Gegensatz zu deklarativen Views, deren Code für Menschen leicht verständlich ist, gibt es auch imperative. Bei dieser Art von View-Entwicklung beschreibt der Code, **wie** die View zu erstellen ist. Die Beschreibung ist vergleichbar mit einer Art Bauanleitung für den Computer, die aber schlecht lesbar und begreifbar für den Menschen ist.

Obwohl MVVM auf deklarative Views abzielt, ist die Art der View-Entwicklung nicht entscheidend, da das Model und die View keine direkte Verbindung zueinander haben, wie

es sonst bei anderen Entwurfsmustern üblich ist, um Daten auf der Benutzeroberfläche anzuzeigen. Stattdessen wird zur Verbindung der beiden das ViewModel benötigt. Diese bindet die View an das Model durch eine Bindungsfunktion (data binding). Dabei entsteht eine bidirektionale Zuordnung von Daten aus dem ViewModel zu den jeweiligen Objekten auf der View. Zum Beispiel bildet eine Liste von Zahlen im Model die Einträge in einem Dropdown-Menü auf der View ab. Aber auch das Binden von Daten aus dem Model an Benutzereingaben durch Maus, Tastatur, Touch-Screens oder Gesten ist möglich. Beispielsweise kann durch das Tippen der Tastatur ein Text im Model verändert werden. Die Änderungen im Model werden dann direkt auf der View reflektiert. [5, 9, 12, 23]

5.2 SwiftUI

SwiftUI[28] ist ein Framework von Apple und basiert auf dem Entwurfsmuster MVVM. Es bietet die Möglichkeit, Views deklarativ zu erstellen und Zustände in einer leichtgewichtigen Weise zu modellieren, sodass Zustandsänderungen sofort in der Benutzeroberfläche reflektiert werden. SwiftUI stellt außerdem Event-Handler für Gesten und anderen Arten von Eingaben bereit. Des Weiteren existieren vorgefertigte Views, Steuerelemente und Layout-Strukturen, woraus eigene individuelle Ansichten für alle Apple-Plattformen entwickelt werden können, die sogar Quellcode zwischen den Plattformen teilen. Darüber hinaus wird auch Barrierefreiheit und verschiedene Sprachen, Länder oder Kulturregionen unterstützt.

5.3 Combine

Combine ist ein Framework von Apple für funktionale reaktive Programmierung (FRP), auch Datenflussprogrammierung genannt. Diese Art von Programmierung wird auf asynchronen Strömen von Daten oder Elementen angewendet, die im Verlauf der Zeit verarbeitet werden können. Im Konzept von FRP spielt das Beobachtermuster eine wichtige Rolle. Es ist wie MVVM ebenfalls ein Entwurfsmuster. Das Beobachtermuster kann an ein Objekt „geheftet“ werden und liefert Benachrichtigungen über Änderungen und Aktualisierungen des Objekts. Wenn man diese Benachrichtigungen im Laufe der Zeit betrachtet, bilden sie einen Strom von Objekten und somit eine Art Quelle des Stroms. Mit funktionaler reaktiver Programmierung, in diesem Fall Combine, kann Code erstellt werden. Dieser beschreibt, was passiert, wenn Daten von einem Datenstrom abgerufen werden.

Ein klassisches Beispiel für die Anwendung funktionaler reaktiver Programmierung sind Benutzeroberflächen, wie die in SwiftUI. Beispielsweise führen Benutzerereignisse wie

[28] SwiftUI Dokumentation (02.07.2020, 15:20) - `https://developer.apple.com/documentation/swiftui`

das Ändern von Textfeldern, Tippen oder Mausklicks auf UI-Elemente dazu, dass Daten übertragen werden. Combine ermöglicht auch das Überwachen von Eigenschaften bzw. Variablen, das Binden an Objekte für MVVM. Zudem wird fast die gesamte API von Apple unterstützt. Combine beschränkt sich somit nicht nur auf Benutzeroberflächen. Jede Sequenz von asynchronen Operationen kann als Strom oder Pipeline wirksam sein, insbesondere wenn die Ergebnisse jedes Schrittes in den nächsten Schritt fließen. Ein Beispiel hierfür könnte eine Reihe von Netzwerk-Serviceanforderungen sein, gefolgt von der Decodierung der Ergebnisse. Auch eventuell anfallende Fehler können mit Hilfe von einer Fehlerbehandlung direkt im Strom abgefangen und beispielsweise mit Standardwerten ersetzt werden. [3, 11, 35]

6 Konzept der Architektur

Aufbauend auf den Anforderungen und den Grundlagen werden in diesem Kapitel Struktur und Funktionsweise des Konzepts erläutert. Dazu wird die grundlegende Architektur iterativ in jedem Abschnitt um Funktionalität erweitert, bis alle Anforderungen integriert bzw. erfüllt sind.

6.1 Struktur der Architektur

Der Schwerpunkt der Architektur ist ihre Flexibilität. Zum einen soll sie eigenständig funktionieren und zum anderen muss sie in der Lage sein, sich jeder Zeit mit einem Architektur-Pendant kombinieren zu lassen, sodass ein fließender Übergang für den Benutzer entsteht. Dazu sollten wie im Abschnitt 4.2 geschildert, die Eingabemöglichkeiten so abstrahiert sein, dass sie plattformunabhängig verarbeitet werden können.

In Anlehnung an die Plattformunabhängigkeit gibt es ein weiteres Entwurfsmuster, welches aufgrund seiner Funktionsweise prädestiniert als Lösung ist. Das sogenannte *state management pattern*[29], was Zustandsverwaltungsmuster bedeutet. Es basiert auf dem *state* Entwurfsmuster [10] und dient als zentraler Speicher für alle Komponenten einer Anwendung, wobei Regeln sicherstellen, dass der Zustand nur auf vorhersehbare Weise verändert werden kann.

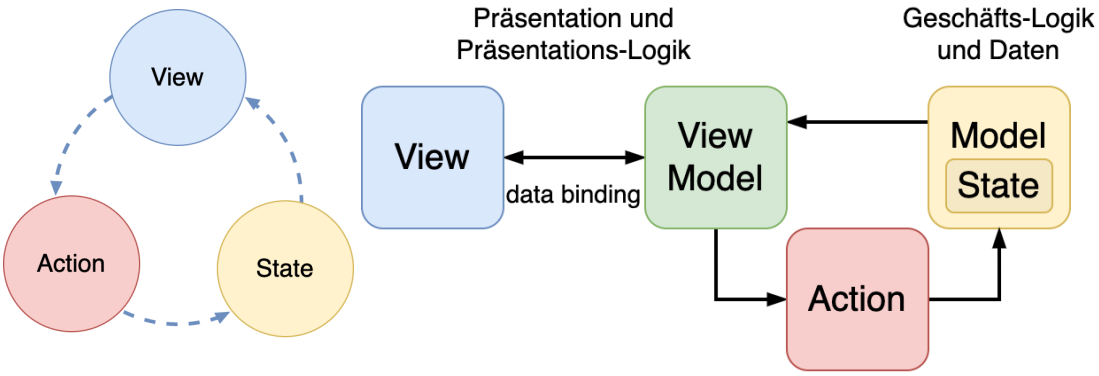
Konkret bedeutet das für eine Anwendung, dass sie aus den folgenden drei Teilen besteht:

- *State* / Anwendungszustand
  Der Anwendungszustand ist die sogenannte *single source of truth*, was so viel wie die einzige Quelle der Wahrheit bedeutet. Damit ist der zentrale Speicher gemeint, der alle Daten der Anwendungen enthält.
- *View* / Benutzeroberfläche
  Die View ist im herkömmlichen Sinne eine Benutzeroberfläche. In diesem Fall ist die View aber auch eine deklarative Abbildung des Anwendungszustandes und vergleichbar mit der View bei MVVM. Das bedeutet, dass alle Daten in dem State über die View für den Benutzer sichtbar sind.
- *Actions* / Aktionen
  Eine Action ist typischerweise eine Reaktion auf eine Benutzerinterkation auf der View. Sie beschreibt, wie der aktuelle Zustand in den nächsten Zustand übergeht. Eine Action ist die Regel, die den Zustand auf vorhersehbare Weise verändern

[29] State management pattern (14.07.2020, 13:45) - `https://vuex.vuejs.org/#what-is-a-state-management-pattern`

kann. Sie ist vergleichbar mit einem Zustandsübergang in einem Zustandsautomat.

Durch die drei Bestandteile und Eigenschaften entsteht ein unidirektionaler Datenfluss, der in Abbildung 6.1a abgebildet ist. Dabei werden die Daten im State von der View reflektiert. Über die View können Actions ausgelöst werden, die wiederum den State verändern. Des Weiteren sorgt die Eigenschaft, dass Actions nur eine Reaktion auf eine Benutzereingabe sind, für eine Abstrahierung der Eingabemöglichkeiten, sogar über Plattformen hinweg. Beispielsweise könnte eine von der AR-Brille erkannte Geste die gleiche Action auslösen, wie ein Knopfdruck auf der Benutzeroberfläche der Smartwatch, sodass auf beiden Geräten der State gleichermaßen verändert wird. Jedoch ist dadurch noch keine komplette Plattformunabhängigkeit erreicht. Denn das *state management pattern* wird nicht direkt von Benutzeroberflächen-Frameworks umgesetzt, wie es bei MVC oder MVVM der Fall ist. Dessen ungeachtet kann aber das *state management pattern* mit MVVM kombiniert werden. Dieses ist aufgrund seiner Beschaffenheit und Funktionalität besonders gut zur Kombination geeignet. Wie in Abbildung 6.1b zu sehen ist, wird der State in das Model als einzige Datenquelle eingesetzt. Zugleich gehen keine Eigenschaften von MVVM verloren, weshalb nun eine Unabhängigkeit des Models von der View und entgegengesetzt gewährleistet ist. Darauf aufbauend kann beispielsweise die gleiche Geschäftslogik auf einer AR-Brille und einer Smartwatch verwendet werden. Nennenswert ist außerdem, dass das *state management pattern* auch mit MVC kombinierbar ist. Problematisch dabei ist jedoch, dass der Controller der Anwendungslogik sehr stark mit View zusammenhängt, was eine Wiederverwendung der Logik auf anderen Plattformen erschwert.



(a) Schematische Darstellung des Zustandsverwaltungs Entwurfmusters

(b) Schematische Darstellung von MVVM zusammen mit dem Zustandsverwaltungs Entwurfmuster

Abbildung 6.1: Schematische Darstellung des *state management patterns* allein und in Kombination mit MVVM

Der einzige Unterschied zum herkömmlichen MVVM ist nun, dass Benutzeraktionen und Änderungen auf der View nicht über das ViewModel direkt an den State gehen. Stattdessen wird eine Action von der View über ViewModel an das Model gesendet. Dort ist die Geschäftslogik hinterlegt, wie die Action den State verändert. Zudem ist der State von außen schreibgeschützt, sodass wirklich nur noch Actions Veränderungen

auslösen können.

Auf Basis des bisherigen Konzepts zeigt Abbildung 6.2 eine konkrete Struktur der Architektur. Der Store (gelb) repräsentiert die Laufzeit, welche die Anwendung leitet. Übertragen auf das Schema in Abbildung 6.1b repräsentiert der Store das Model (gelb) samt State. Das bedeutet, dass es pro Anwendung genau einen globalen Store gibt. Dieser muss für alle Views (blau) zugänglich sein, damit sie den State (orange) mithilfe von MVVMs data binding reflektieren. Über die Views können außerdem Actions (rot) aufgerufen werden. Diese gelangen zuerst in einen sogenannten Dispatcher (lila), welcher als Verteiler dient. Anschließend beauftragt der Dispatcher den Reducer (grün) die Action auszuführen, wodurch der State geändert wird. Der Reducer implementiert für jede Action die Regeln, wie der State verändert werden soll. Somit ist der Reducer die Geschäftslogik der Anwendung. Ein großer Nachteil ist allerdings, dass State und Reducer aufgrund der Menge an Variablen und Logik sehr unübersichtlich werden. Deshalb können Reducer, States und Actions ineinander geschachtelt werden. So erhält man aus einem flachen State mit vielen Variablen eine Baumstruktur. Diese ist äquivalent zu einem zusammenhängenden kreisfreien ungerichteten Graph mit einem Wurzelknoten, Knoten und Blättern. Selbiges gilt für die Reducer und Actions. Dadurch ist es möglich, gezielte kleine Stores aus dem Haupt-Store zu erstellen, die von der Haupt-View an darunter liegende Views weitergegeben werden können. Der Quellcode A.1 im Anhang zeigt eine mögliche Umsetzung dieser Architektur in Swift, wobei Funktion die `erledigen(actionType: _)` dem Dispatcher entspricht.

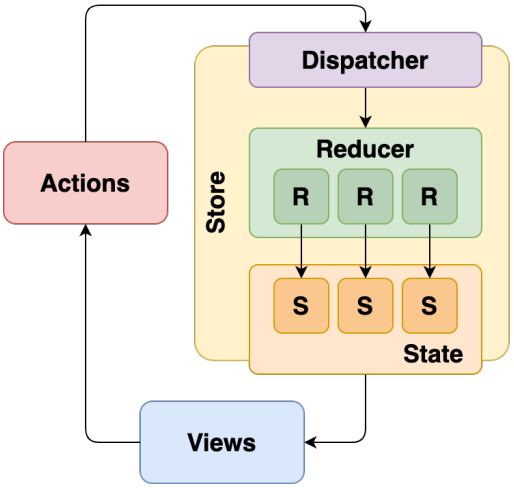


Abbildung 6.2: Schematisches Konzept für die Architektur Grundlage

6.2 Datensynchronisation

Dieses Konzept lässt sich auf die Synchronisation zweier gekoppelter Anwendungen mit gleicher Anwendungslogik und identischen States übertragen. Eine in Anwendung A ausgelöste Action gelangt zum Reducer bzw. zur Anwendungslogik und ändert den

State der Anwendung A. Sendet man von dort dieselbe Action über einen Kommunikationskanal zur Anwendung B, ändert sich auch deren State exakt gleich. Somit erzeugt man eine Synchronisation der States in zwei oder sogar mehreren Anwendungen, die in Raum und Zeit getrennt sind. Diese Struktur ist in Abbildung 6.3 schematisch dargestellt, wobei der gepunktete Pfeil einen Kommunikationskanal symbolisiert.

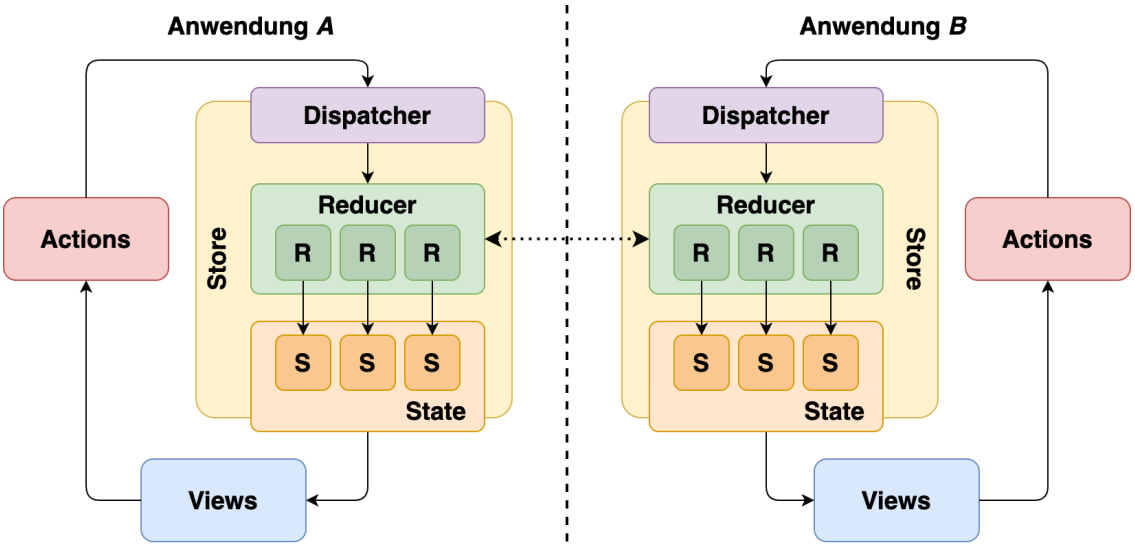


Abbildung 6.3: Schematisches Konzept der Architektur zum Synchronisieren

Eine korrekte Synchronisation ist jedoch von derselben Ausgangssituation abhängig. Wurden bei State A beispielsweise schon drei Actions vor der Kopplung ausgeführt, sind die Daten in den States A und B nicht mehr identisch. Um das Problem zu umgehen, gibt es zwei Möglichkeiten. Entweder es werden an den State B alle schon in A ausgeführten Actions in der gleichen Reihenfolge gesendet. Oder aber es wird der komplette State von A zu B gesendet, um die identische Ausgangssituation zu erhalten. Wichtig für das Senden des States und der Actions ist, dass sie serialisierbar sind. Serialisierung beschreibt den Mechanismus, Objekte in einen eindimensionalen Bitstrom zu überführen und diesen Bitstrom wieder in die ursprünglichen Objekte zu transformieren.[30] So können Actions und States in Form des Bitstroms über einen Transportmechanismus gesendet und empfangen werden.

Allerdings ist dieser Vorgang nicht auf identische Anwendungslogik sowie States und Actions beschränkt. Beispielsweise kann es sein, dass die Anwendung A extra Datenelemente im State enthält, die Anwendung B nicht benötigt. Um nun trotzdem jeweils den einen State in den andern zu überführen, verwendet man sogenannte *Mapper*, was auf Deutsch so viel wie Zuordner bedeutet. Die Aufgabe eines Mappers ist es, jedem Datenelement aus dem einen State ein Datenelement aus dem anderen zu zuordnen. Diese Zuordnung muss für zwei Zuordnungsrichtungen festgelegt sein. Entweder wird A in B überführt oder umgekehrt. Im Fall, dass der State A mit den zusätzlichen Da-

[30] Definition Serialisierung (12.07.2020, 12:30) - `https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization`

tenelementen in State B portiert wird, müssen die überschüssigen Datenelemente nicht weiter berücksichtigt werden, da sie in B nicht gebraucht werden. Möchte man aber den State B ohne die extra Datenelemente in den State A überführen, gibt es verschiedene Möglichkeiten, die fehlenden Werte zu ersetzten. Beispielsweise könnten Standardwerte oder die Werte, die aktuell im State A vorhanden sind, verwendet werden. Wichtig dabei ist, dass der Mapper, genau wie die States, individuell entwickelt werden muss, da er von den States abhängig ist.

Mit demselben Prinzip lassen sich auch Actions von einer Anwendung in andere Actions überführen. Beispielsweise kann eine Action von Plattform A eine Action von Plattform B auslösen. Darüber hinaus müssen sich die Actions nicht auf Benutzerinteraktionen beschränken, sondern können auch durch andere Events herbeigeführt werden. Wichtig beim Senden einer Action ist, dass nur der Name oder vielmehr eine Identifikation der Action und eventuell dazugehörige Werte übermittelt werden. Die Geschäftslogik, die beschreibt, wie die Action auszuführen ist, ist stets in der jeweiligen Anwendung hinterlegt und kann sich zwischen den Plattformen unterscheiden. Bei identischer Geschäftslogik auf allen Plattformen sind keine Probleme zu erwarten. Unterscheidet sich die Logik jedoch bei ein und derselben Action, ist der Entwickler für die Synchronität der States verantwortlich. Dieses Verhalten der Architektur ist erwünscht, damit sich die Plattformen ergänzen können, ohne dabei eine einzige Geschäftslogik zu teilen.

6.3 Spezifikationen für die Kommunikation

Durch das bisherige Konzept ist die Architektur flexibel genug, um sich jederzeit einmalig mit einem Pendant zu kombinieren. Jedoch bleibt offen, wie das System auf Verbindungsabbrüche oder andere Fehler reagiert. Als Vorbereitung dazu werden in diesem Abschnitt Spezifikationen für die Kommunikation definiert, um potenzielle Fehlerquellen von vornherein auszuschließen.

Mit Hilfe von sogenannten Kommunikationssitzungen sollen die Anwendungen in der Lage sein, einen Kommunikationskanal zu initialisieren, um sich miteinander zu koppeln. Diese Sitzungen (eng. Session) sind jedoch nicht im Rahmen der Architektur enthalten. Stattdessen stellt die Architektur universale Schnittstellen zur Verfügung, sodass verschiedenste Arten von Sitzungen mit denselben Grundfunktionen unterstützt werden können. Über die Schnittstellen kann die Architektur eine Session zum Koppeln starten oder stoppen, Daten versenden und empfangen. Des Weiteren muss die Architektur durch die Session über eine bestehende Kopplung, eventuelle Verbindungsabbrüche sowie aufgetretene Fehler über die Schnittstelle informiert werden. Darüber hinaus ist die Session für die Serialisierung aller Daten verantwortlich, sodass diese gesendet und empfangen werden können. Bei der Verbindung zwischen den Geräten soll es sich nach den Anforderungen aus Kapitel 4 um eine Peer-to-Peer-Verbindung handeln. Wie in Abschnitt 3.1 beschrieben, könnten dazu Bluetooth, W-LAN oder andere Funkverbin-

dungen verwendet werden. Abhängig von der Funktechnologie muss eine Session ein geeignetes Kommunikationsprotokoll nutzen. Es muss sicher stellen, dass Datenpakete in der gleichen Reihenfolge beim Empfänger ankommen, wie sie vom Sender losgeschickt wurden. Diese Einschränkung sorgt zwar dafür, dass alle Pakete in der richtigen Reihenfolge beim Anwendungspendant zugestellt werden. Allerdings kann es passieren, dass aufgrund von Latenz der Funkverbindungen, andere Actions in der Zwischenzeit ausgeführt wurden. Um dieses Problem zu erkennen, kommen zusätzliche Paketnummern sowie ein Protokoll zum Einsatz, welches in den Sessions als Grundfunktion zwingend notwendig ist. Der Entwickler muss sich dementsprechend selbstständig um die Implementierung der Sessions kümmern, die den hier genannten Anforderungen genügen.

Die in Abbildung 6.4 dargestellte Architektur ist um eine Session erweitert worden. Diese enthält den in Abschnitt 6.2 eingeführten Mapper sowie einen Serialisierer. Anhand der dicken roten Pfeile sowie Ziffern ist der Weg einer Action aus Anwendung A zu B ersichtlich. Kommt die Action im Reducer an, wird sie ausgeführt und an den Serialisierer weitergegeben (1.). Nun kann die Action als Bitstrom zur Session von Anwendung B gesendet werden (2.). Dort angekommen, wird der Bitstrom in eine Action decodiert. Anschließend kann die Action der Anwendung A in eine Action der Anwendung B durch den Mapper überführt werden (3.). Haben beide Anwendungen dieselben Actions, ist dieser Schritt nicht nötig. Zuletzt wird die Action an den Dispatcher weitergegeben (4.), damit diese dem richtigen Reducer zugeordnet werden kann.

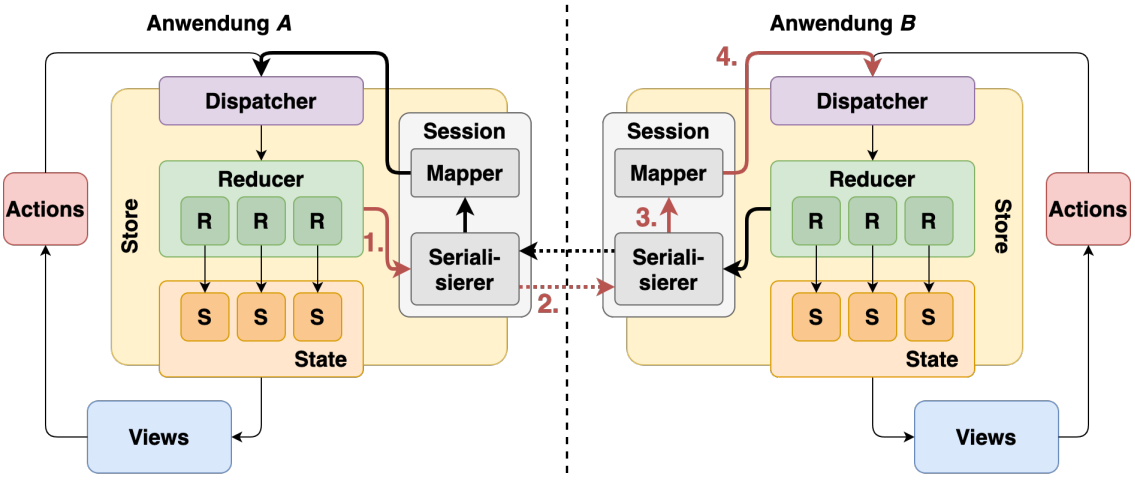


Abbildung 6.4: Schematisches Konzept der Architektur mit Session

6.4 Fehlererkennung bei der Datensynchronisation

Zur Realisierung der Paketnummer benötigen die jeweiligen Sessions einen fortlaufenden Zähler. Wie in Abbildung 6.5 zu sehen ist, wird jeder Zähler direkt nach der Kopplung auf den Startwert 0 gesetzt. Anschließend können beide Anwendungen Actions

senden. Exemplarisch möchte Anwendung A eine Action zu Anwendung B senden. Dabei gibt es folgenden Ablauf mit zwei Aktionen:

1. Der Zähler von Anwendung A wird um 1 erhöht.
2. Das Daten-Paket erhält den neuen Wert des Zählers als Paketnummer.

Während dieses Ablaufs darf der Zähler nicht anderweitig verändert werden, da eine höhere oder niedrigere Paketnummer zu Fehlern führt. Um das zu gewährleisten, muss der Ablauf atomar verlaufen. Das bedeutet, dass die Einzeloperationen als logische Einheit betrachtet werden und als Ganzes abläuft. Möchte man eine Action von Anwendung A zu Anwendung B senden, wird zuerst der Zähler von Anwendung A um 1 erhöht. Anschließend wird die Action in Form eines sogenannten Pakets mit dem Wert des Zählers A als Paketnummer an Anwendung B gesendet. Sobald Anwendung B das Paket erhält, wird die Paketnummer mit dem Wert des Zählers B verglichen. Die Paketnummer sollte nur genau um 1 größer sein, als der Wert des Zählers B. Ist das der Fall, ist kein Fehler aufgetreten. Die empfangene Action kann ausgeführt werden und der Zähler B muss um 1 erhöht werden. Konkret bedeutet der Unterschied von 1 zwischen Paketnummer und Zähler, dass die Actions in Anwendung A und in Anwendung B in der gleichen Reihenfolge ausgeführt wurden.

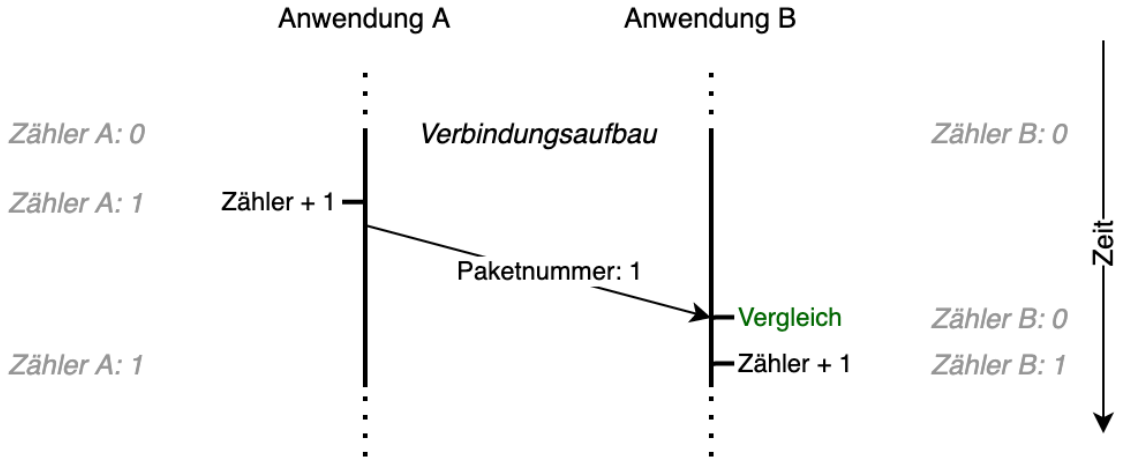


Abbildung 6.5: Fehlerfreier Ablauf beim Senden von Actions

Wäre der Wert des Zählers B gleich groß oder größer als die Paketnummer, würde es bedeuten, dass Anwendung B weitere Actions ausgeführt und gesendet hat und die Reihenfolge der Actions nicht gewährleistet ist. In Abbildung 6.6 ist dieses Problem beispielhaft veranschaulicht. Anwendung A und B senden eine Action zur jeweils anderen. Das Protokoll erkennt beim Vergleich der eingetroffenen Paketnummern, dass ein Konflikt aufgetreten ist, wodurch Reihenfolge der Actions bei beiden Anwendungen unterschiedlich sein kann. Allerdings ist das Protokoll nicht in der Lage, das Problem zu beheben.

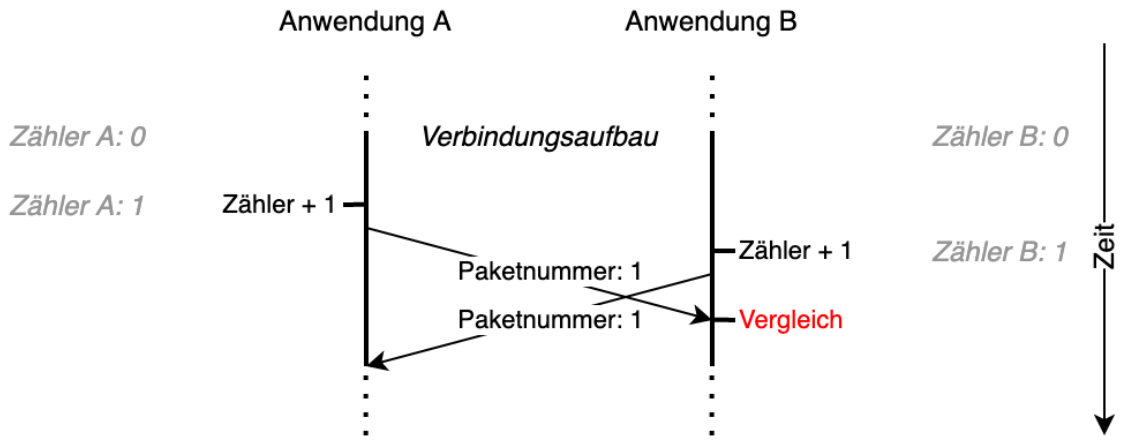


Abbildung 6.6: Auftritt eines Fehler beim Senden von Actions

6.5 Fehlerbehandlung bei der Datensynchronisation durch das Zeitreisekonzept

Wurde ein Fehler in der Reihenfolge der Actions entdeckt, kann dieser durch das sogenannte Zeitreisekonzept behoben werden. Dieses Konzept basiert darauf, dass nur Actions Änderungen im State auslösen können. Um die Reihenfolge von Actions zu ändern, ist es notwendig, alte States wieder herzustellen und darauf die Actions in der richtigen Reihenfolge auszuführen. Dazu werden alle ausgelösten Actions sowie deren jeweilige Ausgangszustände oder Ausgangs-States in einer chronologischen Liste gespeichert. Diese Zusammenstellung ist somit eine Art Zeitstrahl, der alle Änderungen am State mit den jeweiligen Actions verknüpft. In Abbildung 6.7 ist solch eine exemplarische Liste abgebildet. Die erste Spalte beinhaltet alle ausgeführten Actions, die zweite die jeweiligen Ausgangs-Zustände. In diesem Beispiel besteht der State aus einer einzigen Variable `zahl`, mit dem Startwert `0`. Des Weiteren gibt es drei Actions:

- `inkrementieren`: Addiert `zahl` mit Eins.
- `dekrementieren`: Subtrahiert von `zahl` Eins.
- `multiplizieren(2)`: Multipliziert den Wert von `zahl` mit Zwei.

Werden nun die Actions `inkrementieren`, `multiplizieren(2)` und `dekrementieren` in der Reihenfolge ausgeführt, ändert sich der State von `0` zu `1`, `2` und anschließend wieder zu `1`.

	Action	Ausgangs-State
1.	• inkrementieren	zahl = 0
2.	• multiplizieren(2)	zahl = 1
3.	• dekrementieren	zahl = 2
4.	• ...	zahl = 1

Abbildung 6.7: Exemplarische Historie

Wurde beispielsweise der Fehler gemeldet, dass die Action `dekrementieren` vor der Action `multiplizieren(2)` ausgeführt werden muss, kann nun das Zeitreisekonzept das Problem lösen. Dazu wird der aktuelle State mit dem Ausgangs-State von `multiplizieren(2)` überschrieben. Dieser ist in Abbildung 6.8 rot markiert. Anschließend werden die Actions in der richtigen Reihenfolge ausgeführt und wieder in der Liste mit den jeweiligen Ausgangs-States gespeichert. Beim Vergleich der States der Abbildungen 6.7 und 6.8 ist außerdem die Wichtigkeit der Reihenfolge der Actions zu erkennen. Je nachdem welche Action zuerst ausgeführt wurde, ist `zahl` am Ende eine Eins oder eine Null.

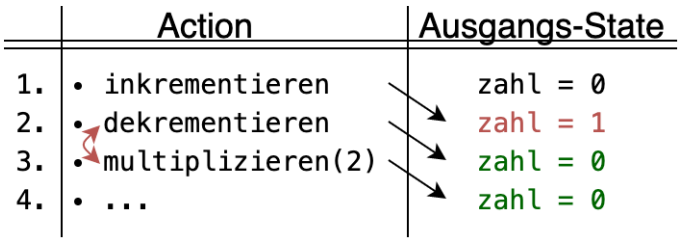


Abbildung 6.8: Exemplarische Historie mit geänderter Reihenfolge

Bei der Fehlererkennung und der Fehlerbehandlung durch das Zeitreisekonzept ist wichtig, dass nur eine der Anwendungen die Reihenfolge der Actions „korrigiert“, ansonsten bleibt der Fehler weiterhin bestehen. Aus diesem Grund muss vorher festgelegt werden, ob die Actions von Anwendung A oder B priorisiert werden.

Eine weitere Fehlerbehandlung bezieht sich auf Fehler des Kommunikationsprotokolls und auf Verbindungsabbrüche. Wenn eine Action nicht zugestellt werden kann oder andere Verbindungsfehler auftreten, müssen diese als Verbindungsabbruch behandelt werden. Dies hat den Hintergrund, dass die Anwendungen im Einzel-Modus weiter laufen können, ohne auf das Pendant zu warten. Danach sind die Anwendungen jedoch nicht mehr synchron. Um die Anwendungen wieder zu koppeln, muss zuerst sichergestellt werden, dass das jeweilige Pendant erreichbar ist. Anschließend müssen die States synchronisiert werden, sodass beide Anwendungen die gleiche Ausgangssituation haben. Dabei kommen die in Abschnitt 6.2 beschrieben Mapper zum Einsatz. Erst nachdem die States synchronisiert wurden, können neue Actions ausgeführt und gesendet werden. Mithilfe dieser Fehlerbehandlung ist es zwei Anwendungen möglich, sich jederzeit zu koppeln und zu trennen.

6.6 Zusammenfassung

Die in Abbildung 6.9 dargestellte Architektur besitzt sowohl die Fehlererkennung durch das Paketprotokoll innerhalb der Session, als auch die Fehlerbehandlung durch das Zeitreisekonzept. Durch dieses Konzept wird der reguläre Store der Anwendung in einen speziellen Zeitreise-Store verpackt, dieser bleibt dadurch jedoch unverändert. Grundlage dieser Schachtelung ist die in Abschnitt 6.1 beschriebene Baumstruktur. Ähnliches

gilt auch für die regulären Views und Actions. Diese werden ebenfalls in einen speziellen Zeitreise-Typen verpackt, wie in Abbildung 6.9 zu sehen ist.

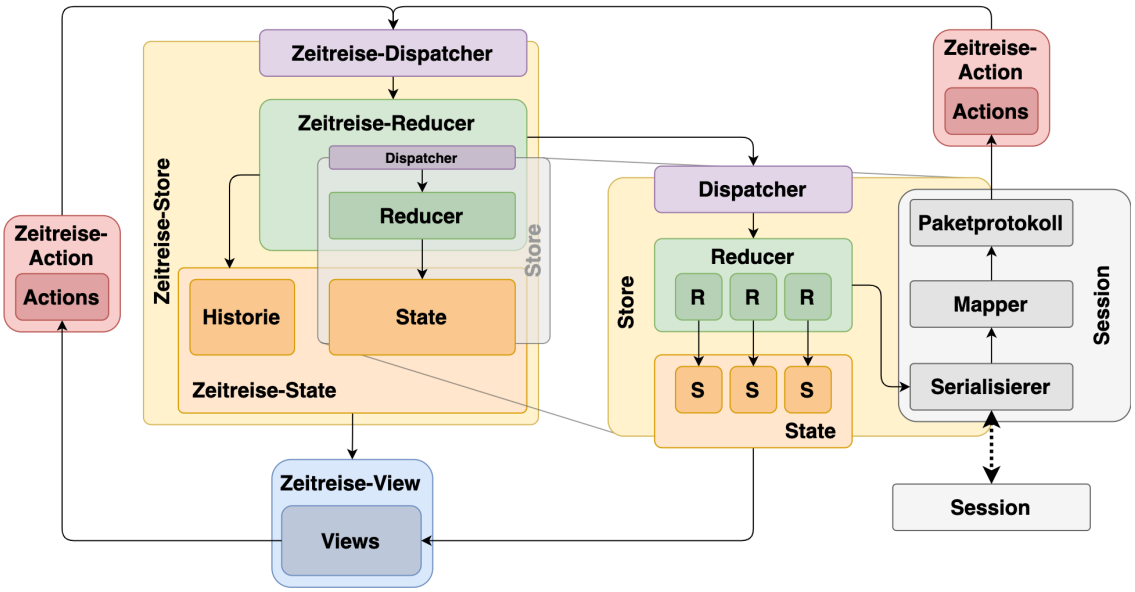


Abbildung 6.9: Schematische Abbildung der vollständigen Architektur

Konkret sieht ein exemplarischer Ablauf eines Programms mit dieser Architektur wie folgt aus:

- Eine auf den regulären Views ausgelöste Action, wird im Hintergrund in eine Zeitreise-Action verpackt. Das ist notwendig, da alle regulären Actions und State-Änderungen durch die Zeitreise erfasst werden müssen. Die verpackte Action gelangt über den Zeitreise-Dispatcher in den Zeitreise-Store, welcher in dieser Anwendung den Wurzel-Store der Anwendung darstellt.
- Innerhalb des Zeitreise-Reducers erfolgen drei Schritte:
  1. Die gekapselte Action wird untersucht. Dabei wird geschaut, ob es sich um eine reguläre Action handelt, die auf der View ausgelöst wurde oder ob die Action von der Session empfangen wurde.
  2. Da es sich im Beispiel um eine auf der View ausgelösten Action handelt, wird diese samt dem aktuellen regulären States in der Historie hinterlegt. So entsteht der in Abschnitt 6.5 beschriebene Zeitstrahl.
  3. Die Action wird an den regulären Dispatcher bzw. Store weitergegeben.
- Im Store nimmt die Action den regulären Ablauf, wie in Abschnitt 6.1 beschrieben. Besteht zudem eine Kopplung zu einer anderen Session, kommt zusätzlich der in Abschnitt 6.3 erläuterte Ablauf hinzu.

Wenn eine Action empfangen wird, ergibt sich folgender Ablauf:

- Nachdem der Bitstrom vom Serialisierer und Mapper in eine Action umgewandelt wurde, überprüft das Paketprotokoll die Paketnummer. Tritt kein Fehler auf, wird die Action wieder in eine Zeitreise-Action gekapselt und an den Zeitreise-Dispatcher geleitet.

- Innerhalb des Zeitreise-Reducers erfolgen abermals die drei Schritte aus dem vorherigen Ablauf. Jedoch unterscheidet sich der Letzte leicht. Würde die Action an den regulären Store gesendet, könnte diese Action erneut versendet werden, was eine endlose Abfolge an Actions zur Folge hat. Stattdessen wird die Action über eine spezielle Kopie des regulären Reducers ohne Verbindung zur Session ausgeführt.

Für den Fall, dass eine Action empfangen wird und das Paketprotokoll einen Reihenfolge-Fehler meldet, gibt es folgenden Ablauf:

- Stellt das Paketprotokoll einen Fehler fest, kann anhand der Differenz zwischen Paketnummer und dem Zähler in der Session ermittelt werden, wie viele Actions in der Zwischenzeit ausgeführt wurden. Diese Information sowie die empfangene Action werden in eine Zeitreise-Action gekapselt und an den Zeitreise-Dispatcher geleitet.
- Dort wird die gekapselte Action analysiert und folgende Schritte aus der in Abschnitt 6.5 beschriebenen Fehlerbehandlung veranlasst:
  1. Anhand der mitgelieferten Anzahl an Actions, die in der Zwischenzeit ausgeführt wurden, wird der aktuelle State auf den jeweiligen alten State zurückgesetzt.
  2. Alle Actions, die vom alten State bis hierhin ausgeführt wurden, werden zwischengespeichert und aus der Zeitstrahl-Liste samt Ausgangs-States entfernt.
  3. Die neue empfangene Action wird auf dem alten und nun aktuellen State ausgeführt und in den Zeitstrahl samt alten State gespeichert. Dabei kommt wieder die spezielle Reducer-Kopie ohne Session zum Einsatz.
  4. Jede der zwischengespeicherten Actions wird nach einander ausgeführt und samt Ausgangs-State zwischengespeichert.

  In der Zwischenzeit werden neue Actions auf der View blockiert. Dies ist möglich, da die View in einer Zeitreise-View ebenfalls gekapselt ist.
- Nach der Fehlerbehandlung werden alle Actions auf der View wieder freigegeben.

Ein ähnlicher Ablauf gilt auch für Verbindungsabbrüche oder andere Fehler, die als solche behandelt werden sollen:

- Dem Session-Pendant wird eine Action gesendet, um die States zu synchronisieren. Solange keine Antwort vom Pendant kommt, sind Actions über die View noch nicht blockiert.
- Nach Empfangen einer positiven Antwort werden Actions blockiert und der aktuelle State gesendet. Anschließend werden Actions wieder freigegeben. Kommt es zu einem Fehler, wird der Vorgang wiederholt, sobald die Session meldet, dass das Pendant erreichbar ist.
- Aufseiten des Pendants werden beim Erhalt des States, Actions auf der View blockiert.
- Mithilfe des Mappers wird der empfangene State auf den eigenen übertragen.

Hierbei ist anzumerken, dass die Synchronisation nur von einem Gerät aus erfolgt. Das hat den Hintergrund, dass aufgrund der Komplexität auf ein robusteres System verzichtet wurde. Es wäre sinnvoller, eine State-Synchronisation zu verwenden, wie sie in vielen Client-Server-Anwendungen zu finden ist.

7 Begrenzungen

In diesem Kapitel werden die Grenzen der Umsetzung genauer definiert. Dabei wird auf Hardware und Plattform eingegangen und deren Verwendung begründet. Darüber hinaus werden Probleme der Tracking-Verfahren und deren weitere Einschränkungen erläutert.

7.1 Hardware und Plattform

Bei der Umsetzung, Demonstration und Evaluierung kommt eine Apple Watch Series 4 (44 mm) und ein iPhone X als Alternative zur AR-Brille zum Einsatz. Dies hat mehrere Hintergründe. Zum einen sind die vorhandenen Frameworks im Apple-Ökosystem bei der Entwicklung dieser prototypischen Architektur zielführend, sodass der Fokus der Entwicklung auf der Architektur und nicht z. B. auf der Kommunikation oder der AR-Umsetzung liegt. Zum anderen bietet sich das iPhone X als idealer AR-Brillen-Ersatz an. Das Patent gibt Hinweise darüber, welche Hardware-Komponenten für eine AR-Brille wichtig sind. Darunter sind Sensoren für visuelle und nicht-visuelle Daten, Prozessoren und Datenverarbeitungselemente, wie CPU, GPU und RAM, aber auch Hardware zur Datenübertragung und Energieversorgung. Weiter werden Lautsprecher, verschiedene Display-Technologie und Bedienelemente, wie Knöpfe, Touchoberfläche, Mikrofone und Schieberegler erwähnt. Diese Vorgaben ähneln stark kommerziellen AR-Brillen, wie zum Beispiel der Holo Lens 2[31] von Microsoft. Vergleicht man beispielsweise die Holo Lens 2 mit dem iPhone X[32] bezüglich Hardware-Eigenschaften und anderen Komponenten, die für AR wichtig sind, fällt auf, dass die Geräte sich sehr ähnlich sind, wie man in Tabelle 7.1 sieht.

	iPhone X	**HoloLens 2**
	Display	
Technologie	5,8” All-Screen OLED Multi-Touch Display	Holographische Durchsichtlinsen (Wellenleiter)
	Konnektivität	
WiFi	ja	ja
Bluetooth	ja	ja
	Audio	
Mikrofone	ja	ja
Lautsprecher	ja	ja

[31] HoloLens 2 Spezifikationen (20.08.2020, 10:00) - https://www.microsoft.com/de-de/hololens/hardware

[32] iPhone X Spezifikationen (20.08.2020, 10:00) - https://support.apple.com/kb/sp770?locale=de_DE

	iPhone X	HoloLens 2
	Sensoren	
Videokamera	7 Megapixel, 1080p 30/60 fps	8 Megapixel, 1080p 30 fps
Trägheits-messgerät	Beschleunigungssensor, 3-Achsen Gyrosensor, Magnetometer	Beschleunigungssensor, 3-Achsen Gyrosensor, Magnetometer
Kopfverfolgung	nein	4 Kameras für sichtbares Licht
Blickverfolgung	nein	2 Infrarot-Kameras
	Menschliche Verständigung	
Handverfolgung & Interaktion	möglich ab iOS 14[33]	ja
Spracherkennung	ja, mit Internetverbindung	ja, mit Internetverbindung
Sprachassitenten	Siri	Windows Hello
	Umgebungswahrnehmung	
6 Freiheitsgrad-Verfolgung	ja	ja
Räumliche Kartierung	ja	ja

Tabelle 7.1: Vergleich von iPhone X und HoloLens 2

Natürlich ist die AR-Brille dem iPhone in fast allen AR-Bereichen weit voraus und auch die unterschiedliche Handhabung und Eingabemöglichkeiten sind nicht zu übersehen. Jedoch spielen diese Aspekte im Bezug auf die Architektur erst einmal keine Rolle, da die verschiedensten Eingabemöglichkeiten auf Actions und Wertänderungen heruntergebrochen werden können.

7.2 Smartwatch-Tracking

Tracking gehört zu den wichtigsten Bestandteilen für gutes AR. Wie sich aber zeigt, müssen hinsichtlich der Umsetzung einige Einschränkungen gemacht werden. Das Apple eigene Framework ARKit[34], welches diese Aufgaben übernimmt, hat zwei Tracking-Varianten, die sich für das Tracking der Smartwatch eignen: Bild- und (3D-)Objekt-Tracking. Andere Frameworks oder SKDs wie ARCore[35], Vuforia[36] oder Wikitude[37] sind aufgrund ihres Umfangs ungeeignet.

[33] Detecting Hand Poses with Vision (11.09.2020, 12:00) - `https://developer.apple.com/documentation/vision/detecting_hand_poses_with_vision`

[34] Dokumentation von ARKit (14.09.2020, 18:15) - `https://developer.apple.com/documentation/arkit/`

[35] ARCore Internetseite (22.08.2020, 10:00) - `https://developers.google.com/ar`

[36] Vuforia Internetseite (22.08.2020, 10:00) - `https://www.ptc.com/en/products/augmented-reality/vuforia`

[37] Wikitude Internetseite (22.08.2020, 10:00) - `https://www.wikitude.com/`

7.2.1 3D-Objekt-Tracking

Zuerst wird das Objekt-Tracking zum Erkennen und Tracken der Apple Watch untersucht. Es scheint zunächst die geeignetste Methode zu sein, da die Uhr ein dreidimensionales Objekt ist. Wie in Kapitel 5 beschrieben, benötigt man für 3D-Objekterkennung ein Referenz-Modell. Im Falle von ARKit wird für dieses Modell ein bestimmtes Format `.arobject` erwartet. Mit Hilfe einer von Apple zur Verfügung gestellten App, kann dieses Modell erstellt und für eigene Projekte exportiert werden.[38] Da die Objektscan-API öffentlich zugänglich ist, wäre es auch möglich, eine eigene Scan-App mit der gleichen Funktionalität zu entwickeln. Andere Wege, um solch ein Model zu erstellen gibt es allerdings nicht.

Beim Verwenden solch einer App, egal ob selbst oder von Apple programmiert, sind einige Dinge zu beachten:

- Das Objekt sollte mit einer Beleuchtungsstärke von 250 bis 400 Lux und von allen Seiten ausgeleuchtet sein.
- Die Lichttemperatur sollte in etwa 6500 Kelvin (D65) betragen, was laut der Internationalen Beleuchtungskommision (CIE) der Standardbeleuchtung Klasse D (Daylight) entspricht.[39]
- Das Objekt sollte beim Scannen einen matten, mittelgrauen Hintergrund haben.

Das Scannen von Objekten ist auch außerhalb dieser Spezifikationen möglich.

Apple hält zudem eine Anleitung zum Scannen von Objekten mit dieser App bereit. Zusammengefasst besteht diese aus sechs Teilschritten:

1. App auf dem iOS-Gerät installieren. (Das Gerät benötigt den A9 Prozessor oder einen neueren.)
2. Objekt so positionieren, dass keine anderen Objekte in das Kamerabild kommen und die oben genannten Spezifikationen beachten.
3. In der App soll nun das Objekt fokussiert werden. Anschließend muss ein dreidimensionaler Begrenzungskasten festgelegt werden, um der App mitzuteilen, an welcher Stelle im Raum sich das Objekt befindet und wie groß das Objekt ist. Außerdem soll darauf geachtet werden, dass nur Merkmale des Objekts im Begrenzungskasten sind.
4. Nun kann das Objekt gescannt werden. Dabei muss die Kamera um das Objekt herum bewegt werden, um es aus verschiedenen Winkeln zu betrachten. Schnelle und abrupte Bewegungen sollen vermieden werden. Die App hebt Teile des Begrenzungsrahmens hervor, um anzuzeigen, ob genug gescannt wurde, um das Objekt aus der entsprechenden Richtung zu erkennen.

[38] *ARKit Scanner* App von Apple (10.08.2020, 13:20) - `https://developer.apple.com/documentation/arkit/scanning_and_detecting_3d_objects`

[39] *ISO 11664-2:2007* (13.08.2020, 09:00) - `https://www.iso.org/obp/ui/#iso:std:iso:11664:-2:ed-1:v1:en`

5. Der Koordinatenursprung des Modells kann festgelegt werden, um beispielsweise AR-Modelle anhand des erkannten Objekts zu positionieren.
6. Das Modell kann getestet und exportiert werden. Beim Testen sollte das reale Objekt aus verschiedenen Winkeln, in unterschiedlichen Umgebungen und Lichtverhältnissen betrachtet werden, um zu überprüfen, ob ARKit seine Position und Orientierung zuverlässig erkennt.

Neben den Hinweisen aus der Anleitung und der Spezifikation für die Aufnahmeverhältnisse sind zwei weitere Faktoren entscheidend, die das Scannen der Apple Watch schwierig gestalten. Zum einen spielen die Maße eine entscheidende Rolle. Alle Dimensionen des Begrenzungskastens sollen mindestens $1\ cm$ lang sein und das Volumen des Begrenzungskastens mindestens $500\ cm^3$ betragen.[40] Zum anderen sind die Oberflächen-Materialien der Smartwatch ungeeignet, da diese überwiegend glänzen und reflektieren. Dadurch gibt es kaum geeignete Stellen mit hohem Kontrast, die für markerloses Tracking wichtig sind. Beim Versuch, die Uhr direkt am Handgelenk einzuscannen, um so einen größeren Begrenzungskasten zu benutzen, entstanden zu wenige Feature Points. Ursache ist vermutlich der Arm, der an zwei Enden des Begrenzungskatens herausragt. Um mindestens 100 Feature Points[40] zu erfassen, wurden deshalb farbige Referenzpunkten an der Smartwatch angebracht, die das Scannen verbesserten. Oft wurden jedoch auch Reflexionen auf dem Glas als Feature Points erkannt. Problematisch dabei ist, dass die Reflexionen nur bei einem bestimmten Blickwinkel zu sehen ist. Wird dieser geändert, verschieben sich dementsprechend auch die jeweiligen Feature Points, wodurch das Ergebnis des Modells beeinflusst wird. Des Weiteren muss sich das einzuscannende Objekt auf einem flachen Untergrund befinden, da der Begrenzungskasten nicht in der Luft platziert werden kann. Aufgrund der vielen Probleme und Einschränkungen konnte kein gutes Modell der Uhr angefertigt und exportiert werden. Stattdessen wurde die Verwendung des Bild-Tracking-Verfahrens angestrebt.

7.2.2 Bild-Tracking

Die zweite Methode - Bild-Tracking - erkennt nicht die Apple Watch, sonder die Inhalte auf dem Bildschirm, wobei Screenshots der Benutzeroberfläche als Referenzbilder dienen. Es besteht die Überlegung, bei Änderungen auf dem Display der Smartwatch ein Screenshot zu erstellen und an das AR-Gerät zu senden, sodass das Bild als neue Referenz verwendet wird. Allerdings ist es unter SwiftUI nicht möglich, Screenshots programmatisch auf der Apple Watch zu erstellen. Diese müssten per Hand für jede View gemacht und in der AR-App hinterlegt werden. Das hat auch zur Folge, dass dynamische Inhalte bei dieser Art von Tracking schwer umzusetzten sind. Wichtig ist auch, dass die Bilder hohe Kontraste und auffällige Stellen aufweisen, ähnlich wie es beim 3D-Objekt-Tracking der Fall ist. Außerdem sollten möglichst große Aufnahmen verwendet werden und die Maße der Originalbilder sollten bekannt sein. Da es sich um eine Apple Watch Series 4 mit einer Höhe von $44\ mm$ handelt, ist laut Spezifikation der Bildschirm

[40] Quelle: Fehlerbenachrichtigung bzw. Quellcode der App von Apple

$3\ cm$ breit und $3,6\ cm$ hoch.[41] Diese Werte werden von ARKit für die korrekte Berechnung der AR-Umgebung auf Distanz benötigt. Bei Tests mit Screenshot-Tracking war das Ergebnis deutlich besser. Auch aus sehr schrägen Blickwinkeln auf den Bildschirm konnte die AR-Umgebung dennoch exakt dargestellt werden. Selbst bei schnellen Bewegungen oder nur teilweiser Erfassung der Smartwatch durch die Kamera, wurde die AR-Umgebung angezeigt. Bei halber Verdeckung des Bildschirms durch z. B. einen Finger bricht das Tracking ab, was für intensive Benutzung ungeeignet ist. Zudem ist das Tracking wegen Reflexionen auf der Smartwatch und dem Auto-Fokus der Kamera, anfangs recht langsam. [29] Bei der Umsetzung des Demonstrators wurde schlussendlich darauf verzichtet, die AR-Komponenten in die Anwendung zu implementieren. Das hat den Hintergrund, dass ein deutlicher Mehraufwand entsteht, wenn man die Anwendung testen möchte. Die Simulatoren der Entwicklungsumgebung Xcode konnten nicht verwendet werden, da sie über keinen Kamerazugriff verfügen, der von AR zwingend benötigt wird. Stattdessen müssten echte Geräte zum Testen genutzt werden. Allerdings dauert das Starten der Anwendungen auf echten Geräten im Testmodus mehrere Minuten länger, da die Apple Watch nur eine drahtlose Verbindung besitzt und die komplette Anwendung zu jedem Teststart an die Smartwatch gesendet werden muss.

Bei der Entwicklung zukünftiger Anwendungen basierend auf dieser Architektur, sollte daher ein Simulator zum Einsatz kommen, der im Ausblick noch einmal aufgegriffen und genauer erläutert wird.

[41] Apple Watch Series 5 Spezifikation (09.09.2020, 10:30) `https://support.apple.com/kb/SP778?locale=de_DE`

8 Umsetzung

In diesem Kapitel wird zu Beginn der Aufbau bzw. die Struktur der Beispielanwendung beschrieben. Anschließend werden Architekturbausteine vorgestellt, die ein Anwender beim Verwenden der Architektur benötigt. Darauf aufbauend wird auf Implementierungsdetails des Zeitreisekonzepts sowie des Demonstrators eingegangen. Bei der Entwicklung der Software wurde ausschließlich Xcode[42] als Entwicklungsumgebung verwendet.

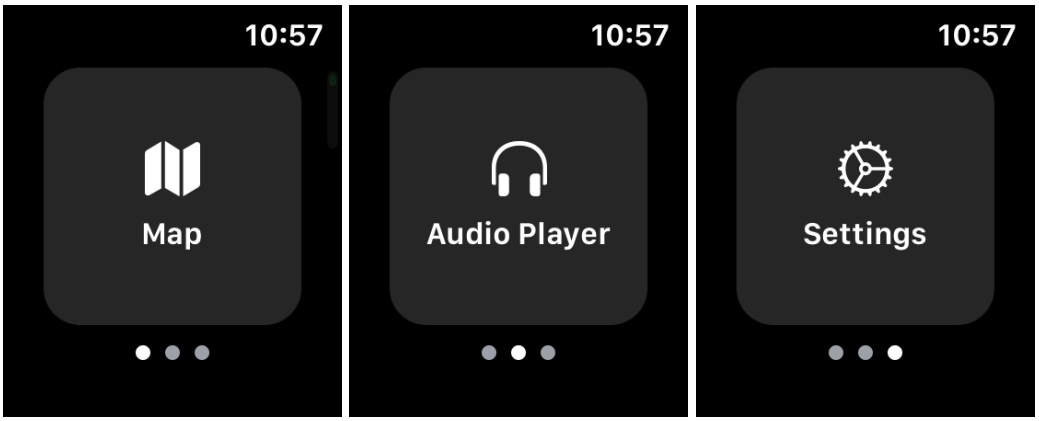
8.1 Aufbau der Beispielanwendung und ihre Anwendungsfälle

Ziel der Arbeit ist es, die entworfene und entwickelte Architektur zu evaluieren. Jedoch ist eine Evaluierung ohne das Vorhandensein einer konkreten Anwendung nicht praxisnah. Es werden deshalb Anwendungsfälle erstellt und in eine Anwendungsarchitektur integriert, welche für die Evaluation als Grundlage dienen. Bei der Auswahl der Anwendungsfälle wurden unterschiedliche Software-Qualitätsmerkmale berücksichtigt. Damit soll in der Evaluation eine umfangreiche Auswertung anhand der Software-Qualitätsmerkmale nach ISO/IEC 25010:2011 [2] möglich sein. Aus diesem Grund ist die Anwendungsarchitektur zugleich die Architektur zur Evaluation.

8.1.1 Hauptmenü

Die Beispielanwendung auf der Smartwatch besteht aus einem Hauptmenü (siehe Abbildung 8.1) und mehreren Detailansichten (siehe Abbildung 8.2). Durch seitliches Wischen oder durch das Drehen der Crown kann im Hauptmenü zwischen Map, Audio Player und Settings ausgewählt werden. Beim Drücken auf eines dieser Elemente wird zur jeweiligen Detailansicht navigiert. Einen ähnlichen Aufbau besitzt die iOS-Anwendung, die stellvertretend für eine AR-Anwendung steht. Wie in Abbildung 8.3 zu sehen ist, besteht die Smartphone-Anwendung ebenfalls aus einem Hauptmenü. Der Navigationstitel des Hauptmenüs ändert sich entsprechend dem Element, welches in der darunter liegenden Auswahl selektiert wurde. Dieses Auswahlmenü entspricht exakt der Auswahl auf der Smartwatch. Zudem haben beide Auswahlmenüs eine funktionale Gleichberechtigung bzw. Parität, sobald die Anwendungen gekoppelt werden. Das bedeutet, dass bei Auswahl des Menüpunkts Settings auf einem der Geräte, diese Auswahl auf dem anderen Gerät reflektiert wird. Auf der iOS-Anwendung ist es jedoch nicht möglich, die Detailansichten zu öffnen. Dieser Anwendungsfall dient hauptsächlich nur zur Überprüfung des Software-Qualitätsmerkmals Benutzbarkeit.

[42] Xcode (12.09.2020, 17:00) `https://developer.apple.com/xcode/`



(a) Auswahlelement Map (b) Auswahlelement Audio Player (c) Auswahlelement Settings

Abbildung 8.1: Hauptmenü auf der watchOS-Anwendung

8.1.2 Map

Dieser Anwendungsfall ist der Kartenanwendung des Patents nachempfunden (siehe Abbildung 3.4, 8.2a und 8.3a). Demzufolge sollen sich die Koordinaten der beiden Anwendungen bei einer Kopplung synchronisieren. Dadurch ergänzt die Smartphone-Anwendung den begrenzten Ausschnitt der Smartwatch-Anwendung. Dieses und auch die beiden weiteren Anwendungsbeispiele dienen der Überprüfung des Software-Qualitätsmerkmals Wartbarkeit sowie dessen Unterpunkt Wiederverwendbarkeit. Das Kriterium Wiederverwendbarkeit beschreibt den Grad, mit dem ein Objekt in mehr als einem System oder beim Bau anderer Objekte wiederverwendet werden kann.[43] Dabei ist die Idee entstanden, verschiedene Grade der Wiederverwendbarkeit auf der Architektur zu simulieren, um eine vielseitigere Evaluation zu ermöglichen. Infolge dessen wurden diese drei Grade definiert:

1. Grad: Dieser Fall repräsentiert den Extremfall, dass kein Quellcode wiederverwendet werden darf.
2. Grad: Der gesamte Quellcode eines Stores soll wiederverwendet werden, hierbei muss auf generische oder plattformabhängige Teile verzichtet werden.
3. Grad: Der gesamte Quellcode eines Stores soll wiederverwendet werden, wobei generische oder plattformabhängige Teile erwünscht sind.

Wegen des geringen Umfangs erscheint es hier sinnvoll, den 1. Grad zu demonstrieren.

8.1.3 Audio Player

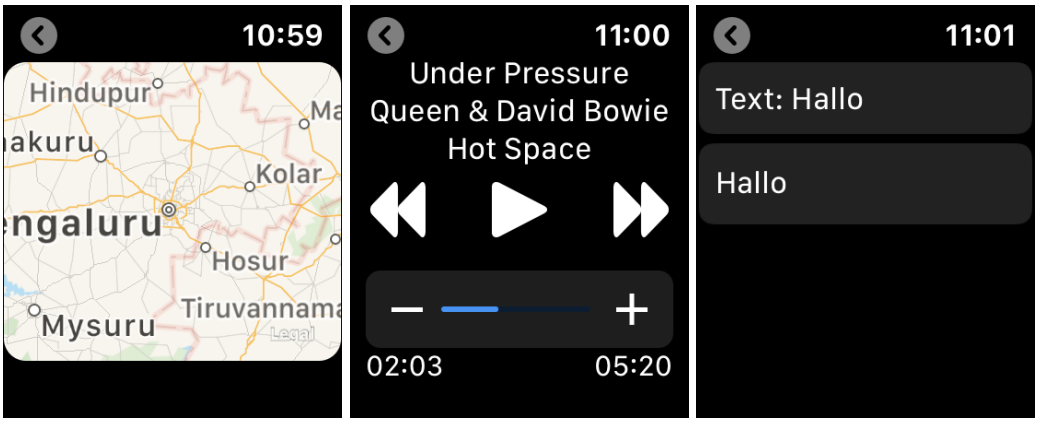
Dieser Anwendungsfall ist dem AR-Audio-Player des Patents nachempfunden (siehe Abbildung 3.3, 8.2b und 8.3b). Die Detailansicht dieser View, aber auch die Funktiona-

[43] Definition Wartbarkeit (10.09.2020, 16:20) - `https://www.iso.org/obp/ui/#iso_std_iso-iec_25010_ed-1_v1_en_term_4.2.7.2`

lität ist auf beiden Geräten identisch. Die obere Hälfte der View besteht aus Informationen zu dem aktuellen Song und die untere Hälfte enthält Komponenten zum Starten und Stoppen sowie Vor- und Zurückspulen des Songs. Im gekoppelten Zustand sollen die beiden Views synchron bleiben und die verbleibende Spieldauer des Songs richtig darstellen. Das Besondere an diesem Anwendungsfall ist, dass die Geschäftslogik eines Players in einem abgekapselten Objekt steckt. Um auf die Funktionen des Objekts zuzugreifen, stellt dieses eigene Schnittstellen zur Verfügung. Die Geschäftslogik der Anwendung muss demnach das Objekt über diese Schnittstellen ansprechen, um einen Song abzuspielen. Solch eine Aufteilung ist in vielen Anwendungen zu finden und somit wichtig für die Evaluierung der Architektur. Darüber hinaus wurde der Quellcode des Objektes auf beiden Anwendungen wiederverwendet. Dieser Anwendungsfall dient der Überprüfung der Software-Qualitätsmerkmale Zuverlässigkeit, Kompatibilität und Wiederverwendbarkeit vom Grad 2.

8.1.4 Settings

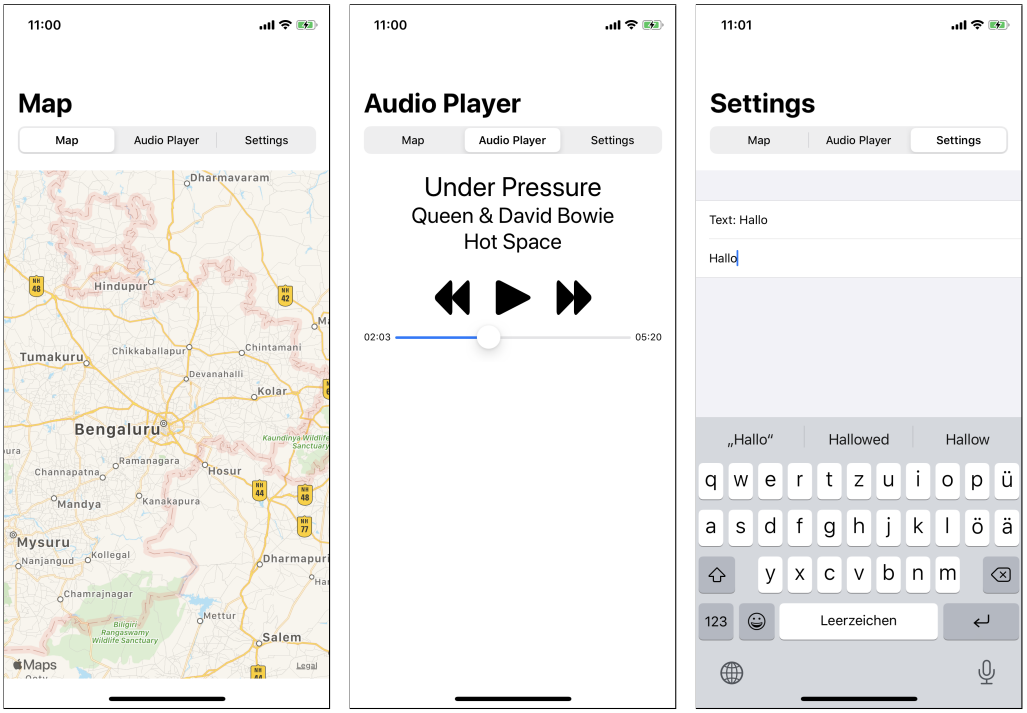
Wie auch beim Audioplayer ist diese Ansicht auf beiden Geräten identisch. Sie enthält ein einfaches Textfeld zur Eingabe und darüber den entsprechenden Text, welcher den Input im Textfeld wiedergibt (siehe Abbildung 8.2c und 8.3c). Sobald die Anwendungen gekoppelt sind, wird der Inhalt der Textfelder synchronisiert. Die Besonderheit besteht darin, dass die Eingabe auf der Uhr nicht über eine Tastatur erfolgt, sondern diktiert oder „gekritzelt“[44] werden kann. Durch diesen Anwendungsfall soll die Wiederverwendbarkeit vom Grad 3. sowie die Funktionsweise eines Textfelds unter iOS und watchOS repräsentiert werden.



(a) Map Detailansicht (b) Audio Player Detailansicht (c) Settings Detailansicht mit Eingabe in das Textfeld

Abbildung 8.2: Detailansichten auf der watchOS-Anwendung

[44] Wortherkunft von „kritzeln“ (11.09.2020, 12:30) - https://support.apple.com/de-de/guide/watch/apd92a90f882/6.0/watchos/6.0



(a) Map Detailansicht (b) Audio Player Detailansicht (c) Settings Detailansicht mit Eingabe in das Textfeld

Abbildung 8.3: Detailansichten auf der iOS-Anwendung

8.2 The Composable Architecture

Quellcode A.1 zeigt, dass das *state mangement pattern* in wenigen Codezeilen in Swift umsetzbar ist. Eine sehr ähnliche Architektur wurde für eine Anwendung verwendet, die im Praktikumsbericht [30] des Autors beschrieben wurde. Allerdings können komplexe geschachtelte API-Aufrufe oder andere Maschine-zu-Maschine-Kommunikation nur aufwendig mit solch einfache strukturierter Architekturgrundlage umgesetzt werden. Da jedoch ein großer Teil der Architektur sich auf der Kommunikation zwischen zwei Maschinen bezieht, sollte eine robustere Variante benutzt werden. Aus diesem Grund wurde bei der Umsetzung die Bibliothek *The Composable Architecture* (TCA)[45] herangezogen, welche in den folgenden Abschnitten genauer erläutert wird. Folgendene Themen werden dabei betrachtet:

- Entstehung des Frameworks,
- Konzept und dessen Vorteile gegenüber herkömmlichen SwiftUI,
- wichtige Bausteine, die essentiell für die Architektur sind und
- Funktionsweise des Frameworks anhand eines Beispiels.

[45] TCA Github Repository (11.09.2020, 10:50) - `https://github.com/pointfreeco/swift-composable-architecture/`

8.2.1 Hintergrund

TCA ist eine Open-Source-Software-Bibliothek zum Erstellen von konsistenten und verständlichen Applikationen unter besonderer Berücksichtigung von Struktur, Testmöglichkeiten und Ergonomie. TCA wurde ursprünglich auf einer Grundlage von Ideen anderer Konzepte und Technologien aufgebaut und in der Programmiersprache Swift umgesetzt. Inspiriert wurden Brandon Williams und Stephen Celis, die beiden Hauptentwickler von TCA, insbesondere von der Programmiersprache Elm[46] und der JavaScript Bibliothek Redux[47]. Der Auslöser für die Entwicklung von TCA ist allerdings SwiftUI.[48] Unter herkömmlichen SwiftUI ist es möglich, Zustände der Anwendung mit einfachen Werttypen zu verwalten und den Zustand auf viele Bildschirme zu verteilen, sodass Änderungen in einer View sofort auf einem weiteren Bildschirm betrachtet werden können. Dennoch stellten die beiden Entwickler einige Probleme bei der Entwicklung mit SwiftUI fest, für diese das Framework keine Lösungsansätze bereithält. Mit der Absicht die Probleme zu beheben, stellt TCA einige Werkzeuge zur Verfügung, die zur Erstellung von Anwendungen mit unterschiedlichem Zweck und unterschiedlicher Komplexität verwendet werden können.

8.2.2 TCA Konzept

Die Bibliothek gibt vor, wie eine Anwendung gebaut werden muss, um Vorteile gegenüber herkömmlichen SwiftUI-Anwendungen zu erhalten:

- **Modularisierung und Zusammensetzung:**
  Wie der Name schon verrät, ist die Architektur *composable*, was zusammensetzbar bedeutet. Durch wenige grundlegende Operatoren können kleine Komponenten der Anwendungslogik entwickelt und zu einer großen Komponente zusammengefügt werden. Umgekehrt kann, ein umfangreiches komplexes System in viele kleine, einfache Funktionen zerlegt werden. Weiter können Anwendungen auch in einzelne unabhängige Module aufgeteilt werden, sodass eine Anwendung vollständig modularisierbar ist. Das bedeutet, dass man Anwendungslogik komplett von der Benutzeroberfläche und von Plattformen isoliert erstellen und austauschen kann. Möchte man beispielsweise die Anwendung umfassend überarbeiten, muss nun nicht die gesamte Anwendungslogik auf einmal angepasst werden. Stattdessen ändert man Komponente für Komponente. Im Gegensatz dazu ist es unter herkömmlichem SwiftUI üblich, die Anwendungslogik direkt mit in die Views zu schreiben. Des Weiteren ist es zwar möglich, modulare Logik getrennt von Views zu entwickeln, das Zusammenfügen ist jedoch nur mit sehr großem Aufwand verbunden.

[46] Elm Internetseite (23.08.2020, 16:00) - `https://elm-lang.org/`
[47] Redux Internetseite (23.08.2020, 15:50) - `https://redux.js.org/`
[48] Transkript: *A Tour of the Composable Architecture: Part 1* (10.09.2020, 10:00) - `https://www.pointfree.co/episodes/ep100-a-tour-of-the-composable-architecture-part-1#t47`

- **Nebeneffekte oder Side Effects:**
  Unter Nebeneffekten oder Side Effects versteht man Außeneinwirkungen auf die Anwendung, also Änderungen am Zustand der App durch Einflüsse von außen. In den meisten Fällen sind Nebeneffekte schwierig zu handhaben und zu testen. Beispielsweise werden bei einer Wetter-App Wetterdaten von einem Server abgerufen. Treten dabei Fehler auf, kommt anstelle der erwarteten Daten ein Fehlercode zurück, der behandelt werden muss, da sonst die App womöglich abstürzt. Aber nicht nur Serveranfragen und -rückrufe sind Nebeneffekte. Auch das Empfangen von Messdaten eines Gyroskop-Sensors oder das Lesen und Schreiben einer Datei oder Datenbank stellen Nebeneffekte dar. TCA hat für Nebeneffekte einen speziellen Typen, den sogenannten `Effect`-Typ. Dieser kann einen Arbeitsauftrag kapseln, wodurch Nebeneffekte verständlich werden und nicht auf Testbarkeit und Modularität verzichtet werden muss.
- **Testen:**
  Nicht nur Nebeneffekte müssen getestet werden, sondern die gesamte Architektur bzw. Anwendung. Das grundlegende Konzept dabei ist, dass man zunächst eine Behauptung aufstellt und diese Behauptung dann auf Korrektheit überprüft. Dafür hat TCA eine Helfer-Funktion, die dem Benutzer ermöglicht, eine Reihe an Benutzeraktionen, wie das Tippen auf einer Schaltfläche oder die Eingabe in ein Textfeld, auszuführen. Diese Funktion zwingt den Tester, eine Behauptung im Bezug auf die Änderung des Anwendungszustandes aufzustellen. Stimmen alle Behauptungen und tatsächlichen Zustandsänderungen überein, gilt der Test als bestanden. Tritt dagegen ein Fehler auf, fällt der Test negativ aus. Ähnliches gilt auch beim Testen von Nebeneffekten. Hier zwingt die Helfer-Funktion den Tester sogar dazu, die genaue Ausgabe des `Effect`-Typs anzugeben, wodurch die Effekte gleichzeitig auf Vollständigkeit überprüft werden können. Insgesamt ist eine breite Testabdeckung möglich, die sowohl umfasst, wie sich der Zustand der Anwendung im Laufe der Zeit entwickelt, als auch die gegenseitige Beeinflussung der Effekte im gesamten System. Zudem kann garantiert werden, dass die Anwendungslogik erwartungsgemäß funktioniert.

8.2.3 Wichtige Bausteine der TCA

Um das Konzept umzusetzen, stellt die Bibliothek einige grundlegende Werkzeuge oder Bausteine zur Verfügung, die bei der Entwicklung von Komponenten der Anwendungslogik definiert werden müssen.

- **State:**
  Der State, auf Deutsch Zustand, ist eine Datenstruktur, welche alle relevanten Daten enthält, die bei der Ausführung von Logik und zur Darstellung der Benutzeroberfläche einer Komponente benötigt werden.
- **Action:**
  Action ist eine Ansammlung von Aktionen bzw. Vorgänge, die in einer Kompo-

nente auftreten können, wie zum Beispiel Benutzeraktionen, Benachrichtigungen, Ereignisse und so weiter.

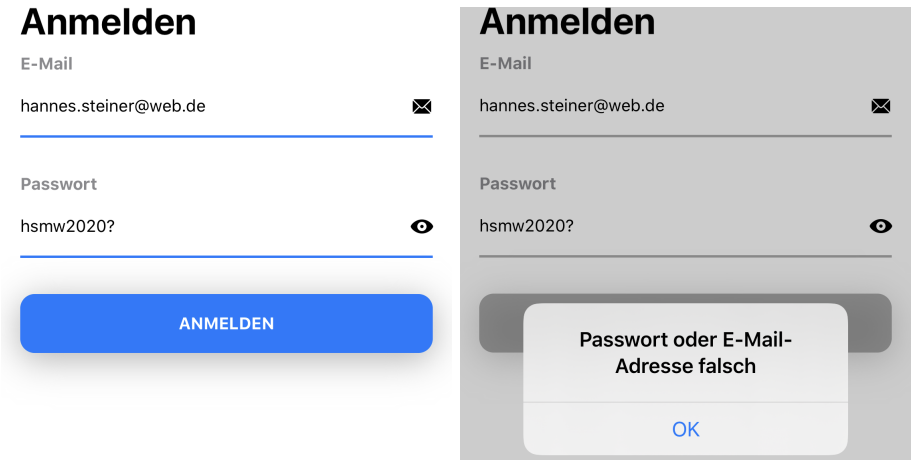
- **Environment:**
  Hierbei handelt es sich um eine Struktur, die sogenannte Abhängigkeiten bzw. Dependencies enthält. Abhängigkeit ist ein weit gefasster Begriff im Softwarebereich, der verwendet wird, wenn eine Software von einer anderen abhängig ist. [1] Demnach ist ein Environment, was auf Deutsch so viel heißt wie Kontext oder Umgebung, eine abstrakte Schnittstelle zu anderen Softwarekomponenten. Beispiele für solche Abhängigkeiten könnte ein sogenannter API-Client sein, der mit einem Server kommunizieren kann oder ein Sensor-Client, der Messwerte von einem Gyroskop-Sensor erhält.
- **Reducer:**
  Der Reducer ist eine Funktion, die den Übergang vom aktuelle Zustand in den nächsten Zustand der Anwendung, durch eine Action, beschreibt. Innerhalb des Reducers werden die Vorgänge aller Actions einer Komponente beschrieben. Der Reducer ist auch für das Ausführen von Nebeneffekten verantwortlich, die durch das Environment entstehen können.
- **Store:**
  Der Store ist die Umgebung, die eine Komponente steuert. Alle durch Benutzeraktionen ausgelösten Komponenten-Actions werden an den Store gesendet, damit dieser den Reducer und die `Effects` ausführen kann. Über den Store können auch die Zustandsänderungen im State beobachtet werden, die auf der Benutzeroberfläche reflektiert werden.

8.2.4 Funktionsweise anhand eines Beispiels

Für die folgenden Abschnitte des Kapitels werden Kenntnisse der Informatik zu Betriebssystemen [32] und Datentypen vorausgesetzt.

In diesem Abschnitt wird die Funktionsweise von TCA veranschaulicht. Dabei wird eine klassische Anmelde-Seite einer Anwendung als Beispiel verwendet, die in Abbildung 8.4a zu sehen ist. Dieser Teil der Anwendung besteht aus zwei Textfeldern für eine E-Mail und ein Passwort sowie einem Anmelde-Knopf. Nach Betätigen werden die eingegeben Daten an einen Server gesendet. Die Antwort des Servers kann entweder einen Fehler oder der Benutzername sein. Tritt ein Fehler auf, soll die Fehlermeldung dem Benutzer als Warnhinweis angezeigt werden.

In diesem Beispiel bietet es sich aufgrund des geringen Umfangs an Funktionen an, die View in einen State abzubilden. Das bedeutet, dass der State jeweils eine Variable für die E-Mail-Adresse und das Passwort besitzt. Des Weiteren muss es eine Variable für die Fehlermeldung des Servers geben. In diesem Fall ist die Variable durch ein Fragezeichen als optional gekennzeichnet, wie im Quellcode 8.1 zu sehen ist. Das be-



(a) `LoginView` der Anwendung (b) `LoginView` der Anwendung mit Warnhinweis

Abbildung 8.4: Exemplarische Views einer Anwendung mit TCA

deutet, dass solange die Variable `fehlerNachricht` keinen Wert hat (`nil`) auch kein Warnhinweis angezeigt wird.

```
1 struct LoginState: Equatable {
2   var email: String = ""
3   var passwort: String = ""
4   var fehlerNachricht: String?
5 }
```

Quellcode 8.1: `LoginState` der Anwendung

Um die Funktionen der View umzusetzen, benötigt es Actions. Die offensichtlichste Action ist das Drücken des Anmelde-Knopfes. Weiter gibt es jeweils eine Action zum Ändern des Textfeldeinhalts und eine zum Wegdrücken des Warnhinweises. Weniger offensichtlich ist dahingegen, dass eine Action beim Empfangen einer Serverantwort auftritt. Wie im Quellcode 8.2 ersichtlich ist, hat die Action `anmeldeClientAntwort` einen besonderen Datentyp `Result`. Auf diesen wird im Verlauf noch weiter eingegangen.

```
1 enum LoginAction: Equatable {
2   case loginButtonGedrueckt
3   case emailGeaendert(adresse: String)
4   case passwortGeaendert(passwort: String)
5   case anmeldeClientAntwort(Result<String, Error>)
6   case warnhinweisVerwerfen
7 }
```

Quellcode 8.2: `LoginAction` der Anwendung

Nun kann das Environment modelliert werden, welches im Quellcode 8.3 zu sehen ist. Um insbesondere eine Serverantwort zu empfangen, wird der `Effect`-Typ benötigt, der die Antwort des Servers kapselt. So ist diese Dependency eine Funktion, die eine E-Mail und ein Passwort als `String` entgegennimmt und einen `Effect<String, Error>` zurückgibt. Dabei ist der `String` im `Effect`, der vom Server gelieferte Benutzername. Der `Effect` erledigt seine Arbeit normalerweise in einem Hintergrund-Thread, um Ressourcen für wichtigere Aufgaben freizulassen. Jedoch werden Werte, die für die Benutzeroberfläche von Bedeutung sind, immer im Haupt-Thread abgelegt, sodass die View sich sofort erneuern kann. Um nun die Werte auf der Hauptwarteschlange empfange zu können, verwendet man einen sogenannten Scheduler. TCA stellt eine Klasse `AnyScheduler` zur Verfügung, die eine Besonderheit hat. Um die Anwendung später zu testen, kann statt dem echten Haupt-Scheduler auch ein selbstentwickelter Test-Scheduler verwendet werden, über den der Programmierer die volle Kontrolle hat.

```
1 public struct LoginEnvironment {
2   var mainQueue: AnySchedulerOf<DispatchQueue>
3   var anmeldung: (adresse: String, passwort: String) -> Effect<String, Error>
4 }
```

Quellcode 8.3: `LoginEnvironment` der Anwendung

Zusätzlich zum State, den Actions und dem Environment benötigt es den Reducer, der diese drei Bausteine verweint und die Geschäftslogik der Anwendung implementiert. Im Quellcode 8.4 ist die Struktur des `Reducer` zu sehen. Dabei fällt auf, dass dieser intern nur eine Funktion ist. Aus diesem Grund muss auch eine Funktion beim Initialisieren übergeben werden. Die Funktion des `Reducer`s verwendet drei Argumente: `State`, `Action` und `Environment`. Der `State` ist dabei mit `inout` gekennzeichnet, sodass sämtliche Änderungen des States direkt intern vorgenommen werden können. Ohne `inout` würde das `struct` als Kopie weitergegeben und die Änderungen deshalb nur an der Kopie ausgeführt werden. Des Weiteren ist es durch `@escaping` möglich, die Funktion sofort in den Initialisierer zu schreiben[49]. Dadurch kommt eine spezielle Schreibweise zustande, welche im Quellcode 8.5 in Zeile 2 zu sehen ist. Zudem muss der `Reducer` einen `Effect` zurückgeben, der typischerweise mithilfe von Objekten innerhalb des `Environments` erzeugt wird.[50] Wenn kein `Effect` ausgeführt werden muss, kann ein `.none-Effect` zurückgegeben werden.

```
1 public struct Reducer<State, Action, Environment> {
2
```

[49] Erklärung Escaping Closures (02.09.2020, 14:40) - https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546

[50] Dokumentation des TCA-Reducers (11.09.2020, 11:10) - https://github.com/pointfreeco/swift-composable-architecture/blob/main/Sources/ComposableArchitecture/Reducer.swift

```
3   private let reducer:
4     (inout State, Action, Environment) -> Effect<Action, Never>
5
6   public init(_ reducer:
7     @escaping (inout State, Action, Environment) -> Effect<Action, Never>
8   ) {
9     self.reducer = reducer
10  }
11 }
```

Quellcode 8.4: Codeauszug des TCA-`Reducers`

Wie im Quellcode 8.5 zu sehen ist, wird zu jeder Action beschrieben, wie der aktuelle State in den nächsten State versetzt werden kann und welche *Effects* ausgeführt werden müssen. Einigen Actions müssen jedoch keine `Effects` ausführen, weshalb diese `.none` zurückgeben. In den Zeilen 5 und 6 wird die Anmelde-Funktion aus dem Environment aufgerufen. Der Rückgabewert der Anmelde-Funktion wird in Zeile 7 auf der im Environment hinterlegten `mainQueue` entgegengenommen und weiter an die Action `LoginAction.anmeldeClientAntwort(Result<String, Error>)` übertragen. Diese Action wird anschließend in einen `Effect` gekapselt und zurück gegeben. Das bedeutet: der Reducer wird noch einmal mit der Action `LoginAction.anmeldeClientAntwort` ausgeführt. Die dazugehörige Geschäftslogik befindet sich in den Zeilen 16 bis 23. In Zeile 17 wird der mitgelieferte `Result`-Datentyp untersucht, da dieser entweder einen Erfolg oder einen Misserfolg mit jeweils einem zugehörigen Wert darstellt. Bei Erfolg bzw. `.success`, war die Anmeldung erfolgreich und der Benutzername kann ausgelesen werden. Bei einem Misserfolg bzw. `.failure` dagegen wird die Beschreibung des `Error`-Datentyp der `fehlerNachricht` im State übergeben, sodass der Warnhinweis erscheint. In Abbildung 8.4b ist eine mögliche Fehlerbeschreibung als Warnhinweis zu sehen.

```
1  let loginReducer = Reducer<LoginState, LoginAction, LoginEnvironment>
2   { state, action, environment in
3     switch action {
4       case .loginButtonGedrueckt:
5         return environment
6           .anmeldung(adresse: state.email, passwort: state.passwort)
7           .receive(on: environment.mainQueue)
8           .catchToEffect()
9           .map(LoginViewAction.anmeldeClientAntwort)
10      case .emailGeaendert(adresse: let email):
11        state.email = email
12        return .none
13      case .passwortGeaendert(passwort: let passwort):
14        state.passwort = passwort
15        return .none
16      case let .anmeldeClientAntwort(result):
17        switch result {
```

```
18          case .success(let benutzerName):
19            // Anmeldung erfolgreich
20          case .failure(let error):
21            state.fehlerNachricht = error.beschreibung
22        }
23        return .none
24      case .alertVerwerfen:
25        state.fehlerNachricht = nil
26        return .none
27    }
28  }
```

Quellcode 8.5: `LoginReducer` der Anwendung

Letztendlich werden alle aufgeführten Bausteine in einem `Store<LoginState, LoginAction>` zusammengeführt. Dieser steht hierbei stellvertretend für das Model aus der Abbildung 6.1b. Der Store wird in der View gespeichert, sodass Zustandsänderungen beobachtet und gerändert werden können. Wie in Abschnitt 6.1 erläutert, werden die Änderungen jedoch über das ViewModel beobachtet. Aus diesem Grund liefert TCA einen besonderen SwiftUI-ViewBuilder, der in Zeile 5 im Quellcode 8.6 zu sehen ist. Dieser `WithViewStore`-ViewBuilder nimmt den Store entgegen und liefert einen `viewStore`, welcher mit dem ViewModel vergleichbar ist. Über diesen ViewStore können die hierarchisch darunterliegenden Views ihre Daten aus dem Store binden, wie in den beiden Textfeldern anhand von `viewStore.binding(...)` zu sehen und die jeweiligen Actions auslösen. Um den Warnhinweis mittels `.alert(...)` anzuzeigen, ist ein zusätzliches identifizierbares `LoginAlert struct` nötig. Der Vorgang des Bindens ist auch hier der gleiche.

```
1  struct LoginView: View {
2    let store: Store<LoginState, LoginAction>
3
4    var body: some View {
5      WithViewStore(self.store) { viewStore in
6        VStack {
7          TextField("E-Mail"
8            text: self.viewStore.binding(
9              get: { $0.email },
10             send: LoginAction.emailGeaendert(adresse:)
11           )
12         )
13         TextField("Passwort"
14           text: self.viewStore.binding(
15             get: { $0.passwort },
16             send: LoginAction.passwortGeaendert(passwort:)
17           )
18         )
19         Button(action: {
```

```
20            self.viewStore.send(.loginButtonGedrueckt)
21          }) {
22            Text("Anmelden")
23            }
24        }
25        .alert(
26          item: viewStore.binding(
27            get: { $0.fehlerNachricht.map(LoginAlert.init(titel:)) },
28            send: .warnhinweisVerwerfen
29          ),
30          content: { Alert(titel: Text($0.titel)) }
31        )
32      }.navigationBarTitel("Anmelden")
33    }
34 }
35
36 struct LoginAlert: Identifiable {
37   var title: String
38   var id: String { self.title }
39 }
```

Quellcode 8.6: `LoginView` der Anwendung

Anhand dieses Beispiels wird ein wichtiger Aspekt von TCA deutlich. Da die Geschäftslogik getrennt von der View entwickelt wird, kann diese auf anderen Plattformen ohne Probleme wiederverwendet werden. Beispielsweise müsste lediglich die View für eine Desktop-Anwendung angepasst werden, was die Entwicklung von einer Anwendung für viele Plattformen erheblich vereinfacht.

8.3 Anwendungsseitige Implementierung des Zeitreisekonzepts

In diesem Abschnitt wird die Funktionsweise des Zeitreisekonzepts der Architektur anhand von Quellcode verdeutlicht. Zum besseren Verständnis ist in vielen Fällen der Originalquellcode ins Deutsche übersetzt und verkürzt wurden.

8.3.1 Zeitreise-State

Wie in Kapitel 6 erklärt, existiert ein Zeitreise-State, der in Quellcode 8.7 vereinfacht dargestellt ist. Infolge des Anspruchs, möglichst viel Quellcode wiederzuverwenden, ist dieser State generisch, wodurch er jede Art von State und Action unterstützen kann. Dabei müssen, wie in Zeile 1 zu sehen ist, State und Action *Equatable* sein. Hintergrund

dabei ist Performanceoptimierung im Zusammenhang mit SwiftUI[51].

Der `TimeTravelState` besitzt eine Variable `historie`, die die Liste aus Abschnitt 6.5 bzw. die Historie aus Abbildung 6.7 modelliert. Konkret bestehen die Elemente der Liste aus Tupeln `(ausgangsState: State, action: Action)`, die aus dem Ausgangsstate `State` und der ausgeführten `Action` zusammengesetzt sind. Die zweite Variable `aktuell` ist der aktuelle State, der durch das Zeitreisekonzept beeinflusst werden soll. Diese Variable entspricht dem regulären State auf Abbildung 6.9 und muss bei der Initialisierung des `TimeTravelState` mitgegeben werden. Die letzte Variable `sindActionsGeblockt` wird dazu verwendet, um reguläre Actions zu blockieren. Konkret wird in der Zeitreise-View (vgl. 6.9) die reguläre View mit einer andern View überlagert, sodass keine Actions ausgelöst werden können.

```
1 struct TimeTravelState<State: Equatable, Action: Equatable> {
2   var historie: [(ausgangsState: State, action: Action)] = []
3   var aktuell: State
4   ...
5   var sindActionsGeblockt: Bool = false
6 }
```

Quellcode 8.7: Darstellung des `TimeTravelStates`

8.3.2 Zeitreise-Actions

Wie ebenfalls in Kapitel 6 erläutert, werden alle regulären Actions in Zeitreise-Actions gekapselt. Um die Kapselung zu realisieren, müssen die Zeitreise-Action ebenso generisch sein, wie in Quellcode 8.8 zu sehen ist. Zudem benötigt der Zeitreise-Store nur die Action `case ausgeloesteAction(Action)`, da dieser keine weitere Funktionalität besitzen soll.

```
1 enum TimeTravelAction<Action: Equatable>: Equatable {
2   case ausgeloesteAction(Action)
3 }
```

Quellcode 8.8: Aufzählung der `TimeTravelActions`

8.3.3 Zeitreise-Reducer

Der Zeitreise-Reducer kapselt zwar die darunterliegende Anwendungslogik, wie in Abbildung 6.9 zu sehen ist, soll allerdings nur im Hintergrund der Anwendung wirken.

[51] TCA Equatable (10.09.2020, 14:10) - https://www.pointfree.co/collections/composable-architecture/a-tour-of-the-composable-architecture/ep100-a-tour-of-the-composable-architecture-part-1#t1080

Dementsprechend muss der reguläre Anwendungs-Reducer um eine Zeitreise-Funktion erweitert werden. Da der `Reducer` von TCA keine simple Funktion, sondern ein Datentyp ist, ist die Erweiterung möglich, wie anhand von `extension Reducer` in Zeile 1 von Quellcode 8.10 zu sehen ist. Jedoch ist zu beachten, dass sich die Funktionalität des Zeitreise-Reducers je nach Gerät unterscheiden soll, wie in Abschnitt 6.5 erläutert. Aus diesem Grund müssen die Rahmenbedingungen der Erweiterung exakt festgelegt werden. Dabei gibt der `Reducer` vor, dass es einen `State`, eine `Action` und ein `Environment` gibt, deren Datentyp definiert werden muss. Dementsprechend folgt `where State == MainMenuState, Action == MainMenuAction, Environment == MainMenuEnvironment { ... }`. Dabei sind `MainMenuState`, `MainMenuAction` und `MainMenuEnvironment` die verwendeten Datentypen des darunterliegenden Stores auf der watchOS-Anwendung. Gleichermaßen muss diese Erweiterung des `Reducers` auch für die iOS-Anwendung mit deren Datentypen für den `State`, `Action` und `Environment` entwickelt werden. Wenn sich die Datentypen jedoch nicht unterscheiden, kann das Verhalten des Zeitreise-Reducers durch sogenannte Kompilierungsbedingungen[52] geändert werden. Solch ein Beispiel ist im Quellcode 8.9 zu sehen.

```
1 #if os(iOS)
2   print("iOS")
3 #elseif os(watchOS)
4   print("watchOS")
5 #else
6   print("macOS oder tvOS")
7 #endif
```

Quellcode 8.9: Beispiel für Quellcode mit Compilerbedingungen

Darauf aufbauend wird die Funktion in Zeile 6 umgesetzt. `timeTravel()` hat keine Parameter, gibt allerdings einen `Reducer` zurück. Dieser wird später im Zeitreise-Store benötigt und aufgerufen. Zudem müssen, da ein `Reducer<State, Action, Environment>` generisch ist, die Typenparametern für `State`, `Action` und `Environment` festgelegt werden. In diesem Fall werden, wie in Zeile 7 und 8 zu sehen ist, für den `State` und die `Action` die `TimeTravel`-Typen verwendet. Deren Typenparameter wiederum werden beim Kompilieren durch `MainMenuState` sowie `MainMenuAction` ersetzt. Da der TimeTravel-Store kein eigenes `Environment` benötigt, kann für den Typenparameter das `MainMenuEnvironment` verwendet werden, was in Zeile 9 zu sehen ist.

In Zeile 11 wird nun der Reducer mit einer Funktion initialisiert und zurückgeben. Die Funktion besteht, wie in Unterabschnitt 8.2.4 erläutert, aus `State`, `Action` und `Environment` und enthält die Logik des Zeitreisekonzepts, welches in Kapitel 6 steht. Demzufolge wird zu jeder ausgelösten `Action` der Ausgangs-State und die Action in

[52] Conditional Compilation Block (09.09.2020, 09:00) - `https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539`

der Historie gespeichert, wie in Zeile 17 bis 19 dargestellt ist. Anschließend sollten alte Elemente aus der Historie entfernt werden, um Speicherplatz freizugeben. Erst nachdem der Ausgangs-State gespeichert ist, darf die `Action` ausgeführt werden. Dabei wird der reguläre Anwendungs-Reducer als Funktion aufgerufen, was anhand `let effect = self(...)` in Zeile 21 zu erkennen ist. Der Ausdruck `self(...)` kann deshalb verwendet werden, da der reguläre Reducer *selbst* erweitert wird. Die Parameter dieses Reducers sind wiederum der `MainMenuState`, die `MainMenuAction` und das `MainMenuEnvironment`. Der aktuelle `MainMenuState` ist dabei als Referenz unter `&timeState.aktuell` zu finden. Die `MainMenuAction` liegt gekapselt in der `.ausgeloesteAction()`-Action vor und das Environment kann direkt von den Attributen des Reducers übernommen werden. Nach dem Ausführen der Action entsteht ein `Effect<Action, Never>`, der als Variable `let effect` zwischengespeichert wird. Abschließend wird durch Zeile 22 `return effect.map(TimeTravelAction.ausgeloesteAction)` der `Effect<Action, Never>` in einen `Effect<TimeTravelAction <Action>, Never>` überführt und zurückgegeben, da der Zeitreise-Reducer nur diese eine Art von `Effect` zurückgibt.

Zudem ist in Zeile 14 bis 16 zu sehen, dass zu regulären Actions weiterer Code ausgeführt werden kann. Beispielsweise müssen dort alle Session-Fehler behandelt werden, da der Zugriff auf die Historie nur von hier aus möglich ist. Speziell die Logik aus Abschnitt 6.5 ist an dieser Stelle implementiert.

```
1  extension Reducer where State == MainMenuState, Action == MainMenuAction,
   ↪  Environment == MainMenuEnvironment {
2
3    func timeTravel() -> Reducer<
4      TimeTravelState<State, Action>, // State
5      TimeTravelAction<Action>,       // Action
6      Environment                     // Environment
7    > {
8      return Reducer { timeState, timeAction, environment in
9        switch timeAction {
10         case let .ausgeloesteAction(anwendungsAction):
11           switch anwendungsAction {
12            // Fehlerbehandlung Session-Actions
13           }
14           timeState.historie.append(
15             (ausgangsState: timeState.aktuell, action: anwendungsAction)
16           )
17           // alte elemente aus der Historie loeschen
18           let effect = self(&timeState.aktuell, anwendungsAction, environment)
19           return effect.map(TimeTravelAction.ausgeloesteAction)
20       }
21     }
22   }
23 }
```

Quellcode 8.10: Aufbau der `timeTravel`-Funktion

8.3.4 Zeitreise-View

Die Zeitreise-View soll, wie in Kapitel 6 erläutert, an oberster Stelle der View-Hierarchie stehen, damit die Actions auf tieferliegenden Views blockiert werden können. Dementsprechend wurde die Initialisierungsfunktion so entwickelt, dass die `TimeTravelView` einen initialen State, einen Reducer und ein Environment der regulären Anwendung entgegennehmen und in die Zeitreise-Logik überführen kann. Nach der Initialisierung gibt die `TimeTravelView` den passenden Store für die reguläre Haupt-View zurück, sodass das Zeitreisekonzept im Hintergrund immer mitläuft und den regulären Store beeinflussen kann. Quellcode 8.11 zeigt diesen Aufruf der `TimeTravelView`. Innerhalb der `MainMenuView` ändert sich dadurch nichts. Demzufolge lassen sich fertige Anwendungen, die mit TCA und SwiftUI entwickelt wurden, um diese Konzept erweitern.

```
1 TimeTravelView(
2   initialState: MainMenuState(),
3   reducer: mainMenuReducer,
4   environment: MainMenuEnvironment()
5 ) { store in
6   MainMenuView(store)
7 }
```

Quellcode 8.11: Verwendung der `TimeTravelView`

Die interne Umsetzung der `TimeTravelView` ist im Quellcode 8.12 zu sehen. Dabei ist in Zeile 1 zu beachten, dass lediglich der `Content` generisch ist, wobei es sich um eine Struktur vom Typ `View` handeln muss. Das bedeutet, dass die View beispielsweise für Testzwecke ausgetauscht werden kann. Alle anderen Parameter sind nicht generisch, da die `Reducer`-Erweiterung `timeTravel()` in Zeile 13 ebenfalls nicht generisch ist. Demnach muss die View für die iOS-Anwendung mit den jeweiligen Datentypen extra erstellt werden.

Bei der `init`-Funktion ist in den Zeilen 11 bis 15 zu sehen, wie der Zeitreise-Store aufgebaut wird. In Zeile 12 wird eine TimeTravelState erzeugt, deren Variable `aktuell` den initialen State erhält. Diese Variable wird später zum Erzeugen des Stores in Zeile 24 wiederverwendet. Der Reducer für den Zeitreise-Store besteht aus dem regulären Reducer, welcher um die `.timeTravel`-Funktion erweitert wurde. Das Environment bleibt, wie in Unterabschnitt 8.3.3 beschrieben, der gleiche.

In den Zeilen 23 bis 26 wird der reguläre Store für die View erzeugt und sozusagen rausgegeben. Dies ist durch eine spezielle Funktion `.scope(state: _, action: _)`

von TCA möglich. Sie dient dazu, einen lokalen Store aus einem globalen oder übergeordneten Store zu erzeugen. Die Zeile `state: \TimeTravelState.aktuell` bedeutet dabei, dass der State für den Store an der Stelle `TimeTravelState.aktuell` zu finden ist. Die darauffolgende Zeile `action: \TimeTravelAction.ausgeloesteAction` beschreibt die Action, die zur Kapselung der lokalen Actions verwendet wird. Die restlichen Zeilen im `body` der `TimeTravelView` haben die Funktion, lokale Actions zu blockieren.

```
struct TimeTravelView<Content: View>: View {
  private let timeTravelStore: Store<TimeTravelState<MainMenuState,
   ↪ MainMenuAction>, TimeTravelAction<MainMenuAction>>
  private let content: (Store<MainMenuState, MainMenuAction>) -> Content

  init(
    initialState: MainMenuState,
    reducer: Reducer<MainMenuState, MainMenuAction, MainMenuEnvironment>,
    environment: MainMenuEnvironment,
    @ViewBuilder content: @escaping (Store<MainMenuState, MainMenuAction>) ->
     ↪ Content
  ) {
    self.timeStore = Store<TimeTravelState<MainMenuState, MainMenuAction>,
     ↪ TimeTravelAction<MainMenuAction>>.init(
      initialState: TimeTravelState(aktuell: initialState),
      reducer: reducer.timeTravel(),
      environment: environment
    )
    self.content = content
  }

  var body: some View {
    WithViewStore(self.timeStore) { viewStore in
      ZStack {
        self.content(
          self.timeStore.scope(
            state: \TimeTravelState.aktuell,
            action: TimeTravelAction.ausgeloesteAction
          )
        ).disabled(!viewStore.sindActionsGeblockt)
        if !viewStore.sindActionsGeblockt {
          // Einblendung oder Ueberlagerung
        }
      }
    }
  }
}
```

Quellcode 8.12: Struktur der `TimeTravelView`

8.4 Implementierung einer Session

Im Kapitel 6 wurde erläutert, dass die Architektur eine Schnittstelle für verschiedenste Sessions bereitstellt. Diese Schnittstelle ist der sogenannte `SessionClient`, welcher im Quellcode 8.13 zu sehen ist. Dabei hat ein `SessionClient` die öffentliche Funktionen `start() -> Effect<SessionAction, Never>`, `senden(action: _) -> Effect<Never, Never>` und `syncAusloesen(state: _) -> Effect<Never, Never>`, deren Funktionsweise jedoch noch umgesetzt werden muss. Das hat den Hintergrund, dass so die Architektur immer dieselben Schnittstellen anspricht, die Funktionsweise der Session sich aber ändern kann. Beispielsweise können somit neben verschiedenen Sessions mit unterschiedlichen Kommunikationsprotokollen auch `SessionClient`-Attrappen erstellt werden, wie im Quellcode A.2 dargestellt.

```
1  public struct SessionClient {
2    enum SessionAction: Equatable {
3      case erhalteAction(MainMenuAction)
4      case erhalteActionUndError(action: MainMenuAction, position: Int)
5      case erhalteError(Error)
6      case erhalteState(MainMenuState)
7    }
8
9    private var erstellen: () -> Effect<SessionAction, Never>
10   private var abschicken: (PendantAction) -> Effect<Never, Never>
11   private var sync: (MainMenuState) -> Effect<Never, Never>
12
13   func start() -> Effect<SessionAction, Never> {
14     self.erstellen()
15   }
16
17   func senden(action: PendantAction) -> Effect<Never, Never> {
18     return self.abschicken(action)
19   }
20
21   func syncAusloesen(state: MainMenuState) -> Effect<Never, Never> {
22     return self.sync(state)
23   }
24 }
```

Quellcode 8.13: Struktur des `SessionClients`

Zudem besitzt ein `SessionClient` Actions, wie in den Zeilen 2 bis 7 im Quellcode 8.13 zu sehen ist. Diese sollen vom SessionClient verwendet werden, um den State zu ändern. Exemplarisch für die Nutzung dieser Actions ist im Quellcode 8.14 ein Auszug eines verwendeten `SessionClients` zu sehen. Die Funktion `erstellen()` hat die Aufgabe, eine Session zu initialisieren (8.14 Zeile 5), die asynchronen Rückrufe der Session in `SessionActions` zu verpacken und an den Store zurückzugeben. Aus diesem Grund

ist der Rückgabewert dieser Funktion `Effect<SessionAction, Never>` (8.14 Zeile 3). Da eine unbestimmte Anzahl an Rückrufen zu erwarten ist, wird die TCA-Funktion `.run()` verwendet. Diese Funktion initialisiert zu jedem Rückruf einen `Effect`, welcher im Store empfangen werden kann.

```
1  extension SessionClient {
2    static let live = SessionClient(
3      erstellen: { () -> Effect<SessionAction, Never> in
4        .run { effect in
5          let session = Session { rueckruf in
6            if let state = rueckruf["state"] {
7              effect.send(.erhalteState(state as! MainMenuState))
8              return
9            }
10           guard let action = rueckruf["action"] else {
11             let error = rueckruf["error"] as! SessionError
12             effect.send(.erhalteError(error))
13             return
14           }
15           if let position = rueckruf["number"] {
16             effect.send(.erhalteActionUndError(
17               action: action as! MainMenuAction,
18               position: position as! Int)
19             )
20           } else {
21             effect.send(.erhalteAction(action as! MainMenuAction))
22           }
23         }
24         gemeinsameSession = session
25         return AnyCancellable { /* Aufraumen */ }
26       }
27     },
28     abschicken: { ... },
29     sync: { ... }
30   )
31 }
```

Quellcode 8.14: `SessionClient` mit Beispiel Funktionen

Bei der Initialisierung der Session muss ein sogenannter completion handler angegeben werden, der die Rückrufe auf deren Inhalt untersucht und anschließend in `SessionActions` kapselt. Dieses Vorgehen ist in den Zeilen 6 bis 22 zu sehen. Enthält der Rückruf einen State, soll dieser in der `SessionAction .erhalteState(_)` verpackt werden (8.14 Zeile 6 bis 9). Ähnliches gilt für gesendete Actions und Fehler. In den Zeilen 15 bis 20 kann beispielsweise der Rückruf auch eine Position enthalten. Diese Zahl ist der Wert des Paketprotokolls, welcher angibt, wie viele Actions in der Zwischenzeit ausgeführt wurden. Damit kann im Zeitreise-Store die richtige Reihenfolge wieder hergestellt werden.

Im Anschluss an die Initialisierung der Session wird diese in einer globalen Variable `gemeinsameSession` gespeichert (Zeile 24). Das hat den Hintergrund, dass die Session in den Funktionen `abschicken: { ... }` und `sync: { ... }` wiederverwendet werden muss (8.14 Zeile 28, 29).

Zur Umsetzung der Session ist eine `WCSession`[53], für die Kommunikation verwendet wurden. Sie initiiert die Kommunikation zwischen einer WatchKit-Erweiterung, also der watchOS-Anwendung und ihrer iOS-Begleitanwendung. Zudem hält sie die Spezifikationen aus Abschnitt 6.3 ein[53].

Zudem wurden spezielle Actions zum Senden bzw. Empfangen entwickelt, welche mit JSON[54] serialisiert werden können. Es ist zwar möglich, dass man genau die Actions sendet, die auch standardmäßig in der Geschäftslogik verwendet werden. Allerdings stellte sich bei der Entwicklung heraus, dass es Vorteile hat, zusätzliche spezielle Actions zum Senden zu benutzen.

Beispielsweise kann dadurch zwischen empfangenen und vom Benutzer ausgelösten Actions unterschieden und deren Geschäftslogik angepasst werden. Des Weiteren sollten die Actions, die nur spezifisch für eine Plattform sind, nicht gesendet werden. Um dementsprechend die Schnittstellen zu optimieren, ist es sinnvoll, spezielle Actions zum Senden zu verwenden.

8.5 Entwicklung eines Mappers

Da die Beispielanwendung vielerlei Anwendungsfälle zur Demonstration abdecken und ihre flexible Funktionalität präsentieren sollte, wurde darauf verzichtet, alle States identisch zu strukturieren. Dadurch konnte Quellcode zwar weniger oft wiederverwendet, die Funktionsweise des Mappers aber besser gezeigt werden. Zur Demonstration des Mappers wurden die beiden Hauptmenü-States in den Anwendungen an einer Stelle unterschiedlich strukturiert, wie in Quellcode 8.15 zu sehen ist.

Die erste Variable `auswahl` ist in beiden States funktional gleich und modelliert, welche Ansicht in den beiden Menüs aktuell ausgewählt ist. Dabei wurde ein `enum MainMenuView` verwendet, welcher in Zeile 17 von 8.15 zu finden ist. Dieser `enum`-Datentyp besteht aus drei Zuständen, welche die drei Detailansichten repräsentieren. `auswahl` kann zu jeder Zeit nur genau einen dieser drei Zustände annehmen.

Die zweite Variable `sichtbareView` in Zeile 3 modelliert, *ob* und *welche* der Detailansichten gerade geöffnet ist. Der Initialwert ist hier `nil`. Infolgedessen ist keine De-

[53] Dokumentation WCSession (04.09.2020, 08:40) - `https://developer.apple.com/documentation/watchconnectivity/wcsession`

[54] The JSON Data Interchange Syntax (ECMA-404, ISO/IEC 21778:2017) (04.09.2020, 09:30) - `https://www.ecma-international.org/publications/standards/Ecma-404.htm`

tailansicht, dafür das Hauptmenü zu sehen. Jedoch kann es auch sinnvoll sein, das *„Ob“* und das *„Welche“* nicht in ein und derselben Variablen zu modellieren. Wie in den Zeilen 10 bis 12 ersichtlich ist, hat hierbei jede Detailansicht eine eigene Variable, welche angibt, ob diese Detailansicht zu sehen ist oder nicht. Ist keine geöffnet, steht dementsprechend das Hauptmenü im Vordergrund. Bei dieser Modellierung ist jedoch zu beachten, dass zwei oder auch alle Detailansichten sichtbar sein könnten. Solche Ausnahmen müssen in der Geschäftslogik und in der View berücksichtigt werden. Dieses Beispiel dient jedoch, wie schon erwähnt, der Demonstration des Mappers und der Architektur.

```
1  struct iOSMainMenuState: Equatable {
2    var auswahl: MainMenuView = .map
3    var sichtbareView: MainMenuView? = nil
4    ...
5  }
6
7  struct watchOSMainMenuState: Equatable {
8    var auswahl: MainMenuView = .map
9    var istMapViewSichtbar: Bool = false
10   var istAudioPlayerSichtbar: Bool = false
11   var istSettingsViewSichtbar: Bool = false
12   ...
13 }
14
15 enum MainMenuView: Equatable {
16   case map
17   case player
18   case settings
19 }
```

Quellcode 8.15: Vergleich der beiden `MainMenuStates`

Möchte man nun den `iOSMainMenuState` in den `watchOSMainMenuState` überführen, kann man den Mapper so gestallten, wie im Quellcode 8.16 zu sehen ist. Für die Sichtbarkeit der Detailansichten wird den drei Variablen mithilfe des bedingten Ternär-Operators[55] ein Initialwert zugeordnet, der abhängig von der Variable `sichtbareView` ist.

[55] Ternary Conditional Operator (04.09.2020, 10:30) - `https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID71`

```
1 extension watchOSMainMenuState {
2   public static func mapWatchOSMainMenuState(von empfangenerState:
    ↪   iOSMainMenuState) -> watchOSMainMenuState {
3     return MainMenuState(
4       auswahl: empfangenerState.auswahl,
5       istMapViewSichtbar:
6         empfangenerState.sichtbareView == .map ? true : false ,
7       istAudioPlayerSichtbar:
8         empfangenerState.sichtbareView == .player ? true : false ,
9       istSettingsViewSichtbar:
10        empfangenerState.sichtbareView == .settings ? true : false ,
11      ...
12    )
13  }
14 }
```

Quellcode 8.16: Mapper-Funktion von `iOSMainMenuState` in `watchOSMainMenuState`

Für den Fall, dass man den `watchOSMainMenuState` in den `iOSMainMenuState` überführen möchte, muss man folglich die Zuordnung rückgängig gestalten, sodass aus den drei `ist...Sichtbar`-Variablen eine `sichtbareView`-Variable entsteht, die bei der Initialisierung des `iOSMainMenuState` verwendet werden kann.

8.6 Hauptmenü

Der Schwerpunkt des Hauptmenüs ist einen Anwendungsfall zu finden, der speziell die Bedienbarkeit untersucht. Konkret wird geschaut, wie sehr die Architektur dieses Kriterium einschränkt. Aus diesem Grund soll die Auswahl des Menüs auf verschiedenste Art und Weise geändert werden. Zum einen kann eine horizontale Wischgeste oder die Digital Crown die Auswahl über die Uhr ändern. Und zum anderen kann die Auswahl über eine sogenanntes Segmented-Picker-Menü auf der iOS-Anwendung angepasst werden. Wichtig für den fließenden Übergang der Bedienbarkeit sind zwei Kennzeichen:

- Eine gute Abstraktion der View im State, wie im Quellcode 8.15 zu sehen ist
- und die Verknüpfung von Präsentationslogik und Anwendungslogik über Actions, wie Quellcode A.3 dargestellt.

In A.3 ist in den Zeilen 7 bis 10 und 26 bis 29 zu sehen, dass die Anwendungslogik über MVVM-Binding mit der Präsentationslogik verknüpft wird. Dabei hilft die TCA-Funktion `.binding(get: _, send: _)`, welche die Variable `auswahl` aus dem State bezieht und bei einer Änderung die Action `MainMenuAction.auswahlAngepasst(value:)` auslöst. Im Falle des `Pickers`[56] (Zeile 6) handelt es sich um eine fertige View von Swift-

[56] Dokumentation des `Pickers` (04.08.2020, 10:30) - `https://developer.apple.com/documentation/swiftui/picker`

UI, deren Präsentationslogik nicht öffentlich ist. Jedoch benötigen die jeweiligen Auswahlelemente, einen `.tag()`-Modifizierer (Zeile 12), damit die Präsentationslogik des `Picker`s funktioniert. Bei dem watchOS-Pendant wurde eine eigene `MenuPicker`-View erstellt, die exemplarisch für benutzerdefinierte Präsentationslogik steht. Die Variable `@Binding var aktuellerIndex: Int` stellt dabei die aktuelle Auswahl dar, welche über MVVM-Binding mit dem State der Anwendung verknüpft werden soll. Mithilfe von Gesten[57]- und Digital Crown[58]-Modifizieren können die bestimmten Benutzereingaben bzw. Events überwacht werden.

Weitere Beispiele, die sich mit dem Kriterium Bedienbarkeit befassen, sind in der TCA-Bibliothek[59] zu finden. Dazu gehört u. a. eine Spracherkennungs-Anwendung. Deren Funktionalität könnte in diese Anwendung integriert werden, um die Auswahl der Detailansicht per Sprache zu ändern.

8.7 Map

Wie in Unterabschnitt 8.1.2 beschrieben, konzentriert sich dieser Anwendungsfall auf den 1. Grad der Wiederverwendbarkeit und ahmt die Kartenanwendung des Patents nach. Demnach wurde für jedes der beiden Geräte jeweils ein State, die Actions und das Environment extra entwickelt. Es ist zu erwarten, dass mehr Codezeilen benötigt werden, als wenn Quellcode weiterverwendet werden könnte.

Auf der Smartwatch ist der Funktionsumfang der Kartenansicht sehr begrenzt. Deshalb kann der Kartenausschnitt auf der Uhr nicht über den Touchscreen verändert werden. Auch löst das Senden neuer Koordinaten vom Smartphone keine Updates auf der View aus. Das hat den Hintergrund, dass die Karten-View Teil des Frameworks WatchKit[60] ist. Unter SwiftUI können zwar WatchKit-Oberflächen-Objekte integriert und einmalig geupdatet werden. Neue Änderungen am State werden aber nicht reflektiert.[61]

[57] Dokumentation von `DragGesture` (04.08.2020, 10:30) - `https://developer.apple.com/documentation/swiftui/draggesture`

[58] Dokumentation des `digitalCrownRotation()`-Modifizierers (04.08.2020, 10:30) - `https://developer.apple.com/documentation/swiftui/view/digitalcrownrotation(_:from:through:by:sensitivity:iscontinuous:ishapticfeedbackenabled:)`

[59] TCA-Beispielanwendngen (04.08.2020, 13:20) - `https://github.com/pointfreeco/swift-composable-architecture#examples`

[60] Dokumentation von WatchKit (05.08.2020, 10:00) - `https://developer.apple.com/documentation/watchkit`

[61] Anleitung: *Building watchOS App Interfaces with SwiftUI* (05.08.2020, 10:30) - `https://developer.apple.com/documentation/watchkit/building_watchos_app_interfaces_with_swiftui`

8.8 Audio Player

Bei der Umsetzung des Audio-Player-Anwendungsfalls soll berücksichtigt werden, dass der gesamte Quellcode eines Stores wiederverwendet werden kann. Auf generische oder plattformabhängige Teile jedoch verzichtet werden muss. Das Problem dabei ist, dass durch diese Einschränkung das Environment des Stores, keine Sessions enthalten kann, da diese nicht generisch sind. Allerdings ist es möglich, wie im Zeitreise-Store zu sehen ist, Actions auf einer höheren Ebene zu verarbeiten bzw. in diesem Fall zu senden. Quellcode A.4 zeigt die Actions, den State, das Environment sowie den Reducer für diesen Anwendungsfall. Dabei ist der `AudioPlayerClient` in Zeile 26 zu beachten, denn die Geschäftslogik des Audio-Players soll von der Anwendungslogik getrennt sein. Vor diesem Hintergrund wird untersucht, wie Abhängigkeiten in solch einer Architektur behandelt werden können. Das Prinzip ist dabei identisch wie bei der Session. Es wird ein `AudioPlayerClient`-Objekt entworfen, welches über öffentliche Schnittstellen verfügt, worüber die Schnittstellen eines echten `AudioPlayer`-Objekts aufgerufen oder Funktionalität nachgebildet werden kann. Der `AudioPlayer` und dessen `Client` sind im Quellcode A.5 vollständig hinterlegt. Wichtig dabei ist, dass die Funktionalität des `AudioPlayer`s nur durch einen `Timer` simuliert wird und der Song nicht gewechselt werden kann. Grund dafür ist, dass der `MPMusicPlayerController`[62], welcher solche Aufgaben unter iOS übernimmt, watchOS nicht unterstützt. Das Prinzip der Implementierung eines `MPMusicPlayerController`s könnte aufgrund dessen Schnittstellen aber identisch sein.

Darüber hinaus muss bei diesem Anwendungsfall besonders die Wiederherstellbarkeit berücksichtigt werden. Wie im Quellcode A.4 zu sehen ist, gibt der `AudioPlayerClient` immer die aktuelle Spieldauer des Songs an den State zurück (Zeile 33 bis 36). Dadurch kann die View den Fortschritt reflektieren, während die Geschäftslogik des Players im Hintergrund agiert. Demnach werden auch nur die Actions, Starten und Stoppen des Songs sowie das manuelle Vor- und Zurückspulen gesendet. Nun gibt es den Fall, dass der Player auf beiden Anwendungen synchron läuft. Anschließend wird Anwendung B geschlossen, während auf Anwendung A der Player weiter läuft. Kommt es nach einer Zeit zu einer Kopplung und somit zu einer Synchronisation der States, muss aber auch der `AudioPlayer` der Anwendung B aktualisiert werden. Ansonsten würde auf beiden Geräten unterschiedliche Songs oder derselbe Song mit verschiedenem Fortschritt abgespielt werden.

Des Weiteren müssen Lebenszyklus-Funktionen[63] der Anwendung dafür sorgen, dass der AudioPlayer gestoppt wird, sobald die Anwendung in den Hintergrund tritt. Die Architektur selbst hat dafür nämlich keine eigenständige Lösung vorgesehen. Das in diesem

[62] Dokumentation des `MPMusicPlayerController` (27.08.2020, 11:50) - `https://developer.apple.com/documentation/mediaplayer/mpmusicplayercontroller`

[63] watchOS App Life Cycle (28.08.2020, 09:30) - `https://developer.apple.com/documentation/watchkit/working_with_the_watchos_app_life_cycle`

Abschnitt gezeigte Vorgehen kann auch auf andere Abhängigkeiten übertragen werden.

8.9 Settings

Das Hauptaugenmerk des Settings-Anwendungsfalls liegt darin, die Wiederverwendbarkeit vom Grad 3 zu untersuchen. Wie im Quellcode 8.17 zu sehen ist, wurden Compilerbedingungen genutzt, sodass jedes Einzelteil des Stores wiederverwendet werden kann. Zu beachten ist dabei, dass hauptsächlich die Sessions für Sonderregeln sorgen, da diese nicht generisch sind (Zeile 14 bis 18 und 28 bis 36). Bei der View wurden keine Compilerbedingungen angewendet, da diese nur aus einem Textfeld besteht, wie im Unterabschnitt 8.1.4 beschrieben.

```
1  #if os(iOS) || os(watchOS)
2  public struct SettingsState: Equatable {
3    var name: String = ""
4  }
5
6  public enum SettingsAction: Equatable {
7    case nameChanged(to: String)
8  }
9
10 public struct SettingsEnvironment {
11   #if os(iOS)
12   var sessionClient: AppWKSessionClient = .live
13   #elseif os(watchOS)
14   var connectivityClient: WKSessionClient = .live
15   #endif
16   var mainQueue: AnySchedulerOf<DispatchQueue> =
       ↪ DispatchQueue.main.eraseToAnyScheduler()
17 }
18
19 let settingsReducer = Reducer<SettingsState, SettingsAction,
     ↪ SettingsEnvironment> { state, action, environment in
20   switch action {
21     case let .nameChanged(to: name):
22       state.name = name
23       #if os(watchOS)
24       return environment.connectivityClient.send(
25           action: WKCoreAction.SettingsNameChanged(name: name)
26       ).fireAndForget()
27       #else
28       return environment.sessionClient.send(
29           action: AppCoreAction.SettingsNameChanged(name: name)
30       ).fireAndForget()
31       #endif
32   }
```

```
33 }
34 #endif
```

Quellcode 8.17: Actions, State, Environment und Reducer für den Settings-Anwendungsfall

9 Evaluationskonzept und Demonstrator

Wie zu Beginn der Umsetzung in Kapitel 8 erläutert, ist für eine komplexe Architektur keine umfängliche Evaluation im Sinne einer geschlossenen Durchführung praktikabel. Stattdessen werden wesentliche Software-Qualitätsmerkmale, des ISO/IEC Standards 25010:2011 [2] herausgearbeitet, an Hand derer sich eine Evaluation anwendungsfallbezogen orientieren würde. Zudem erfolgt eine Auflistung, mit welchen Konzepten oder auf welche Weise die Architektur hilft, diese Kriterien zu gewährleisten. Die vollständige Umsetzung der Kriterien ist aber anwendungsspezifisch, Aufgabe des Entwicklers und würde den Rahmen eine Bachelorarbeit sprengen.

9.1 Benutzbarkeit

9.1.1 Schutz vor Bedienfehlern

Wie im Abschnitt 6.5 geschildert, schützt die Kombination aus dem Paketprotokoll und dem Zeitreisekonzept vor kollidierenden bzw. gleichzeitig ausgelösten Actions. Die Anwendungen sind somit vor unzulässigen Zuständen und eventuellen Folgefehlern sicher. Vor Bedienfehlern anderer Art schützt die Architektur allerdings nicht, da die Geschäftslogik nicht Teil der Architektur ist.

9.1.2 Bedienbarkeit

Anhand des Beispiels aus Abschnitt 8.6 wird deutlich, dass die Architektur keine Einschränkungen auf Anwendungen hinsichtlich der Bedienbarkeit aufweist. Weitere Beispiele sind der komplexere Anwendungsfall aus Abschnitt 8.8 und die Beispielanwendungen[64] von TCA.

9.2 Portabilität

9.2.1 Anpassbarkeit

TCA ist für Apple-Plattformen entwickelt worden. Es ist in Swift umgesetzt, hängt vom Combine-Framework ab und erfordert daher mindestens iOS 13, macOS 10.15, Mac Catalyst 13, tvOS 13 und watchOS 6. Wenn eine Anwendung ältere Betriebssysteme

[64] TCA-Beispielanwendngen (10.09.2020, 16:30) - `https://github.com/pointfreeco/swift-composable-architecture#examples`

unterstützen möchte, können jedoch verschiedene Alternativen zu Combine verwendet werden. Dazu gehören ReactiveSwift[65] und RxSwift[66]. Zu beiden Frameworks existieren außerdem schon angepasste TCA-Projekte[67,68]. Möchte man die Architektur in einer anderen Programmiersprache umsetzen, muss bei der Auswahl darauf geachtet werden, dass MVVM und funktionale Programmierung, vergleichbar mit Combine, unterstützt werden.

9.3 Zuverlässigkeit

9.3.1 Verfügbarkeit

Die Verfügbarkeit in Bezug auf die Anwendung hängt von der Kompatibilität aller Module und Funktionen auf beiden Geräten ab. Die Architektur bewirkt jedoch, wie im Kapitel 4 erklärt, keine Einschränkung, da sie vollwertige Anwendungen auf beiden Geräten gleichzeitig unterstützt.

9.3.2 Fehlertoleranz

Die Fehlertoleranz der Architektur bezieht sich auf nur kollidierende Actions und Verbindungsabbrüche, welche in den Abschnitte 6.3 und 6.4 beschrieben sind. Die jeweilige Fehlerbehandlung ist im Abschnitt 6.5 erläutert.

9.3.3 Wiederherstellbarkeit

Wie in Abschnitt 6.3 beschrieben, umfasst die Architektur die Wiederherstellbarkeit nach einem Verbindungsabbruch. Die Qualität der Wiederherstellbarkeit ist dabei von der Implementierung des Mappers abhängig. Müssen beispielsweise Standardwerte beim Mappen verwendet werden, wie in Abschnitt 6.2 erläutert, ist eine vollständige Wiederherstellbarkeit ausgeschlossen. Wie der Audio-Player-Anwendungsfall aus Abschnitt 8.8 jedoch zeigt, sind neben dem Mapper auch Abhängigkeiten zu beachten, die bei einer Wiederherstellung nicht ausschließlich von der Architektur gehandhabt werden können. Nach einem Verbindungsabbruch sollten deshalb auch Lebenszyklus-Funktionen der Anwendung zum Einsatz kommen, um bei der Wiederherstellbarkeit zu

[65] ReactiveSwift GitHub Repository (01.09.2020, 11:50) - `https://github.com/ReactiveCocoa/ReactiveSwift`

[66] RxSwift GitHub Repository (01.09.2020, 11:50) - `https://github.com/ReactiveX/RxSwift`

[67] A RxSwift fork of The Composable Architecture (01.09.2020, 12:00) - `https://github.com/trading-point/reactiveswift-composable-architecture`

[68] A ReactiveSwift fork of The Composable Architecture (01.09.2020, 12:00) - `https://github.com/trading-point/reactiveswift-composable-architecture`

helfen. Darüber hinaus ist vorstellbar, dass ein gesamter State gespeichert und beim Start der Anwendung wiederverwendet werden kann.

9.4 Kompatibilität

9.4.1 Interoperabilität

Wie im Kapitel 6 erläutert, ist diese Architektur auf eine Interoperabilität im Peer-to-Peer Bereich ausgelegt. Jedoch ist auch eine Interoperabilität in einer Client-Server-Struktur vorstellbar. Exemplarisch könnten zwei Kommunikationspartner regulär interagieren und weitere Clients würden nur als Empfänger dienen. Solch ein Szenario ist in Smarthomes vorstellbar, bei denen Lichter über AR-Brille und Smartwatch bedient werden können.

9.5 Leistungseffizienz

9.5.1 Zeitverhalten

Beim Zeitverhalten einer Anwendung spielt die Transferzeit der Actions eine entscheidende Rolle. Treten viele Actions in kurzer Zeit auf, könnten diese nicht in Echtzeit von der Session gesendet werden. Dies wird auch in der Beispielanwendung deutlich, wenn sich die Song-Zeit durch den Schieberegler verändert. Mithilfe von funktionaler reaktiver Programmierung könnte das Senden so gestaltet werden, dass mehrere Actions als Gruppe[69] sowie die letzte[70] aufgetretene Action übermittelt wird.

9.5.2 Ressourcennutzung

Die Architektur unterstützt durch TCA auch optionale States. Diese belegen nur dann Speicherplatz, wenn sie benötigt werden. Zudem sollte die Historie dahingehend verbessert werden, dass nur Änderungen gespeichert werden, wie das bei der Versionsverwaltung Git[71] der Fall ist. Des Weiteren hat die Historie aktuell eine Obergrenze an States, die gespeichert werden können. Diese Grenze könnte aber auch auf belegten Speicherplatz umfunktioniert werden, sodass ein Hard- und ein Softlimit an MB festgelegt wird. Darüber hinaus läuft das Zeitreisekonzept immer im Hintergrund. Dabei muss es nur dann die Arbeit aufnehmen, wenn eine Kopplung besteht.

[69] Dokumentation von `Publishers.Collect` (07.09.2020, 15:00) - `https://developer.apple.com/documentation/combine/publishers/collect`

[70] Dokumentation `Publishers.Last` (07.09.2020, 15:10) - `https://developer.apple.com/documentation/combine/publishers/last`

[71] Git (14.09.2020, 18:50) - `https://git-scm.com`

Des Weiteren gibt es weitere Punkte, die optimiert werden können, jedoch nicht Teil der Architektur sind. Beispielsweise kann die Größe der Actions beim Senden optimiert werden.

9.6 Wartbarkeit

9.6.1 Modularität und Modifizierbarkeit

Wie im Abschnitt 8.2 beschrieben, ist eine hohe Modularität und Modifizierbarkeit durch TCA gegeben. Wie die beiden Kriterien genau erlangt werden können, ist im Kapitel 8 anhand der Session des Audio-Player-Anwendungsfalls erläutert wurden.

9.6.2 Wiederverwendbarkeit

Durch das Architekturkonzept ist eine Portierung der Anwendungslogik auf andere Zielplattformen möglich, wie in Unterabschnitt 9.2.1 erläutert. Wenn die Zielplattform dieselbe Programmiersprache verwendet, können zudem ganze Stores und Views, mit eventuellen Einschränkungen, wiederverwendet werden. Das geht aus der Simulation der drei Grade der Wiederverwendbarkeit in Kapitel 8 hervor. Abgesehen von dem Mehraufwand gibt es keine weiteren negativen Effekte bei Grad 1. Vermutlich spart Grad 3 durch die Compilerbedingungen im Vergleich zu Grad 1 kaum Code-Zeilen. Um das jedoch genau festzustellen, benötigt es eine Analyse von identischen Stores.

Weiter ist vorstellbar, dass fertige TCA-Anwendungen um das Zeitreisekonzept sowie eine Session erweitert werden können. Durch die Struktur der `TimeTravelView` (siehe Quellcode 8.12) muss die Haupt-View der Anwendung dort lediglich eingesetzt werden, wie im Quellcode 8.11 dargestellt und in Unterabschnitt 8.3.4 angedeutet. Dieser minimale Quellcode-Fußabdruck zeigt, dass die Architektur gut strukturiert ist. Dagegen zeugt die Integration der Session in viele Environments von einer schlecht strukturierten Architektur. Deshalb erscheint es sinnvoll, die Session in ein eigenes Zeitreise-Environment zu packen, um den Quellcode-Fußabdruck der Architektur in der regulären Anwendung weiter zu minimieren. Außerdem sind so alle Actions, die gesendet werden müssen, zentral abgelegt und nicht in den jeweiligen Stores verstreut.

9.6.3 Analysierbarkeit und Testbarkeit

Die Analysierbarkeit und Testbarkeit sind auf Implementierungsebene zu erreichen. Wie in Unterabschnitt 8.2.2 erwähnt, können dazu beispielsweise die TCA-Funktionen `.debug()` sowie `.singpost()` verwendet werden, um den Reducer zu beobachten. Die TCA-Funktion `.assert()` wiederum kann bei Unittests eines Stores helfen.

9.7 Sicherheit

9.7.1 Vertraulichkeit und Integrität

Die Architektur hat keine Absicherung der Qualitätskriterien Vertraulichkeit und Integrität und stellt auch keine diesbezüglichen Anforderungen an das Transportprotokoll.

9.7.2 Nachweisbarkeit

Zur Unterstützung dieses Kriteriums können alle oder ausgewählte Actions geloggt werden. Dazu kann nach dem Vorbild der `.degub()`-Funktion von TCA eine Reducer-Funktion entwickelt werden, die jede aufgetretene Action loggt.

9.8 Demonstrator und Testlauf

Die Funktionsweise des fertigen Demonstrators [31] kann auf YouTube während eines Testablaufs betrachtet werden, welcher hier in einem Kurzprotokoll zusammengefasst ist.

Zeitstempel	Beschreibung
0:00 - 0:10	Start der Anwendung auf den Simulatoren
0:10 - 0:17	Wechsel zur Settings-Detailansicht
0:17 - 0:46	Präsentation verschiedener Textfeld-Eingabemöglichkeiten
0:46 - 0:54	Wechsel zur Audio-Player-Detailansicht
0:54 - 1:28	Vorstellung der Interoperabilität des Audio Players
1:28 - 1:35	Kopplung der Simulatoren aufheben
1:35 - 1:43	Weitere Zustandsänderungen auf dem iPhone
1:43 - 1:49	Erneute Kopplung der Simulatoren sowie Synchronisation der States und der Audio-Player-Objekte
1:49 - 2:04	Erneutes Starten des Audio Players
2:05 - 2:31	Auslösen von kollidierender Actions und Behebung durch das Zeitreisekonzept

Tabelle 9.1: Kurzprotokoll zum Ablauf des Demonstrators [31]

Dabei ist anzumerken, dass es sich bei den Geräten um Simulatoren handelt und die Zeitverzögerung zwischen dem Senden und Empfangen der Actions auf echten Geräten nicht bemerkbar ist. Des Weiteren kann die Map-Detailansicht auf der Smartwatch nicht geöffnet werden, da der Simulator ansonsten abstürzt. Die Ursache des Problems ist unter iOS 14 durch eigenständige SwiftUI-Views[72] jedoch behoben. Des Weiteren kann

[72] Dokumentation der `Map` von MapKit unter iOS 14 (08.09.2020, 09:00) - `https://developer.apple.com/documentation/mapkit/map`

auch eine prototypische AR-Anwendung [29] auf YouTube betrachtet werden, welche Bild-Tracking verwendet und die Probleme aus dem Unterabschnitt 7.2.2 veranschaulicht.

10 Fazit

Das Patent *wristwatch based interface for augmented reality eyeware* [26] von Jonathan M. Rodriguez II. beschreibt verschiedene Möglichkeiten, wie eine Smartwatch mit einer AR-Brille kombiniert werden kann, um eine gesellschaftlich akzeptable Methode der gestischen Eingabe zu realisieren. Mit dieser Arbeit wurde eine mögliche Software-Architektur für solch ein kombiniertes System konzipiert, entwickelt und evaluiert.

Dazu wurde das Patent auf Hinweise zu Anforderungen an eine mögliche Architektur untersucht. Darauf aufbauend ist ein Konzept entstanden, welches die Anforderungen umsetzt. Des Weiteren ist eine Konzeption einer Beispielanwendung mit verschiedenen integrierten Anwendungsfällen entwickelt wurden. Mithilfe dieser Anwendungsfälle sollte gezeigt werden, wie ausgewählte Software-Qualitätsmerkmale im Zusammenspiel mit der Architektur erreicht werden können. Die Umsetzung der Architektur erfolgte dabei für eine iOS- und eine watchOS-App, wobei die Smartphone-Anwendung stellvertretend für eine AR-Brille-Anwendung steht. Die Darlegung wesentlicher Implementierungsdetails sind ebenfalls Gegenstand der Arbeit.

Die anschließende Evaluierung zielt auf die Umsetzbarkeit der ausgewählten Software-Qualitätskriterien ab. Sie verdeutlicht zugleich, dass die Architektur noch Optimierungspotenzial besitzt und lediglich als sogenanntes Proof of Concept, also als „Beweis des Konzepts“ gewertet werden sollte. Des Weiteren ist mit Fehlern zu rechnen, da keine Unit- und Integrationtests geschrieben wurden. Bei dem Demonstrator [31] sind jedoch keine Fehler ersichtlich.

11 Ausblick

Wie in der Umsetzung erwähnt, mussten einige Teile des Zeitreisekonzepts speziell für dieses Anwendungsbeispiel entwickelt werden. Das bedeutet, dass die Architektur-Software nicht vollständig für eine neue Anwendung wiederverwendet werden kann. Es besteht daher die Notwendigkeit, die Software dahingehend zu verbessern, dass sie als ein Framework oder eine Bibliothek dient.

Darüber hinaus wurde in Unterabschnitt 7.2.2 ein Simulator erwähnt, der bei der Entwicklung einer Anwendung mit dieser Architektur helfen könnte. Damit ist eine Desktopanwendung gemeint, in der die Views und Geschäftslogiken der Smartwatch und AR-Brille simuliert werden. Konkret könnte der Simulator so gestaltet sein, dass mittig eine Smartwatch mit den entsprechenden Views, umschlossen von der AR-Benutzeroberfläche, dargestellt wird. Dadurch kann auf das Tracking beim Testen verzichtet werden, wodurch eine schnellere Entwicklung möglich wird.

Auch bleibt offen, wie sich die Architektur unter Realbedingungen auf einer AR-Brille verhält und wie eine Abstraktion von erkannten Gesten in Actions aussehen kann. Es ist vorstellbar, die Architektur sowie eine Beispielanwendung für die Holo Lens 2 zu entwickeln und zu testen. Voraussichtlich ist hierfür ein neues Trackingverfahren zu entwickeln, um Smartwatches erfassen und verfolgen zu können.

Zudem beschränkt sich die Architektur nicht nur auf AR-Brillen in Kombination mit Smartwatches. Auch Anwendungen auf anderen Plattformen wie Computer, Tablets, Fernseher, Auto oder Smartphone, könnten mit dieser Architektur durch AR-Benutzeroberflächen erweitert werden. Hinzu kommt, dass die Architektur in einer Client-Server-Umgebung Anwendung findet, um mit mehreren Geräten simultan zu interagieren. Auch eine Einbindung von IoT-Geräte, beispielsweise in einem Smarthome, ist vorstellbar. Exemplarisch könnte das Entsperrmuster aus Abbildung 3.2 dazu genutzt werden, um eine Haustür zu öffnen.

Des Weiteren wird im Patent eine Methode beschrieben, die kollidierende Actions zu verhindern scheint. Dazu werden Eingaben über Touch und die Gestenerkennung zusammengeführt. In Abbildung 11.1 ist dazu das passende Ablaufdiagramm abgebildet. Es zeigt, wie zwischen der Eingabe über die AR-Brille oder über die Uhr gewählt werden kann. Die Smartwatch ist zudem auch in der Lage, Gesten zu registrieren (Abbildung 11.1 **1417** und **1407**). Auch dieser Sachverhalt könnte in Kombination mit der Architektur getestet und erprobt werden.

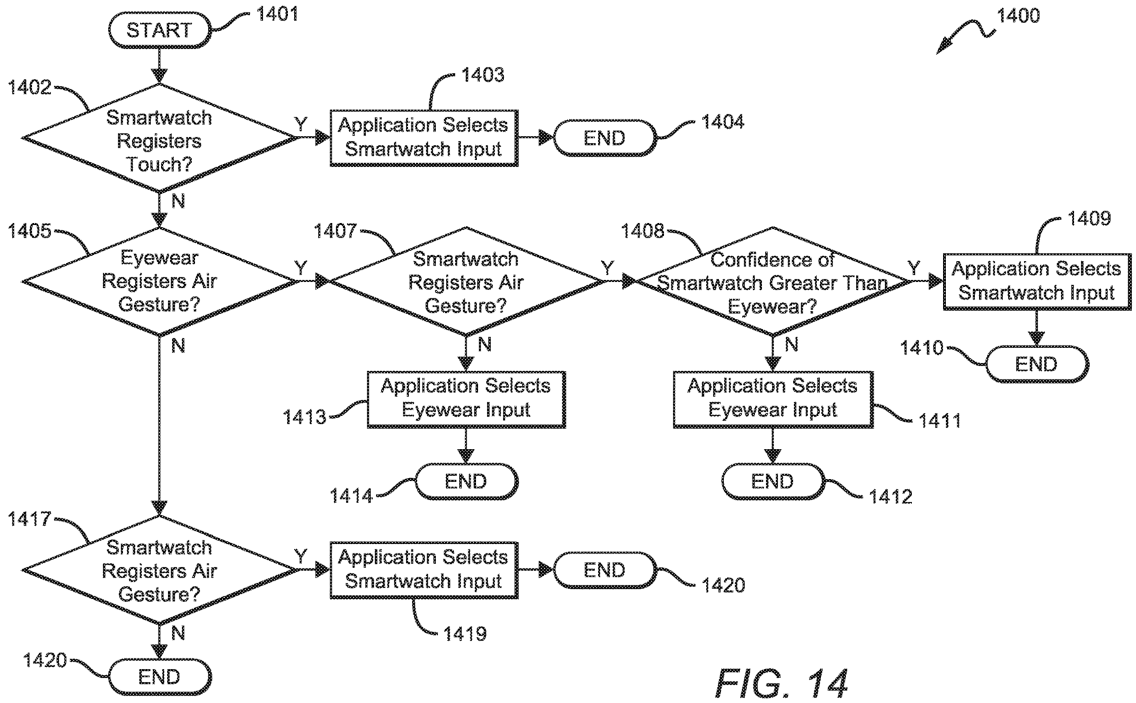


Abbildung 11.1: Karten-App der Smartwatch erweitert durch eine AR-Oberfläche
Quelle: Originalbild aus dem Patent [26, Seite 13 Abbildung 14]

Anhang A: Quellcode

```
// Der State der Anwendung
struct AppState {
    // Eine Variable für Ganzzahlen
    var zahl: Int = 0
}

// Auflistung aller Actions, jedoch ohne Geschaeftslogik
enum ActionType {
    case inkrementieren
    case dekrementieren
    case multiplizieren(Int)
}

// Der Reducer nimmt Actions entgegen und
// wodurch er Aenderungen am State ausloest
class Reducer: ObservableObject {
    // Diese Funktion updated den State,
    // abhängig von der mitgelieferten Action und dem State.
    // Hier steht die Geschaeftslogik der Actions.
    func update(state: inout AppState, action: ActionType) {
        switch action {
            // Variable 'zahl' wird mit 1 addiert
            case .inkrementieren:
                state.zahl += 1
            // Variable 'zahl' wird mit 1 subtrahiert
            case .dekrementieren
                state.zahl -= 1
            // Varibale 'zahl' wird mit dem mitgegebenen Faktor multipliziert
            case .multiplizieren(let faktor)
                state.zahl *= faktor
        }
    }
}

// Geruest zum Verwalten des States
final class Store: ObservableObject {
    // Ein Store benötigt einen State und einen Reducer
    init(initialState: AppState, reducer: Reducer) {
        self.state = initialState
        self.reducer = reducer
    }

    // Der State ist von aussen lesbar.
    // Aenderungen muessen ueber den Reducer gemacht werden.
    // @Published bedeutet, dass andere z.B. Views ueber Aenderungen
    // benachrichtigt werden.
    @Published private(set) var state: AppState

```

```
49      // Der Reducer ist während der Laufzeit nicht manupulierbar
50      private let reducer: Reducer
51
52      // Ueber diese Funktion (Dispatcher) werden Aenderungen beauftragt.
53      // Dazu muss die jeweilige Action mitgeliefert werden.
54      // In groesseren Anwendungen können je nach ActionTyp verschiedene Reducer
55      // zum Ausführen beauftragt werden
56      func erledigen(actionType: ActionType) {
57          self.reducer.update(state: &state, action: actionType)
58      }
59  }
60
61  // Beispielhafter Programmablauf:
62  // Store mit State und Reducer wird erstellt
63  let store = Store(initialState: AppState(), reducer: Reducer())
64  // Actions werden an den Store gesendet:
65  store.erledigen(actionType: .inkrementieren)     // zahl = 1
66  store.erledigen(actionType: .multiplizieren(2)) // zahl = 2
67  store.erledigen(actionType: .dekrementieren)     // zahl = 1
```

Quellcode A.1: Prototypische Umsetzung der Architektur für eine Anwendung in Swift

```
1  extension SessionClient {
2    static let attrappe = SessionClient(
3      erstellen: { () -> Effect<Action, Never> in
4        .run { _ in return AnyCancellable { } } },
5      abschicken: { _ in .fireAndForget { } },
6      sync: { _ in .fireAndForget { } }
7    )
8  }
```

Quellcode A.2: Beispiel einer `SessionClient`-Attrappe ohne Funktion

```swift
// MainMenuView iOS
struct MainMenuView: View {
  let store: Store<MainMenuState,MainMenuAction>
  var body: some View {
    WithViewStore(self.store) { viewStore in
      Picker(
        selection: viewStore.binding(
          get: { $0.auswahl.rawValue },
          send: MainMenuAction.auswahlAngepasst(value:))
      ) {
        ForEach(MainMenuView.allCases, id: \.self ) { viewCase in
          Text("\(viewCase.titel)").tag(viewCase.rawValue)
        }
      }.pickerStyle(SegmentedPickerStyle())
    }
  }
}

// MainMenuView watchOS
struct MainMenuView: View {
  let store: Store<MainMenuState,MainMenuAction>
  var body: some View {
    WithViewStore(self.store) { viewStore in
      MenuPicker(
        detailAnsichtenAnzahl: 3,
        aktuellerIndex: viewStore.binding(
          get: { $0.auswahl.rawValue },
          send: MainMenuAction.auswahlAngepasst(value:))
      ) {
        Card(image: "map.fill", name: "Map").onTapGesture {
          viewStore.send(.mapView(sichtbar: true))
        }
        Card(image: "headphones", name: "Audio Player").onTapGesture {
          viewStore.send(.audioPlayerView(sichtbar: true))
        }
        Card(image: "gear", name: "Settings").onTapGesture {
          viewStore.send(.settingsView(sichtbar: true))
        }
      }
    }
  }
}

// MenuPicker
struct MenuPicker<Content: View>: View {
  @Binding var aktuellerIndex: Int
  @GestureState private var translation: CGFloat = 0
  let detailAnsichtenAnzahl: Int
  let content: Content

```

```
51  init(detailAnsichtenAnzahl: Int, aktuellerIndex: Binding<Int>, @ViewBuilder
    ↪  content: () -> Content) {
52    ...
53  }
54
55  var body: some View {
56    GeometryReader { geometry in
57      HStack(spacing: 0) {
58        self.content.frame(width: geometry.size.width)
59      }
60      .focusable(true)
61      .frame(width: geometry.size.width, alignment: .leading)
62      .offset(x: -CGFloat(self.aktuellerIndex) * geometry.size.width)
63      .offset(x: self.translation)
64      .digitalCrownRotation(
65        Binding.init(
66          get: { Double(self.aktuellerIndex) },
67          set: { if $0.truncatingRemainder(dividingBy: 1) == 0 {
            ↪  self.aktuellerIndex = Int($0) } }
68        ),
69        from: 0.0,
70        through: Double(self.detailAnsichtenAnzahl - 1),
71        by: 1.0,
72        sensitivity: .medium,
73        isContinuous: false,
74        isHapticFeedbackEnabled: true )
75      .gesture(
76        DragGesture().updating(self.$translation) { value, state, _ in
77          state = value.translation.width
78        }.onEnded { value in
79          let offset = value.translation.width / geometry.size.width
80          let newIndex = (CGFloat(self.aktuellerIndex) - offset).rounded()
81          self.aktuellerIndex = min(max(Int(newIndex), 0),
            ↪  self.detailAnsichtenAnzahl - 1)
82        }
83      )
84    }
85  }
86 }
```

Quellcode A.3: Beispiel für die Verknüpfung der Präsentationslogik mit der Anwendungslogik

```
public struct Track: Equatable {
  let titel: String
  let artist: String
  let album: String
  let length: Int
}

public struct AudioState: Equatable {
  var currentTrack: Track
  var time: Int = 0
  var isPlaying: Bool = false
}

public enum AudioAction: Equatable {
  case onAppear
  case play
  case pause
  case back
  case next
  case setTrack(to: Double)
  case update(Result<AudioPlayerClient.Action,Never>)
  case onDisappear
}

struct AudioEnvironment {
  var player: AudioPlayerClient = .live
  var mainQueue = DispatchQueue.main.eraseToAnyScheduler()
}

let audioReducer = Reducer<AudioState,AudioAction,AudioEnvironment> { state,
  ↪ action, environment in
  switch action {
    case .onAppear:
      return environment.player.start(track: state.currentTrack)
        .receive(on: environment.mainQueue)
        .catchToEffect()
        .map(AudioAction.update)
    case .play:
      state.isPlaying = true
      return environment.player.resume().fireAndForget()
    case .pause:
      state.isPlaying = false
      return environment.player.stop().fireAndForget()
    case .back:
      return .none // nicht implementiert
    case .next:
      return .none // nicht implementiert
    case let .setTrack(to: to):
      state.isPlaying = false
      let time = Int(to)
```

```
50        state.time = time
51        return environment.player.setTo(time: time).fireAndForget()
52      case let .update(.success(action)):
53        switch action {
54            case let .update(value):
55              state.time = value
56              return .none
57        }
58      case .onDisappear:
59        return environment.player.stop().fireAndForget()
60    }
61  }
```

Quellcode A.4: Actions, State, Environmanet und Reducer für den Audio-Player-Anwendungsfall

```
public struct AudioPlayerClient {
  public enum Action: Equatable {
    case update(Int)
  }

  private var startPlayer: (Track) -> Effect<Action, Never>
  private var playTrack: (Track) -> Effect<Never, Never>
  private var stopTrack: () -> Effect<Never, Never>
  private var resumeTrack: () -> Effect<Never, Never>
  private var setTrackTo: (Int) -> Effect<Never, Never>

  func play(track: Track) -> Effect<Never, Never> {
    self.playTrack(track)
  }

  func stop() -> Effect<Never, Never> {
    self.stopTrack()
  }

  func resume() -> Effect<Never, Never> {
    self.resumeTrack()
  }

  func setTo(time: Int) -> Effect<Never, Never> {
    self.setTrackTo(time)
  }

  func start(track: Track) -> Effect<Action, Never> {
    self.startPlayer(track)
  }
}

extension AudioPlayerClient {
  static let live = AudioPlayerClient(
    startPlayer: { (track) -> Effect<Action, Never> in
      .run { subscriber in
        AudioPlayer.shared.setTrack(track)
        return AudioPlayer.shared.time.map { value in
          Action.update(value)
        }.sink { (action) in
          subscriber.send(action)
        }
      }
    },
    playTrack: { (track) -> Effect<Never, Never> in
      .fireAndForget {
        AudioPlayer.shared.start(track)
      }
    },
    stopTrack: { () -> Effect<Never, Never> in
```

```
51        .fireAndForget {
52          AudioPlayer.shared.stop()
53        }
54      },
55      resumeTrack: { () -> Effect<Never, Never> in
56        .fireAndForget {
57          AudioPlayer.shared.resume()
58        }
59      },
60      setTrackTo: { (time) -> Effect<Never, Never> in
61        .fireAndForget {
62          AudioPlayer.shared.setTime(to: time)
63        }
64      }
65    )
66  }
67
68  class AudioPlayer: NSObject  {
69    static let shared = AudioPlayer()
70
71    private var track: Track?
72
73    public var time = CurrentValueSubject<Int, Never>(0)
74
75    private weak var timer: Timer?
76
77    private override init( ) { }
78
79    public func start(_ track: Track) {
80      setTrack(track)
81      resume()
82    }
83
84    public func stop() {
85      timer?.invalidate()
86    }
87
88    public func resume() {
89      self.timer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true, block:
        ↪ { [self] _ in
90        if let length = self.track?.length {
91          if self.time.value <= length {
92            self.time.value += 1
93          } else {
94            self.stopTimer()
95          }
96        } else {
97          self.stopTimer()
98        }
99      })
100   }
101
```

```
102   public func setTime(to: Int) {
103     self.timer?.invalidate()
104     self.timer = nil
105     self.time.value = to
106   }
107
108   public func setTrack(_ track: Track) {
109     stopTimer()
110     if self.track != track {
111       self.track = track
112       self.time.value = 0
113     }
114   }
115
116   private func stopTimer() {
117     timer?.invalidate()
118     self.timer = nil
119   }
120 }
```

Quellcode A.5: Quellcode der verwendeten `AudioPlayer`-Attrappe und des `AudioPlayerClients`

Literaturverzeichnis

[1] “ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary,” Dez. 2010. DOI: 10.1109/IEEESTD.2010.5733835, S. 418, besucht am: 09.09.2020, 12:55.

[2] “ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models,” Mär. 2011. [Online]. Verfügbar unter: https://www.iso.org/obp/ui/#iso:std:35733:en, besucht am: 14.09.2020, 18:35.

[3] Apple, “Combine,” Jun. 2019. [Online]. Verfügbar unter: https://developer.apple.com/documentation/combine, besucht am: 02.07.2020, 17:45.

[4] R. T. Azuma, “A Survey of Augmented Reality,” *Presence: Teleoperators and Virtual Environments*, Band 6, Nr. 4, SS. 355–385, Aug. 1997. [Online]. Verfügbar unter: https://doi.org/10.1162/pres.1997.6.4.355, besucht am: 19.06.2020, 13:10.

[5] M. Bragge, “Model-View-Controller architectural pattern and its evolution in graphical user interface frameworks,” Aug. 2013. [Online]. Verfügbar unter: https://lutpub.lut.fi/handle/10024/92156, besucht am: 02.07.2020, 17:40.

[6] A. Comport, E. Marchand, M. Pressigout, und F. Chaumette, “Real-time markerless tracking for augmented reality: The virtual visual servoing framework,” *IEEE transactions on visualization and computer graphics*, Band 12, SS. 615–28, Aug. 2006, besucht am: 01.07.2020, 10:10.

[7] D. Dobbelstein, P. Hock, und E. Rukzio, “Belt: An Unobtrusive Touch Input Device for Head-worn Displays,” in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, Reihe CHI ’15. Seoul, Republic of Korea: Association for Computing Machinery, Apr. 2015. DOI: 10.1145/2702123.2702450,. ISBN: 978-1-4503-3145-6, SS. 2135–2138. [Online]. Verfügbar unter: https://doi.org/10.1145/2702123.2702450, besucht am: 19.06.2020, 13:20.

[8] B. Ens, A. Byagowi, T. Han, J. D. Hincapié-Ramos, und P. Irani, “Combining Ring Input with Hand Tracking for Precise, Natural Interaction with Spatial Analytic Interfaces,” in *Proceedings of the 2016 Symposium on Spatial User Interaction*, Reihe SUI ’16. Tokyo, Japan: Association for Computing Machinery, Okt. 2016. DOI: 10.1145/2983310.2985757,. ISBN: 978-1-4503-4068-7, SS. 99–102. [Online]. Verfügbar unter: https://doi.org/10.1145/2983310.2985757, besucht am: 19.06.2020, 13:30.

[9] A. Freeman, *Pro ASP.NET Core MVC 2*, 2te Aufl., Reihe Expert's Voice in .NET. Apress, 2017. ISBN: 978-1-4842-3150-0,. [Online]. Verfügbar unter: https://www.apress.com/gp/book/9781484231494, besucht am: 02.07.2020, 13:00.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, und G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1te Aufl. Addison-Wesley Professional, Nov. 1994. ISBN: 978-0-201-63361-0,

[11] J. Heck, *Using Combine*, 1te Aufl., Mär. 2020. [Online]. Verfügbar unter: https://heckj.github.io/swiftui-notes/, besucht am: 02.07.2020, 17:40.

[12] P. Hegarty, "Lecture 2: MVVM and the Swift Type System - YouTube," Mai 2020. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=4GjXq2Sr55Q, besucht am: 14.07.2020, 17:15.

[13] S. Hoffmann und R. Lienhart, *OpenMP: Eine Einführung in die parallele Programmierung mit C/C++*, Reihe Informatik im Fokus. Berlin, Heidelberg: Springer-Verlag, 2008. ISBN: 978-3-540-73122-1 978-3-540-73123-8,. [Online]. Verfügbar unter: http://link.springer.com/10.1007/978-3-540-73123-8, besucht am: 13.07.2020, 15:35.

[14] Y.-T. Hsieh, A. Jylhä, V. Orso, L. Gamberini, und G. Jacucci, "Designing a Willing-to-Use-in-Public Hand Gestural Interaction Technique for Smart Glasses," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, Reihe CHI '16. San Jose, California, USA: Association for Computing Machinery, Mai 2016. DOI: 10.1145/2858036.2858436,. ISBN: 978-1-4503-3362-7, SS. 4203–4215. [Online]. Verfügbar unter: https://doi.org/10.1145/2858036.2858436, besucht am: 20.06.2020, 15:10.

[15] D. Jain, L. Findlater, J. Gilkeson, B. Holland, R. Duraiswami, D. Zotkin, C. Vogler, und J. E. Froehlich, "Head-Mounted Display Visualizations to Support Sound Awareness for the Deaf and Hard of Hearing," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, Reihe CHI '15. Seoul, Republic of Korea: Association for Computing Machinery, Apr. 2015. DOI: 10.1145/2702123.2702393,. ISBN: 978-1-4503-3145-6, SS. 241–250. [Online]. Verfügbar unter: https://doi.org/10.1145/2702123.2702393, besucht am: 19.06.2020, 11:15.

[16] H. Kato und M. Billinghurst, "Marker tracking and HMD calibration for a video-based augmentedreality conferencing system," Feb. 1999. DOI: 10.1109/IWAR.1999.803809,. ISBN: 978-0-7695-0359-2, SS. 85–94, besucht am: 30.06.2020, 17:50.

[17] G. Klein und T. Drummond, "Robust Visual Tracking for Non-Instrumented Aug-

mented Reality," in *Proceedings of the 2nd IEEE/ACM International Symposium on Mixed and Augmented Reality*, Reihe ISMAR '03. USA: IEEE Computer Society, Okt. 2003. ISBN: 978-0-7695-2006-3, S. 113, besucht am: 27.06.2020, 19:22.

[18] G. Klein und D. Murray, "Parallel Tracking and Mapping for Small AR Workspaces," in *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan, Nov. 2007. DOI: 10.1109/TVCG.2006.78,. [Online]. Verfügbar unter: https://www.researchgate.net/publication/6979987_Real-time_markerless_tracking_for_augmented_reality_The_virtual_visual_servoing_framework, besucht am: 01.07.2020, 10:20.

[19] D. Lee, Y. Lee, Y. Shin, und I. Oakley, "Designing Socially Acceptable Hand-to-Face Input," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, Reihe UIST '18. Berlin, Germany: Association for Computing Machinery, Okt. 2018. DOI: 10.1145/3242587.3242642,. ISBN: 978-1-4503-5948-1, SS. 711–723. [Online]. Verfügbar unter: https://doi.org/10.1145/3242587.3242642, besucht am: 21.06.2020, 13:20.

[20] A. Mehler-Bicher und L. Steiger, *Augmented Reality: Theorie und Praxis*. De Gruyter Oldenbourg, Mai 2014. ISBN: 978-3-11-035385-3,. [Online]. Verfügbar unter: https://www.degruyter.com/view/title/329764, besucht am: 28.06.2020, 13:00.

[21] P. Milgram und F. Kishino, "A Taxonomy of Mixed Reality Visual Displays," *IEICE Trans. Information Systems*, Band E77-D, Nr. 12, SS. 1321–1329, Dez. 1994.

[22] S. Niedermair, "Augmented reality on mobile devices for architectural visualisation," Masterarbeit, 2012. [Online]. Verfügbar unter: https://repositum.tuwien.at/handle/20.500.12708/13692, besucht am: 30.06.2020, 11:10.

[23] J. Papa, "Fundamental MVVM," Aug. 2011. [Online]. Verfügbar unter: https://visualstudiomagazine.com/articles/2011/08/15/fundamental-mvvm.aspx, besucht am: 02.07.2020, 13:00.

[24] I. Radu, "Augmented reality in education: a meta-review and cross-media analysis," *Personal and Ubiquitous Computing*, Band 18, Nr. 6, SS. 1533–1543, Aug. 2014. [Online]. Verfügbar unter: https://doi.org/10.1007/s00779-013-0747-y, besucht am: 19.06.2020, 10:40.

[25] J. Rekimoto, "Matrix: a realtime object identification and registration method for augmented reality," in *Proceedings. 3rd Asia Pacific Computer Human Interaction (Cat. No.98EX110)*, Jul. 1998. DOI: 10.1109/APCHI.1998.704151, SS. 63–68, besucht am: 30.06.2020, 18:00.

[26] J. M. Rodriguez II., “Wristwatch based interface for augmented reality eyewear,” US Patent US20 200 150 435A1, Mai, 2020. [Online]. Verfügbar unter: https://patents.google.com/patent/US20200150435A1/en, besucht am: 19.06.2020, 11:40.

[27] J. Rolland, Y. Baillot, und A. Goon, “A survey of tracking technology for virtual environments,” *Fundamentals of Wearable Computers and Augmented Reality*, Jan. 2001. [Online]. Verfügbar unter: https://www.researchgate.net/publication/242415577_A_survey_of_tracking_technology_for_virtual_environments, besucht am: 28.06.2020, 18:55.

[28] B. Schwerdtfeger, R. Reif, W. A. Günthner, und G. Klinker, “Pick-by-vision: there is something to pick at the end of the augmented tunnel,” *Virtual Reality*, Band 15, Nr. 2, SS. 213–223, Jun. 2011. [Online]. Verfügbar unter: https://doi.org/10.1007/s10055-011-0187-9, besucht am: 04.06.2020, 12:55.

[29] H. Steiner, “AR-Watch-Konzept Prototyp,” Sep. 2020. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=FqYm-nb1OEc, besucht am: 13.09.2020, 12:55.

[30] H. Steiner, “Entwicklung einer Dokumenten-Scanner-App - Digitalisierung der Verwaltung an der Hochschule Mittweida,” S. 54, Mai 2020. [Online]. Verfügbar unter: https://github.com/SteinerHannes/Praktikumsbericht/blob/master/Beleg.pdf, besucht am: 07.09.2020, 20:00.

[31] H. Steiner, “TCA-Zeitreise-Architektur-Konzept,” Aug. 2020. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=ltAlfOqjvS4, besucht am: 07.09.2020, 17:30.

[32] A. S. Tanenbaum und H. Bos, *Modern Operating Systems*, 4te Aufl. Boston: Pearson, Mär. 2014. ISBN: 978-0-13-359162-0,

[33] M. Tönnis, *Augmented Reality: Einblicke in die Erweiterte Realität*, Reihe Informatik im Fokus. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, Band 0. ISBN: 978-3-642-14178-2 978-3-642-14179-9,. [Online]. Verfügbar unter: http://link.springer.com/10.1007/978-3-642-14179-9, besucht am: 01.07.2020, 11:15.

[34] G. S. Von Itzstein, M. Billinghurst, R. T. Smith, und B. H. Thomas, “Augmented Reality Entertainment: Taking Gaming Out of the Box,” in *Encyclopedia of Computer Graphics and Games*, N. Lee, Hrsg. Springer International Publishing, 2017, SS. 1–9. ISBN: 978-3-319-08234-9,. [Online]. Verfügbar unter: https://doi.org/10.1007/978-3-319-08234-9_81-1, besucht am: 19.06.2020, 13:00.

[35] D. Wals, “An introduction to Combine,” Jan. 2020. [Online]. Verfügbar unter: https://www.donnywals.com/an-introduction-to-combine/, besucht am: 02.07.2020, 20:15.

[36] C.-Y. Wang, W.-C. Chu, P.-T. Chiu, M.-C. Hsiu, Y.-H. Chiang, und M. Y. Chen, “PalmType: Using Palms as Keyboards for Smart Glasses,” in *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, Reihe MobileHCI ’15. Copenhagen, Denmark: Association for Computing Machinery, Aug. 2015. DOI: 10.1145/2785830.2785886,. ISBN: 978-1-4503-3652-9, SS. 153–160. [Online]. Verfügbar unter: https://doi.org/10.1145/2785830.2785886, besucht am: 20.06.2020, 15:30.

[37] N. J. Wei, B. Dougherty, A. Myers, und S. M. Badawy, “Using Google Glass in Surgical Settings: Systematic Review,” *JMIR mHealth and uHealth*, Band 6, Nr. 3, S. 54, Mär. 2018. [Online]. Verfügbar unter: https://mhealth.jmir.org/2018/3/e54/, besucht am: 19.06.2020, 11:50.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 23.09.2020