Inhaltsverzeichnis

1 Einleitung

Bis zum Jahre 1973 und der Entwicklung des ersten Eingabegerätes mit Graphischer Oberfläche (GUI) (Xerox Alto [1]) erfolgte die Interaktion mit einem Computer im Wesentlichen über eine Konsole. Dies reduzierte die Ein- und Ausgabe eines Programms auf rein textuelle Elemente. Seither hat sich viel getan: Die Erschaffung des Internets leitet den Beginn von Webseiten ein, welche von einer anfänglich statischen Ausprägung zu der heutigen dynamisch und komplexen wuchsen. Dazu gesellten sich im Laufe der Zeit mobile Endgeräte - zu Beginn bestückt mit Tasten für die Eingabe, sowie einem primitiven Bildschirm für die Anzeige finden sich heutzutage vorwiegend leistungsfähige, auf einem kapazitiven Touchscreen basierende Smartphones wieder. Hierbei ist über die Jahre der Funktionsumfang von Betriebssystem und Applikationen im Allgemeinen gestiegen.

1.1 Problemfeld

Die Herausforderung für einen Entwickler ist es, die Nutzerschnittstelle (auch: Presenation Layer [2]) , innerhalb einer "Drei-Schichtenarchitektur"[3] sinnvoll aufzubauen und dabei die Modellierung und Verwaltung des Zustandes innerhalb einer Applikation zu berücksichtigen. Hierfür müssen auch externe Vorkommnisse wie bswp. die Aktualisierung der zugrundeliegenden Datenbank miteinbezogen werden.
Da diese Problematik nicht erst seit kurzem sondern seid vielen Jahren besteht, wurden hierfür

1.2 Ziel der Arbeit

1.3 Aufbau der Arbeit

2 Grundlagen

In diesem Kapitel gilt es zu klären, auf welchen Grundlagen, Ideen und Konzepten Model-View-Intent beruht, wie diese miteinander fungieren und weshalb sie als Inspiration dienten.

2.1 Unidirektionaler Datenfluss und der Zustand: Flux, Redux und Elm

In einer Applikation existieren grundsätzlich zwei Komponenten: Eine, die der Nutzer wahrnehmen kann und eine, die für ihn unsichtbar bleibt. Bei ersterer handelt es sich meist um das, was der Nutzer(auf dem Bildschirm) sieht - die sogenannte „View“. Die zweite Komponente beschreibt die Ebene, welche das Geschehen observiert, darauf reagiert und den weiteren Verlauft (zum größten Teil) kontrolliert. Sie kann unter anderem als „Controller“ betitelt werden.

Ein weiterer, essentieller Aspekt einer Anwendung ist ihr Zustand. Dieser kann sich aus mehreren Teilen zusammensetzten:

- Alles was der Nutzer sieht
- Daten die über das Netzwerk geladen werden
- Standort des Nutzer
- Fehler die auftreten
- …

Der Zustand in dem sich eine Applikation befindet kann hierbei von beiden Seiten modifiziert und beobachtet werden. Ist dies der Fall, so handelt es sich um einen bidirektionalen Datenfluss. Bei dieser Variante entsteht die eventuelle Gefahr von kaskadierenden Updates (ein Objekt verändert ein anderes, welches wiederum eine Veränderung bei einem weiteren herbeiführt usw.) als auch in einen unvorhersehbaren Datenfluss zu geraten: Es wird schwer, den Fluss der Daten nachzuvollziehen. Des weiteren muss immer überprüft und sichergestellt werden, dass „View“ und „Controller“ synchronisiert sind, da beide den globalen Zustand darstellen. Schlussendlich verliert man zusätzlich die Fähigkeit zu entscheiden, wann und and welcher Stelle der Zustand manipuliert wird.

Ein anderer Ansatz ist, den Datenfluss in eine Richtung zu beschränken und ihn damit unidirektional [4, 5] operieren zu lassen. Diese Variante erfreut sich an zunehmender Popularität seit der Bekanntmachung der „Flux“ [6] Architektur im Jahre 2015 von Facebook. [7]

2.1.1 Flux

Für die Einhaltung und Umsetzung eines unidirektionalen Datenfluss und der Verwaltung des Zustands bedient sich „Flux“ bei zwei fundamentalen Konzepten: Der Zustand

innerhalb einer Applikation wird als „single source of truth (SSOT)“ angesehen und darf keine direkte Änderung erfahren. Um dies zu Gewährleisten finden sich mehrere Komponenten in „Flux“ wieder:

Action: Eine Aktion beschreibt ein Ereignis, welches unter anderem vom Nutzer ausgelöst werden kann. Sie geben vor, wie mit der Anwendung interagiert wird. Jeder dieser Aktionen wird dabei ein Typ zugewiesen. Insgesamt sollte eine Aktion semantisch und deskriptiv bezüglich der Intention sein. Des weiteren können zusätzliche Attribute an eine Aktion gebunden werden.

```
{
 type: ActionTypes.INCREMENT,
 by: 2
}
```

Dispatcher: Er ist für die Entgegennahme und Verteilung einer Aktion an sogenannte „Stores“ zuständig. Diese haben die Möglichkeit sich beim ihm zu registrieren. Er besitzt die wichtige Eigenschaft der sequentiellen Verarbeitung, d.h., dass er zu jedem Zeitpunkt nur eine „Action“ weiterreicht. Sämtliche „Stores“ werden über alle Aktionen unterrichtet.

Store: Hier befinden sich die Daten, welche einen Teil des globalen Zustands einer Anwendung ausmachen. Die einzige Möglichkeit für eine Veränderung der dort hinterlegten Daten besteht durch ein Reaktion auf eine, vom „Dispatcher“ kommenden, Aktion. Bei jeder Modifikation der Daten erfolgt die Aussendung eines Events an eine „View“, das die Veränderung mitteilt. Ebenso findet sich hier ein Part der Anwendungslogik.

View: Die View ist für die Anzeige und Eingabe von Daten zuständig - sie ist die für den Nutzer sichtbare Komponente, mit welcher dieser interagiert. Ihre Daten erhält sie von einem „Store“, diesen sie abonniert und auf Änderungsereignisse hört. Erhält sie vom „Store“ ein solches Änderungsereignis, so kann sie die neuen Daten abrufen und sich selbst aktualisieren. Der View ist es nicht gestattet, den Zustand direkt zu verändern. Stattdessen generiert sie eine Aktion schickt diese an den Dispatcher.
Ein Beispielhafter Ablauf bei einer Anwendung die einen Wert erhöht oder verringert kann wie folgt aussehen:

1. Die View bekommt einem Store zugewiesen, welcher für das inkre- und dekrementieren der angezeigten Zahl verantwortlich ist.

2. Sie erhält die Anfangszahl und stellt diese in einem leserlichen Format/einer Ansicht dar, welches es dem Nutzer ermöglicht, damit zu interagieren.

3. Betätigt dieser einer der Knöpfe welche die dargestellte Zahl verändern, so wird eine Action erstellt und an Dispatcher geschickt.

4. Dieser wiederum informiert alle Stores.Information
5. Jener Store der für die Verarbeitung dieser Aktion verantwortlich ist, modifiziert die Zahl in seiner internen Datenstruktur und kommuniziert dies über ein Änderungsereignis
6. Diejenige View, welche auf Änderungsereignisse diesen Ursprungs lauscht, erhält die Daten und aktualisiert sich dementsprechend.

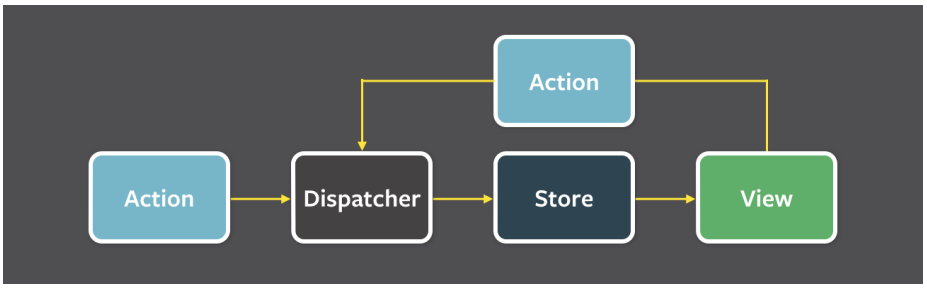

Abbildung 1: Datenfluss in der Flux Architektur

Anhand Abbildung 1 wird der unidirektionale Datenfluss deutlich erkennbar:

1. Die View schickt eine Aktion an den Dispatcher.
2. Dieser leitet diese an alle Stores weiter.
3. Der Store verarbeitet die Daten und informiert die View.

Insgesamt liefert Flux mit diesen Komponenten eine Möglichkeit, einen unidirektionalen Fluss herzustellen und die Verwaltung des Zustands einer Applikation zu vereinfachen.

2.1.2 Redux

Bei Redux handelt sich um eine JavaScript Bibliothek (und kein Framework) welche ihre Inspiration aus Flux und Elm bezieht. Sie wurde im Jahre 2015 von Dan Abramov und Andrew Clark ins Leben gerufen. [8] Auch hier nimm der direktionale Datenfluss eine wesentliche Rolle ein.

Die Bibliothek kann als eine vereinfachte Form von Flux verstanden werden, welche gewisse Elemente und Ansätze übernimmt, aber auch streicht bzw. ersetzt. Genau wie Flux existieren die bereits behandelten Actions, welche über wichtige Informationen für die spätere Veränderung des Zustands verfügen. Auch hier können diese ihren Ursprung in einer vom Nutzer getätigten Aktion haben. Wird eine Aktion ausgeführt, so spricht man von einer Versendung einer Aktion. Dieser Versand findet nur dann statt, wenn man die Intention verfolgt, den Zustand zu ändern. Sie gelangt zu einem sogenannten Reducer". Hier findet sich der erste Grundlegende Unterschied zu Flux.

Ein Reducer"ist für sich genommen eine einfache Funktion, die bei gleicher Eingabe die immer gleiche Ausgabe erzeugt. Sie ist dabei frei von sogenannten Seiteneffekten und wird als pure"bezeichnet. Im Falle eines Reducers"erwartet dieser die vorher erzeugte Aktion und den globalen, derzeitigen Zustand der Anwendung. Sein Aufgabe ist es, aus der Kombination dieser einen neuen Zustand zu generieren. Hierfür wird die Aktion, basierend auf ihrem Typ und eventuellen Inhalt, ausgewertet und der Zustand dementsprechend angepasst. Dabei ist zu beachten, dass keine direkte Manipulation des Zustands möglich ist, stattdessen wird ein komplett neuer Zustand zurückgegeben. Diese Eigenschaft der Unveränderbarkeit wird gemeinhin als "Immutability"erfasst.

```
1 (previousState, action) => newState
```

Das Verbindungsstück zwischen einer Aktion und dem Reducer bildet der Store. Dieser existiert im Gegensatz zu Flux nur ein einziges Mal (und mit auch der Zustand) und ist für die Verwaltung des Zustands verantwortlich. Er übernimmt zugleich auch die Rolle des Dispatchers, wie er in Flux vorkommt, und verteilt die Akionen an alle Reducer weiter. Der aus dem diesen Prozess hervorgehende, neue Zustand wird im Store hinterlegt. Dieses Ereignis wird ebenfalls seitens der View observiert, welche im Anschluss die nötigen Aktualisierungen an der Ansicht vornimmt. Am Ende lässt sich der gesamte Fluss wie folgt darstellen:

```
1 View -> Action -> Reducer(s) -> Store -> View
```

Neben Redux existieren in der JavaScript Welt noch weitere Bibliotheken, die entweder eine Abwandlungen von Flux darstellen oder aber neue Konzepte implementieren. Jedoch verfolgen dabei alle eine ähnliches Ziel: Ein unidirektionaler Datenfluss und die (zentrierte) Verwaltung des Zustands einer Anwendung.

2.1.3 Elm

Bei Flux und Redux handelt es sich jeweils um Bibliotheken die innerhalb einer Anwendung verwendet werden können. Eine weitere Herangehensweise ist die Verankerung solcher Konzepte in der Programmiersprache selbst. Dies findet man z.B. in der Programmiersprache Elm [9] wieder. Elm besitze eine in die Sprache integrierte Architektur die den einfach Namen The Elm Architecture"[10] trägt. Eine andere Variante lautet: Model-View-Update. Anhand dessen lassen sich bereits die Kern-Komponenten der Entwurfsmusters erahnen.

Model: Das Model repräsentiert den Zustand der Applikation als eine simple Datenstruktur.

View: Die View ist eine Funktion, welche aus einem Model HTML code generiert. Ebenfalls wie in Flux wird kommt auch hier das Konzept von pure functions"zum tragen: Die gleiche Eingabe erzeugt die gleich Ausgabe - ohne Ausnahme.

2.2 Funktionale Programmierung

Im Verlaufe der Kapitel wurden bereits Begriffe wie eine reineFunktion (pure functions) oder die Unveränderlichkeit (Immutibilty) einen Datenstruktur angesprochen. Diese Konzepte zählen zu einem Programmierparadigma, welches in den letzten Jahren auch an Bedeutung in Sprachen wie Java gewonnen hat [11] : Funktionale Programmierung.

In der häufig imperativen, Objektorientierte Programmierung wie sie in Java oder auch C# anzutreffen ist, machen Klassen und die Mutation (in möglicher Abhängigkeit von gewissen Konditionen) solcher den Hauptbestandteil des Quellcodes aus. Dabei ist zu beachten, dass die funktionale Programmierung die Objektorientierte nicht zwingenden ausschließt, sondern lediglich in eingeschränkter Form nutzt. Eine Implementierung in Java welche Beispielweise den Name eines Nutzer Objekts ändert, könnte wie folgt aussehen:

```
1 public User changeUserName(String newUserName, User currentUser){
2  if(newUserName != null){
3   currentUser.name = newName
4
5   System.out.println("Changed user name to: " + newUserName)
6  }
7
8  return currentUser
9 }
```

3 Model-View-Intent

Model-View-Intent (MVI) ist eine weiteres Entfursmuster, welches der Feder von André (Medeiros) Staltz entstammt. Er stellte dieses auf einer Javascript Konferenz im Jahre 2015 vor [12]. Es gehört damit zu den jüngsten seiner Art. Seine ursprüngliche Anwendung fand es in dem ebenfalls von André Staltz geschaffenen Framework CycleJs, hat seit der Artikelreihe von Hannes Dorfmann in 2016 [13] aber auch den Sprung in die Welt von Android vollbracht. MVI bezieht den Großteil seines Design aus den hier bereits vorstellten Ideen und Konzepten. Das erste Ziel ist, ähnlich wie bei MVC, Informationen zwischen zwei Welten zu übersetzten: der des digitalen Bereichs des Computers und des mentalen Modells des Benutzers. Oder anders formuliert: Das Programm muss verstehen, was der Nutzer im Sinn hat. Der zweite, zentrale Punkt von MVI besteht in der Handhabung des Zustands der Anwendung. Hierfür ist zu klären, was genau der Zustand inne hat.

MVI betrachtet dafür die Interaktion zwischen einem Nutzer und Programm als einen Kreis(lauf). Betätigt der Nutzer bswp. einen Knopf, sein Output, so gestaltet sich dieser als Input für das Programm. Dieses wiederum erzeugt einen Output (z.B. eine Meldung), welcher zum Input des Nutzers wird (hier: lesen).

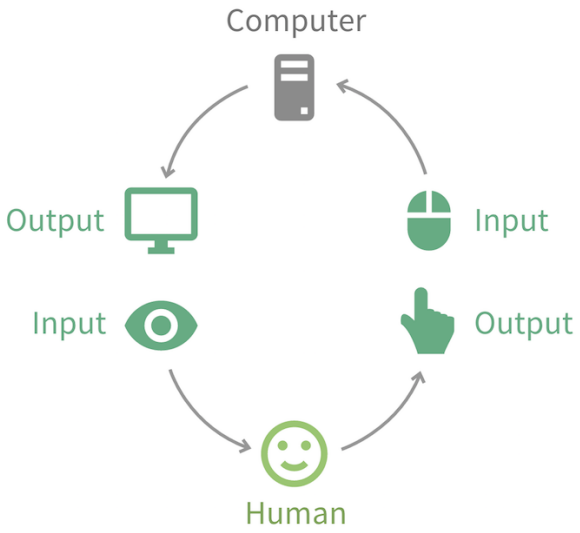

Abbildung 2: Nutzer und Computer als Input und Output

Source: https://cycle.js.org/dialogue.html

Die Grundprinzipien für den Aufbau der Architektur entspringen dabei dem beschriebenen, originalem Model-View-Controller. Das, was MVC jedoch inkompatibel für die von MVI vorhergesehenen Prozesse macht, ist die Tatsache, das der Controller proaktiv ist. Dies bedeutet, dass der Controller selbstbestimmt über das Model und View verfügen

und diese direkt manipulieren kann. Zwangsläufig wissen die jeweiligen Komponenten auch, von welchen Komponenten sie abhängig sind. Oder anders ausgedrückt: Eine Komponente deklariert, welche anderen Komponenten sie beeinflussen, anstatt dass andere Komponenten explizit aktualisiert werden (z.B. das Modell). Dabei wird das Prinzip des unidirektionale Datenfluss verletzt, welches auch in MVI strikt verfolgt wird.

Um dieses unter anderem zu erreichen, setzt MVI zusätzlich auf Reaktive Programmierung. Für MVI bedeutet reaktiv zu sein, dass jede Komponente ihre Abhängigkeiten beobachtet und auf Veränderungen dieser reagiert. Die drei Komponenten werden durch Observables repräsentiert, wobei der Output jeweils der Input einer anderen Komponente ist.

Fast noch wichtiger als der reaktive Ansatz ist zu verstehen, wie der Kreislauf in Figur 1 programmatisch etabliert werden kann. Betrachtet man diesen Kreis etwas genauer, so wird deutlich, dass auf einen Input immer ein Output folgt. Dieses Konzept findet man auch in der Mathematik wieder: Funktionen. Mit diesen lässt sich MVI wie folgt illustrieren:

Intent: Das I in MVI steht für Intent und stellt den Teil da, welches es von den anderen Entwurfsmustern unterscheidet. Das Ziel der Intent-Fuktion ist es, die Absicht des Nutzer im digitalen Kontext des Programms auszudrücken. Ein Ereignis (oder Event), z.B. die Eingabe eines Buchstaben, kann hier der Input sein. Der Output dieser Funktion (z.B. ein String) wird zum Input der nächsten:

Model: Die Model-Funktion nimmt das entgegen, was die Intent-Funktion produziert. Ihre Aufgabe liegt in der Verwaltung des Zustands: Sie verfügt über das Model. Sie kann daher durchaus als das zentrale Element in MVI bezeichnet werden. In Anbetracht der Tatsache, dass MVI sich als auf funktionaler Programmierung basierendes Muster versteht, ist das Model unveränderlich. Daraus ergibt sich zwangsläufig, dass für einen Zustandswechsel das Model kopiert und somit ein neues erzeugt werden muss.
Diese Funktion ist der einziges Teil des Programms, welche eine Zustandsveränderung hervorrufen kann und darf. Zusätzlich ist es der Ort, an dem auf die Business Logik der Anwendung zugegriffen wird.

View: Die View ist die letzte Funktion in der Kette, und ist zuständig für die visuelle Repräsentation des Models.

Nimmt man alle drei Funktion zusammen, ergibt sich folgende Kette:

Listing 1: funktion

```
view(model(intent(input)))
```

Um den Sachverhalt zu verdeutlichten, kann dieses Beispiel in Form von pseudo-code herangezogen werden:

Listing 2: pseudo mvi implementation

```
fun intent(text: String): Event {
 return EnteredTextEvent(text)
}

fun model(event: Event): Model {
 return when(event){
  is EnteredTextEvent -> {
   val newText = event.text.trim() // <-- business logic
   model.copy(text = newText) // <-- immtuable data structure
  }
 }
}

fun view(model: Model){
 textView.text = model.text
}

fun main(args : Array<String>) {
 view(model(intent("Hello World")))
}
```

Bei dieser Implementierung ist jedoch schnell ersichtlich, das es sich hierbei um keinen Kreis(lauf) handelt. Jede Funktion wird nur einmal aufgerufen. Es fehlt der reaktive Part, der MVI unter anderem ausmacht. Um diesen zu realisieren muss der Beispiel-Code wie folgt abgeändert werden:

Listing 3: pseudo mvi implementation

```
// the obersvable from the textView gets passed as a parameter
fun intent(text: Observabkle<String>): Observable<Event> {
 text.map { text -> EnteredTextEvent(text) }
}

fun model(event:  Observable<Event>): Observable<Model> {
 event.map { event ->
  return when(event){
   is EnteredTextEvent -> {
   val newText = event.text.trim() // <-- business logic
   model.copy(text = newText) // <-- immtuable data structure
  }
 }
```

```
}

fun view(model: Observable<Model>){
 // we subscribe to the model to listen for changes
 model.subscribe { model ->
  textView.text = model.text
 }
}

fun main(args : Array<String>) {

 // this listens to arbitary text changes
 val textChanges: Observable<String> = textView.changes()

 view(model(intent(textChanges)))
}
```

Ein Punkt der in dieser Implementation noch offen bleibt, ist woher das Model kommt wie es verwaltet wird.

3.1 Reducer

Schaut man sich die Model-Funktion in Beispiel 3 und ihren Inhalt genau an, so wird ein bestimmtes Muster bzw. ein sich wiederholender Ablauf erkennbar:

1. Die Funktion erhält ein Event
2. Die Funktion evaluiert das Event
3. Die Funktion führt basierend auf dem Event (Business) Logik aus
4. Die Funktion erzeugt ein neues Model
5. Die Funktion gibt das neue Model zurück

Der einzige Schritt der Fehlt, ist die Bereitstellung des derzeitigen oder des vorherigen Models. Hier kommt eine Komponente ins Spiel, die bereits in Kapitel 2.1.2 angesprochen wurde: der Reducer.

3.2 Endlicher Automat

Wenn der in der Model-Funktion ausgeführte Code eine neues Model hervorbringt, so bleibt der Zustand entweder der Gleiche oder er verändert sich. Unabhängig davon geht der Zustand in den selbigen oder in einen neuen Zustand über: es kommt zu einem sogenannten Zustandsübergang. Aber nicht nur das lässt sich aus dem gezeigten Beispiel ableiten; es sind noch weitere Schlussfolgerungen zulässig:

- Es gibt immer einen Anfangszustand bzw. Startzustand
- Es gibt eine endliche Anzahl von Zuständen
- Es gibt eine beliebe Menge von Endzuständen
- Es gibt immer nur einen Zustand, in dem sich die Anwendung befinden kann

Die oben genannten Punkte lassen sich anhand eines weiteren Beispiels besser erläutern: Man nehme an, dass sich auf einem Bildschirm ein Textfeld und ein dazugehöriger Knopf befindet. Das Textfeld ist anfänglich leer und der Knopf deaktiviert. Dieser kann nur aktiviert werden, wenn eine Eingabe im Textfeld erfolgt. Hiermit ist der Startzustand des Knopfs "deaktiviert"(z0). Gibt der Nutzer im Textfeld einen Text ein, so wird der Knopf aktiviert. Dieses Ereignis ist der bereits angeführte Zustandsübergang. Gleichzeitig wird ein Endzustand erreicht (z1) - weitere Eingaben führen zu keinem neuem Zustand. Die Beschreibungen z1 und z2 dienen hierbei als das Eingabealphabet und zeigen die Menge von potenziellen Ereignissen auf.

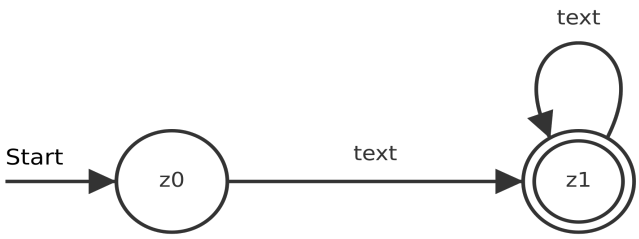

Abbildung 3: Endlicher Automat

Dieses Konzept ist in der Informatik bekannt als "Endliche Automaten". Ihr Ziel ist es, ein bestimmtes Verhalten (wie das obige) zu Modellieren und unter anderem visuell in Form von Abbildung 3 zu präsentieren.

4 Anforderungsanalyse

4.1 Funktionale Anforderungen

Die Funktionalen Anforderungen dienen dafür, um die genaue Funktionalität und das Verhalten eines Systems zu beschreiben. Es wird behandelt, was das System können soll und muss. Um dies besser abbilden zu können, werden die einzelnen Anforderungen einer Gewichtung unterzogen. Diese Gewichtung lässt sich in Form von Muss","Soll"und "Kann"Anforderungen ausdrücken. Aufgrund des kleines Rahmens und Zeitfensters dieser Thesis wird sich im diesen Teil ausschließlich auf MussAnforderungen beschränkt, d.h. jene Funktionalität, welches das Framework erfüllen muss. Für die Übersichtlichkeit werden sämtliche Anforderungen nummeriert und mit dem Kürzel FA"versehen.

[A01] Identifizieren von "Intents"
Dem Nutzer des Frameworks muss es möglich sein, die "Intents"seiner Anwendung eindeutig zu markieren.

[A02] Trennung von Bussines und Ansicht Logik
Ähnlich wie in MVP oder MVC muss das Framework eine strikte Separierung von (Bussines) Logik und der Ansicht Logik fördern.

[A03] Überprüfung des Models als unveränderliche Datenstruktur
Es ist zwingende erforderlich, dass die vom Nutzer gewählte Struktur für das Model nicht direkt verändert werden kann. Hierfür muss das Framework verifizieren, dass es sich um eine unveränderliche Datenstruktur handelt.

[A04] Bereitstellung eines Reducers"
Für das erzeugen von einem neuem Model muss dem Entwickler eine Funktionen in Form eines Reducers bereitgestellt werden. Diese muss das derzeitige Model und Ereignis zur Verfügung stellen.

[A05] Handhabung von Seiteneffekten
Auf Grund der Tatsache, dass in den meisten Anwendungen Seiteneffekte auftreten, muss dieses vom Framework abgedeckt sein.

[A06] Verwalten des Zustands
Ein zentraler Bestandteil des Framework ist es, das Model bzw. den Zustand der Anwendung zu Verwalten.

[A07] Funktion für die Aktualisierung der Ansicht
Nachdem ein neues Model erzeugt wurde, muss dieses an die Ansicht weitergegeben werden, welches sich daraufhin aktualisiert. Hierfür muss das Framework eine dementsprechende Funktion bereitstellen.

[A08] Speichern des Zustands
Der Zustand muss im transienten und persistenten Speicher abgelegt werden.

[A09] Wiederherstellung des Zustands
Sollte es zu einem Verlust der Models kommen, muss dieses ordnungsgemäß und wiederhergestellt werden. Dies sollte ohne Eingriff des Entwicklers von statten gehen.

[A10] Asynchrone Ausführung
Viele der Zugriffe auf eine Datenbank oder einer Rest-Api finden auf unterschiedlichen Threads statt. Hierbei darf es zu keinen unerwarteten Problemen (z.B. Race.Conditions) kommen.

4.2 Nicht funktionale Anforderungen

Die nicht funktionale Anforderungen sind im Gegensatz zu den funktionalen unspezifisch für ein Produkt, d.h. sie haben meist nur einen indirekten Einfluss auf das System. So kann beispielsweise festgelegt werden, dass die Ausführung einen bestimmten Funktionalität nur ein gewisses Maß an Zeit in Anspruch nehmen darf. Dazu gehören auch Qualitätsmerkmale, die erfüllt werden müssen.Für die Übersichtlichkeit werden sämtliche Anforderungen nummeriert und mit dem Kürzel NFA"versehen.

[A11] Dogmatisches Framework
Das Framework soll den Entwickler "and die Hand nehmen"und genaue Vorgaben für die Anwendung kommunizieren. Damit sind zielgenauere Funktionen möglich und die Komplexität kann unter Umständen niedriger gehalten werden. Dies birgt allerdings die Gefahr, dass bei eigenwilliger Anwendung - und der damit einhergehenden Abweichung der Instruktionen - des Entwicklers es zu Komplikationen kommen kann.

[A12] Unidirektional
Der in MVI geforderte unidirektionale Datenfluss muss eingehalten werden.

[A13] Kotlin spezifische Implementierung
Für die Umsetzung wird Kotlin als Programmiersprache herangezogen. Sie bietet einige syntaktische Vorteile gegenüber Java.

[A14] Reaktiv
Reaktiv

[A15] Funktional
Funtkional

4.3 Ubersicht der Anforderungen

Für eine bessere Übersicht der funktionalen (fa) und nicht funktionalen (nfa) Anforderungen werden diese im Folgenden in einer Tabelle zusammengefasst. Zusätzlich wird ein Verweis auf das Design bzw. Konzept und die jeweilige Implementierung angefügt.

ID	Anforderung	Typ	Konzept	Impl.
.1em.1em.1em 01	Identifizieren von ”Intents”	fa	5.1	6.1
02	Trennung von Bussines und Ansicht Logik	fa	5.2	6.2
03	Überprüfung des Models als unveränderliche Datenstruktur	fa	5.3	6.3
04	Bereitstellung eines Reducers”	fa	5.4	6.4
05	Handhabung von Seiteneffekten	fa	5.5	6.5
06	Verwalten des Zustands	fa	5.6	6.6
07	Funktion für die Aktualisierung der Ansicht	fa	5.6	6.6
08	Speichern des Zustands	fa	5.6	6.6
09	Wiederherstellung des Zustands	fa	5.6	6.6
10	Asynchrone Ausführung	fa	5.6	6.6
11	Dogmatisches Framework	fa	5.6	6.6
12	Unidirektional	fa	5.6	6.6
13	Kotlin spezifische Implementierung	fa	5.6	6.6
14	Reaktiv	fa	5.6	6.6
15	Funktional	fa	5.6	6.6

5 Design & Konzept

In diesem Kapitel ...

5.1 Grundlegende Designentscheidungen

Bevor auf Entscheidungen eingegangen wird ...

5.1.1 Android als Plattform

Das Framework richtet sich ausschließlich an Entwickler die Applikationen für die Plattform Android entwickeln. Es ist damit nicht kompatibel zu iOS, dem Web oder Serverseitigen Anwendungen. Die Spezialisierung lässt es jedoch zu, besser auf mögliche Eigenheiten der Plattform einzugehen. Ein weiterer Grund für diese Entscheidung stellt die Tatsache dar, dass MVI seinen Anfang in der Entwickelung von Webseiten fand und es sich im Fall Android um einen Nachzügler handelt.

5.1.2 Framework

Warum Framwork? Besonderheiten...

5.1.3 Kotlin als Programmiersprache

Die Applikationen in Android und das Android-SDK selbst sind bis vor wenigen Jahren fast ausschließlich in der Sprache Java entwickelt wurden. Seit der Google I/0 2017 gehört jedoch eine weitere Sprache zu den offiziell unterstützen: Kotlin. Sie wird von dem Unternehmen Jetbrains entwickelt, die untere anderem die Entwicklungsumgebung Intellij für Java produzieren. Dieses bildet auch die Grundlage für Android Studio. Kotlin hat in den letzten Jahren an Bodenhaftung gewonnen und findet auch intern bei Google Verwendung.
Die Sprache wird als statisch typisierte, objektorientierte Programmiersprache bezeichnet und verfügt über eine hohe Interoperabilität zu Java. Dies bedeutet, dass innerhalb eines in Java geschriebenen Programms ohne viel Aufwand Kotlin genutzt werden kann. Dies ist ein wichtiger Faktor für die immer weiter ansteigende Beliebtheit, da es eine einfache Integration und bisherige Projekte gestattet. Kotlin bringt ein verbesserte Syntax mit und macht beispielsweise die Verwendung von null”explizit. Zu den Verbesserungen gehören dabei auch:

- Ableitung von Typen
- Alles ist eine Expression
- Funktionen sind First-Class-Funktionen”und bilden eine Funktionale Grundlage
- Datenklassen machen den Umgang mit unveränderliche Datenstrukturen einfach

- Erweiterungsfunktionen
- Kovarianz und Kontravarianz werden explizit anwendet
- Standardwerte für Parameter

Listing 4: Kotlin Beispiel

```kotlin
data class Example(
  privat val defaultMessage: String = "Hello World"
  privat val maybeNull: String? = null
){

  // Expression
  fun isHelloWorld() = when(message){
    "Hello World" -> true
    else -> false
  }

  // "?" findet bei null verwendung
  fun printIfNotNull() {
    maybeNull?.run {
     print(this)
    }
  }
}

// Erweiterungsfunktion und Funktion als Parameter
fun HelloWorld.extensionFunction(function: () -> String) {
 val message = function()
 println(message)
}

// kein new Schlüsselwort nötig
// keine Semikolon nötig
val example = Example()

// copy wird automatisch generiert bei einer "data" Klasse
val newExample = example.copy(defaultMessage = "New Message")

// ist der letzte Parameter eine Funktion, so kann auf Klammern
// verzichtet werden
newExample.extensionFunction { "Hello" }
```

Insgesamt is anhand Listing 4 zu erkennen, das Kotlin eine deutlich prägnantere und schlankere Syntax besitzt. Sie vermeidet damit einen großen Teil des mit Java verbundenen ”Boilerplate-Codes”und kann für eine höheren Grad an Produktivität sorgen. Besonders der Umgang von Null als Teil des Typsystems kann vor der berühmten NNullpointer-Exceptionretten. Des weiteren besteht ein größerer Fokus auf dem Einsatz von Konzepten aus der funktionalen Programmierung, welche durch Erweiterungsfunktionen für bswp. Listen zum Einsatz kommen.

5.2 Model & Zustand

Ähnlich wie bei Redux nimmt das Model in MVI und somit der Zustand die zentral Rolle innerhalb der Anwendung ein. Es diktiert das User Interface (UI) und hat wesentlichen Einfluss auf die ausführende Business-Logik.

Für die Repräsentation des Models gibt MVI vor, das dieses Unveränderlich sein muss. Dies wiederum hat zur Folge, dass für jede Zustandsüberführung ein neues Model auf Basis des alten bzw. derzeitigen erzeugt werden muss. Hierfür wird das Model kopiert und während diesem optional Vorgang mit neuen Werten versehen.

Für dieses Szenario stellt Kotlin eine Struktur zur Verfügung, welche den Prozess stark vereinfacht: Die ”data”Klasse. Sie generiert unterschiedliche Methoden welche dem Nutzer ohne weiteres Zutun zur Verfügung stehen. Darunter befindet sich unter anderem die Methode ’copy’. Wie der Name vermuten lässt, erzeugt sie eine Kopie der Klasse. Dabei können der Methode die Parameter übergeben werden, welche der Konstruktor der Klasse inne hat. Dies macht es möglich einzelne Attribute neu zu besetzten und bei den übrigen den derzeitigen Wert beizubehalten. Wie Listing 5.2.3 aufzeigt, gestaltet das den Umgang mit Unveränderlichen Klassen verhältnismäßig einfach.

Listing 5: data class

```
1
2 data class MyClass(val text: String, val anotherText: String)
3
4 val myClass = MyClass("text", "anotherText")
5
6 // 'text' wird verändert, 'anotherText' nicht
7 val newMyClass = myClass.copy(text = "newText")
```

Eine Problematik die herbei jedoch besteht ist, dass bei der ’data’ Klasse unveränderliche Attribute (= val) keine Pflicht darstellen. Somit wäre es dem Entwickler ohne weiteres Möglich, die Instanz direkt zu modifizieren, ohne ein neue zu erzeugen müssen. Um das zu Verhindern gilt es Sicherzustellen, dass das vom Entwickler angedachte Model den Anforderungen der Unveränderlichkeit entspricht. Für diesen Zweck kann aus zwei ’Werkzeugen’ gewählt werden: Der Reflexion (oder Introspektion) und dem generieren von Code.

5.2.1 Reflexion (Introspektion)

Bei dieser Variante ist es dem ausführenden Programm erlaubt seine eigene Struktur - zur Laufzeit - zu analysieren oder auch zu verändern (und Sichtbarkeitseinschrankungen zu umgehen). Es gestattet einem, Informationen über Klassen dynamisch auszulesen. Das beinhaltet Modifier, Variablen, Konstruktoren, Methoden, Annotationen (= reflektive Informationen) usw. Reflexion hat dabei vielfältige Anwendungszwecke:

1. Debugger
2. Interpreter
3. Objekt Serialisation
4. Dynamisches Laden von Code/ erzeugen von Objekten (z.B. Spring @Autowired)
5. Java Beans

In Kotlin muss beachtet werden, dass zwei Schnittstellen für den Einstieg in die Reflexion existieren. Einmal die Standard Schnittstelle für Java, und einmal jene für Kotlin. Erstere erlaubt die Arbeit mit allen Java Konstrukten und zweitere die, die Kotlin exklusiv sind. Für jede Klasse existiert zur Laufzeit ein Objekt des Typs 'Class<T>' oder KClass<T>. T ist dabei der Typ der zu untersuchenden Klasse.

Für den Zugriff auf die Java Reflexion muss auf die '.java' Endung zurückgegriffen werden, wie folgendes Beispiel verdeutlicht:

Listing 6: Java Reflexion

```
1
2 val myClass: Class<MyClass> = MyClass::class.java
3
4 // gibt Methoden wie 'toString' aus
5 myClass.methods.forEach(::println)
```

In Kotlin wird über den 'double-colon' Operator auf die Reflexion zugegriffen:

Listing 7: Kotlin Reflexion

```
1
2 val myClass: KClass<MyClass> = MyClass::class
3
4 // gibt den Namen der Klasse aus
5 println(myClass.qualifiedName)
6
7 // 'false', da 'text' nicht Konstant ist
8 println(myClass::text.isConst)
```

Eine weiterer Interessanter Ansatz ist die Arbeit mit sogenannten Metadaten. Sie stellen Information über Informationen dar. Ein häufiger Anwendungsfall ist die Arbeit mit Annotationen, welche mit '@' eingeleitet werden. So wird mit der 'override' Annotation dem Compiler in Java mitgeteilt, dass diese Methode überschrieben wurde:

Listing 8: Override Annotation

```
1
2  class MyClassJava {
3
4  @Override
5  public String toString() {...}
6  }
7
8  class MyClassKotlin {
9
10 // Hier ist 'override' teil der Deklaration
11 override fun toString() {...}
```

Anhand der Beispiele lässt sich erahnen, dass Reflexion beträchtlichen Einfluss auf die Anwendung haben kann und viele Türen öffnet. Wie üblich ergeben sich dabei gewisse Vor- und Nachteile:

Vorteile:

- Analysieren und modifizieren von Klassen zur Laufzeit
- Erweiterbarkeit durch die Nutzung von Annotationen

Nachteile:

- Der Zugriff auf private APIs kann ein Sicherheitsrisiko darstellen
- Es kann nicht überprüft werden, inwieweit ein korrekter Datentyp vorliegt
- Es kann die Ausführungsgeschwindigkeit negativ beeinflussen, da die JVM Optimierungen nicht durchführen kann

Angesichts dieser Auflistung lässt sich festhalten, dass der Gebrauch von Reflexion auf das notwendige Maß beschränkt werden sollte, da es grundlegende Prinzipien der typisierten Programmierung verletzt.

5.2.2 Code Generation

Ein andere Methodik besteht in der Generierung von Code. Hierbei wird ein Programm geschrieben, welches wiederum ein anderes Programm erzeugt Ein Grund kann z.B. ein repetitiver Prozess sein, der somit automatisiert werden kann. Eine beliebtes Verfahren

ist das Produzieren von Klassen basierend auf '.json' Dateien.

Mithilfe der Bibliothek 'KotlinPoet' kann durch folgenden Code...

Listing 9: Code zum erezugen einer Funktion

```
FunSpec.builder("add")
.addParameter("a", Int::class)
.addParameter(ParameterSpec.builder("b", Int::class)
.defaultValue("%L", 0)
.build())
.addStatement("print(\"a + b = ${ a + b }\")")
.build()
```

folgender Code erzeugt werden:

Listing 10: Erzeugte Funktionen

```
fun add(a: Int, b: Int = 0) {
print("a + b = ${ a + b }")
}
```

welcher in einer '.kt' abgelegt wird. Auf diese Funktion haben sämtliche Klassen zugriff. Vor- und Nachteile...

5.2.3 Überprüfung durch Reflexion

Um die Wartung von generiertem Code, dem Aufwand der Implementierung und damit einhergehenden Zeitaufwand zu umgehen, fällt die Entscheidung auf Reflexion. Im ersten Schritt muss verifiziert werden, dass es sich bei dem Model um eine 'data' Klasse handelt. Hierfür bietet die Koltin Reflexion API ein Attribut:

Listing 11: Kotlin 'isData' Attribut

```
data Class DataClass

println(DataClass()::class.isData) // true
```

Im weiteren Vorgehen müssen die einzelnen Attribute der Klasse auf ihre Unveränderlichkeit überprüft werden. Um dies zu erreichen, kann auf Basis folgender Schritte eine Implementierung stattfinden:

1. Liste sämtlicher Attribute der zu prüfenden Klasse erstellen
2. Filtern von nicht benötigten Attributen
3. Prüfen, inwiefern das Attribut 'var' als Zugriffsmodifikator verwendet wird

   Bei Gebrauch zusätzlich den Typen auf Unveränderlichkeit prüfen

4. Typen auf Unveränderlichkeit prüfen

5. Eine Fehlernachricht generieren, mit den veränderlichen Attributsnamen

Die Funktion zu Überprüfung wird dabei als Extension auf der Klasse 'KClass', Kotlins Klasse für Metadaten, realisiert. Diese besitzt einen generischen Parameter und erwartet einen Typ. Damit sie für alle Typen gültig ist, wird als Typ das Sternzeichen hinterlegt. Es handelt sich dabei um einen sogenannte 'Wildcard' und sagt aus, das beliebige Typen möglich sind.
Bei der 'data' Klasse wird für jedes Attribut ein zusätzliches generiert, das mit dem Wort 'component' beginnt und am Ende eine Zahl von eins bis n (= Anzahl der Attribute) stehen hat. Dadurch kann man das Verfahren der destrukturierenden Zuweisung anwenden. Hierbei werden aus dem Objekt Daten extrahiert und in (mehreren) Variablen abgelegt, wie Listing **??** zeigt.

Listing 12: Destrukturierende Zuweisung

```
val pair = Pair("forename", "surname")

val (forename, surname) = pair
```

```
fun KClass<*>.checkForImmutability(): String =
members
.filter { it.name.startsWith("component").not() }
.fold("") { errorMessage: String, property: KCallable<*> ->
when (property) {
is KMutableProperty<*> -> {
val tmpErrorMessage = "$errorMessage\n${property.name} is not allowed to be '
checkSubTypeForImmutability(property, tmpErrorMessage)
}

else -> checkSubTypeForImmutability(property, errorMessage)
}
}

fun checkSubTypeForImmutability(property: KCallable<*>, errorMessage: String)
when {
property.isSubtypeOf(MutableIterable::class) ->
"$errorMessage\n${property.name} is not allowed to be mutable"

property.isSubtypeOf(Function::class) ->
"$errorMessage\n${property.name} is not allowed to be a function"

else -> errorMessage
}

```

```
26 fun KCallable<*>.isSubtypeOf(clazz: KClass<*>): Boolean =
27 returnType.isSubtypeOf(clazz.starProjectedType)
```

Das Model muss ihm Controller als generischer Typ angegeben werden.

5.3 Intent

Jeder Intention geht ein Ereignis voraus, das entweder vom Nutzer oder der Anwendung selbst initiiert wurde. Es stellt dabei den Einstieg in den von MVI definierten Kreislauf (aus Abbildung) dar.
Klickt der Nutzer in einer Anwendung auf einen ”Zurück”Knopf, so ist seine Intention zum vorherigen Bildschirm zurückzukehren oder die Anwendung zu beenden. Dieses Ereignis kann ohne weitere Informationen stattfinden. Anders ist es, wenn seitens des Nutzers innerhalb eine Liste ein Item ausgewählt wird und dessen Details gelistet werden sollen. Hierfür muss zusätzlich zu der eigentlichen Intention das ausgewählte Item (oder seine ID) übermittelt werden.
Daraus ergeben sich zwei Arten von ”Intents”: Eines ohne und eines mit zusätzlichen Nutzdaten (Englisch payload). Dies bedeutet, dass eine Struktur existieren muss, die entweder Daten beinhaltet oder nur eine semantische Bedeutung hat.

Listing 13: Intent Klasse

```
1 class Intent<T> (val payload: T)
```

Für diesen Fall eignet sich eine Klasse mit einem generischen Typ als Attribut, wie Listing 12 zeigt. Das «T>”in der Klassen Deklaration dient dabei als Platzhalter für den eigentlich Typ, z.B. für ein ”Item”aus dem obigen Beispiel.
Die aufgezeigte Option weißt jedoch gewisse Mängel auf:

1. Der Payload”darf niemals null”sein
2. Der Name ”Intenttransportiert die eigentliche Absicht nicht

Mangel Nr. 1 lässt sich mit einer einfachen Abwandlung von Listing 12 behoben werden.

Listing 14: Intent Klasse

```
1 class Intent<T> (val payload: T? = null)
```

Hierfür muss lediglich von Kotlins Notation für nullTypen gebraucht gemacht werden: Das Fragezeichen. Um zu vermeiden, das bei dem erstellen einer Klasse ohne Inhalt stets null”übergeben werden muss, wird ein Standardparameter verwendet. Listing 15 stellt die Anpassungen dar.
Für den zweiten Mangel gibt es unterschiedliche Ansätze:

1. Ein zweites Attribut das die Absicht beschreibt
2. oder eine eigene Klasse für jede Intention

Listing 15: Intent Enum

```
enum class IntentDescription {
 GO_BACK, DISPLAY_ITEM_DETAILS
}
```

Bei Ansatz Nummer Eins ist ein potenzieller Kandidat die Verwendung eines ”enum”. Diese kann durch Konstanten (Listing 14 gibt einen Eindruck) zum Ausdruck bringen, um welche Intention es sich handelt. Im nächsten Schritt muss das Enum als Attribut in der Intent Klasse hinterlegt werden:

Listing 16: Intent Klasse

```
class Intent<T> (
  val payload: T? = null,
  val description: IntentDescription
)

// die Explizite Angabe von Atrributsnamen
// bei weglassen von anderen Attrbuten ist
// best practice

val intent = Intent(description = IntentDescription.GO_BACK)
```

Aber auch diese Lösung ist suboptimal: Ungeachtet der Absicht ist immer ein pp-ayload”von Nöten, selbst wenn dieser für das weitere Vorgehen nicht verwendet wird. Dies mag in wenigen Fälligen vertretbar sein, wird bei einer hören Anzahl an Intention unübersichtlich. Zusätzlich sollte immer versucht werden null”Werten aus dem Weg zu gehen, wenn nicht zwingend erforderlich. Dies verringert die Gefahr unerwartet der NN-ullPointer Exception”über den Weg zu laufen.

Ein besser Ansatz bildet dabei Nummer zwei. Anstatt mit einer Klasse sämtliche Intentionen abbilden zu wollen, erscheint es sinnvoller für jede Intention eine Klasse zu kreieren. Bei dieser Variante ergeben sich zwei Eigenschaften: Jeder dieser Klassen stellt übergeordnet einen ”Intent”dar und enthält möglicherweise einen payload”, welcher niemals null”ist.
Für dieses Szenario existiert ein Konzept das aus zwei Konstrukten hervorgehen kann. Das erste trägt den Namen Produkt”und charakterisiert eine fundamentale Eigenschaft der Objekt Orientierten Programmierung. Es sagt aus, das mehrere unterschiedlichen Werte ein einzigen Wert bilden können. Darunter fällt bswp. eine Klasse in Java oder Kotlin.

Das zweite Konstrukt, die Summe von Werten liegt vor, wenn anstatt von mehreren Werte zu einem entweder ein Typ oder ein anderer vorliegt. Es ist somit keine Kombination von Werten wie beim Produkt, sondern die Entscheidung für einen der Angegebenen. Das Enum wie in Listing 16 gehört unter anderem zu den Summen Typen.

Listing 17: Summen Typ

```
enum class Color(val rgb: Int) {
 RED(0xFF0000),
 GREEN(0x00FF00),
 BLUE(0x0000FF)
}

val color: Color = Color.RED

print(color is RED) // true
print(color is GREEN) // false
```

Vereint man beide Konstrukte, so ergibt sich die Idee eines algebraischen Daten Typen, der zusätzlich auch primitive Werte umfassen kann. Das Ziel ist, Daten die zusammengehören und einen gemeinsamen Nenner besitzen in einer übersichtlichen und transparenten Form darzustellen. Der Grund, warum diese Typen als ”algebraisch”bezeichnet werden, ist, dass man neue Typen erschaffen kann, indem die ”Summe”oder das PProdukt”bestehender Typen nimmt.

In Kotlin existiert für diesen speziellen Fall eine bestimmte Form der Klasse, die ihn ihrer Deklaration mit dem ”sealed”Schlüsselwort eingeleitet wird. Dies macht es möglich, Listing 15 funktionaler und eleganter zum implementieren:

Listing 18: Intents als sealded class

```
sealed class Intent {
 // 'object' erzeugt ein Singleton
 // es ist keine Instanziierung möglich
 // Attribute folglich nicht gestattet
 object GoBack : Intent()
 class DisplayItemDetails(item: Item): Intent()
}
```

Mit diesem Ansatz verschwindet die Notwendigkeit für einen generischen Typ und das Vorhandensein von null”Werten. Des weiteren ist mit einem Blick erkennbar welche Intents vorkommen, sowie ihre Bedeutung und, wenn definiert, ihr Inhalt. Anders ausgedrückt dienen versiegelte Klassen zur Darstellung eingeschränkter Klassenhierarchien. Dann, wenn ein Wert einen Typ aus einer begrenzten Menge haben kann, aber keinen anderen Typ haben darf. Sie sind in gewisser Weise eine Erweiterung der Enum-Klassen:

Der Wertebereich für einen Enum-Typ ist ebenfalls eingeschränkt, aber jede Enum-Konstante existiert nur einmal. Eine Unterklasse einer versiegelten Klasse kann derweil mehrere Instanzen haben, die überdies einen Zustand enthalten kann. Zu beachten gilt außerdem, das in Kotlin sich dieses Konstrukt innerhalb einer Datei befinden muss.

5.4 View Model

Die Erzeugung eines Intents findet entweder in einer "Activity", einem Fragment"oder Derivaten statt. Infolge der Anlehnung von Model-View-Intent and Model-View-Controller muss eine Komponente existieren, welche die Business Logik innehat, die Intents entgegennimmt und sie im Kontext der Anwendung übersetzt. Weiterhin muss die Komponente reaktiv sein und den geforderten unidirektionalen Fluss einhalten.

Im ersten Schritt muss eine zentrale Funktion definiert werden, die als Parameter einen Intent erhält. Diese muss vom Entwickler überschrieben und implementiert werden. Zu diesem Zweck wird vom Konzept der Vererbung und dem oft damit verbundenen Polymorphismus (griechisch für Vielgestaltigkeit) Gebrauch gemacht. Erbt eine Klasse von einer anderen, so wird sie zu einer sogenannten Subklasse, während die andere eine Superklasse (auch Eltern- oder Basisklasse) darstellt. Sinn und Zweck der Vererbung ist die Wiederverwendbarkeit von Code und somit die Vermeidung von Redundanz. Es ermöglicht Code für Klassen die gewisse Merkmale teilen nur einmal schreiben zu müssen und diese dann davon ableiten zu lassen. Die Polymorphie liegt vor, wenn zwei Klassen denselben Methodennamen verwenden, aber die Implementierung der Methoden sich unterscheidet. Listing 18 verdeutlicht diesen Sachverhalt.

Listing 19: Vererbung & Polymorphismus

```
1  fun main(args: Array<String>) {
2
3   val dog = Dog()
4   val cat = Cat()
5
6   // gleiche Methode, aber unterschiedliche Implementierung
7   (dog as Animal).makeSound() // Wuff
8   (cat as Animal).makeSound() // Miau
9  }
10
11 abstract class Animal {
12
13  abstract fun makeSound()
14 }
15
16 class Dog : Animal(){
17
```

```
18   override fun makeSound(){
19     println("Wuff")
20   }
21 }
22
23 class Cat : Animal(){
24
25   override fun makeSound(){
26     println("Miau")
27   }
28 }
```

Für die Controller ähnliche Komponente bedarf es der Festlegung des vom Entwickler gewählten Intent Datentypen. In diesem Fall kann wieder auf den generischen Aspekt von Kotlin zurückgegriffen werden. Dieser Typ wird ebenfalls als Parameter Typ für die Intent-Funktion genutzt. Ein erster Entwurf kann wie folgt aussehen:

Listing 20: Erster Controller Entwurf

```
1  abstract class Controller<I> {
2
3    abstract fun intents(intent: Intent)
4  }
5
6  sealed class SomeIntent {
7    // ...
8  }
9
10 class SomeCrontroller: Controller<SomeIntent> {
11
12   override fun intents(intent: SomeIntent){
13     // ....
14   }
15 }
```

Im nächsten Schritt müssen die Intents einer Aktion zugeordnet bzw. in eine übersetzt werden. Dadurch, dass die Superklasse als Parameter verwendet, muss zu erst ermittelt werden, um welchen Subklasse von Intent es sich handelt. Für die Auswertung eines Intents kann auf eine weiteres, funktionales Konzept zurückgegriffen werden: Der Musterabgleich (Pattern Matching zu Englisch). Hierbei handelt es sich um ein Verfahren das prüft, inwieweit ein vorgegebenes Muster mit anderen Formen"(hier Klassen) übereinstimmt.

Auch in diesem Fall schafft Kotlin Abhilfe. Es stellt eine Expression bereit, welche eine elegante Handhabung für genau diesen Fall bietet. Sie ist dem ”switch”Statement aus Java sehr ähnlich, offenbart aber mehr Möglichkeiten und ist dementsprechend mächtiger. Ein gewöhnliches switch Statement ist im Grunde nur eine Aussage, die eine Reihe von einfachen if/else ersetzen kann, welche grundlegende Prüfungen durchführen. Folgendes Vorteile kann mit der sogenannten ”when”Expression erzielt werde:

1. Es gibt immer einen Wert zurück (minimal ”Unit”), ohne return”angeben zu müssen
2. Es ist kein Argument nötig
3. Es kann beliebige Bedingungsausdrücke haben
4. Es hat ein allgemein besseres und sicheres Design (mögliche Überprüfung während der Kompilierung)
5. Es unterstützt ”Smart Cast”oder ”Auto Casting”

Das nachfolgende Listing 20 veranschaulicht die Anwendung dieser Expression mit mehreren Bedingungsausdrücken:

Listing 21: When Expression

```
// Mit Argument
// Gibt einen String zurück
val result = when(number) {
 0 -> "Invalid number"
 1, 2 -> "Number too low"
 3 -> "Number correct"
 in 4..10 -> "Number too high, but acceptable"
 !in 100..Int.MAX_VALUE -> "Number too high, but solvable"
 else -> "Number too high"
}

// Ohne Argument
// Gibt ebenfalls einen String zurück
val res = when {
 x in 1..10 -> "cheap"
 s.contains("hello") -> "it's a welcome!"
 else -> ""
}
```

Die Punkte 4. und 5. sind dabei von besonderem Interesse. Durch die Verwendung einer ”sealed”Klasse für die Implementierung der Intents ist es dem Compiler möglich festzustellen, inwieweit alle Subklassen innerhalb der ”when”Expression abgedeckt sind. Fehlt

eine, so wirft der Compiler einen Fehler und die Anwendung kann nicht ausgeführt werden. Des weiteren verliert der ”else”seinen Nutzen, weshalb auf ihn verzichtet werden kann.
Den nächsten Vorteil der sich ergibt, ist die Fähigkeit von Kotlin nach dem Bedingungsausdruck die Basisklasse automatisch in die korrekte Subklasse umzuwandeln. Dies wird als sogenanntes ”Smart Casting”bezeichnet und erspart dem Entwickler ein manuelles umwandeln in die richtige Klasse. Ein Vergleich zwischen Java und Kotin gibt Listing 21 an.

Listing 22: ”whenmit ”smart cast”

```
1
2  // JAVA
3  val intent = Intent.GoBack
4
5  if(intent instanceof Intent.DisplayItemDetails){
6   ((Intent.DisplayItemDetails) intent).item // expliziter Cast
7  } else if(...) {
8    ...
9  }
10
11 // KOTLIN
12
13 // "is" ist Notwendig, sobald die Klasse Konstruktor
14 // Argumente hat.
15 // Bei "object" is keines von Nöten, da die Klasse
16 // nicht erzeugt wird.
17 when(val intent = Intent.GoBack){
18   GoBack -> println("go back")
19   is DisplayItemDetails ->
20     println(intent.item) // <-- Smart Cast
21   // keine "else" Zweig erforderlich, da alle
22   // Subklassen abgedeckt sind
23 }
```

Mit diesem Ansatz lässt sich die Intent-Funktion umsetzen.

5.5 Aktion

Die Aktion wird aus dem Intent gebildet. Sie gibt an, welcher Teil der Business Logik aufgerufen wird. Sie soll mit der gleichen Struktur wie der Intent aufgebaut sein.

5.6 Reducer

Im Reducer findet der Aufruf der Business Logik statt. Es handelt sich dabei um eine Funktion, welche die aus dem Intent hergeleitet Aktion und das derzeitige Model preisgibt.

Abbildungsverzeichnis

Book References

[2] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 13. Jan. 2013, S. 19–22.

[3] Donald Wolfe. *3-Tier Architecture in ASP.NET with C sharp tutorial*. SitePros2000.com, 13. Jan. 2013.

[4] Adam Boduch. *Flux Architecture*. Packt Publishing, 30. Jan. 2017, S. 27, 198, 312.

[5] Ilya Gelman und Boris Dinkevich. *The Complete Redux Book*. Leanpub, 30. Jan. 2017, S. 6–7.

[6] Adam Boduch. *Flux Architecture*. Packt Publishing, 30. Jan. 2017.

[8] Robin Wieruch. *Taming the State in React: Your journey to master Redux and MobX*. CreateSpace Independent Publishing Platform, 5. Juni 2018, S. 3.

[9] Ajdin Imsirovic. *Elm Web Development: An introductory guide to building functional web apps using Elm*. Packt, März 2018.

[10] Ajdin Imsirovic. *Elm Web Development: An introductory guide to building functional web apps using Elm*. Packt, März 2018, S. 50–65.

Artikel Referenzen

[1] Thomas A Wadlow. „The Xerox Alto Computer“. In: (Sep. 1982). URL: https://tech-insider.org/personal-computers/research/acrobat/8109-e.pdf.

[11] Jagatheesan Kunasaikaran und Azlan Iqbal. „A Brief Overview of Functional Programming Languages“. In: *electronic Journal of Computer Science and Information Technology (eJCSI)T Vol. 6, No 1* (2016). URL: ejcsit.uniten.edu.my/index.php/ejcsit/article/view/97/39.

Online References

[7] Facebook Developers. *Hacker Way: Rethinking Web App Development at Facebook*. 4. Mai 2014. URL: https://www.youtube.com/watch?v=nYkdrAPrdcw (besucht am 08. 07. 2019).

[12] Andre Staltz. *What if the user was a function? by Andre Staltz at JSConf Budapest 2015*. Youtube. 4. Juni 2015. URL: https://www.youtube.com/watch?v=1zj7M1LnJV4 (besucht am 14. 07. 2019).

[13] Hannes Dorfmann. *Model-View-Intent on Android*. 4. März 2016. URL: http://hannesdorfmann.com/android/model-view-intent (besucht am 23. 05. 2019).