Hochschule Bremen

Bachelorarbeit

Entwicklung eines Model-View-Intent Framework für die Plattform Android

Medieninformatik B.Sc.

Roman Quistler

30. August 2019

Zusammenfassung

Anhand dieser Arbeit soll ein MVI Framework für die Plattform Android konzeptioniert werden. Dafür findet eine prototypische Realisierung statt, die feste Vorgaben für die Anwendung vom MVI macht und einen Entwickler bei der Umsetzung unterstützt.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift

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 bspw.. die Aktualisierung der zugrundeliegenden Datenbank miteinbezogen werden.

Da diese Problematik nicht erst seit kurzem sondern seit vielen Jahren besteht, wurden hierfür bereits unterschiedliche Ansätze entwickelt. Diese spiegeln sich in sogenannten Architekturmuster wieder und bieten ein Mittel, den „Presentation Layer“ sinnvoll zu organisieren.

Zu diesen Architekturmuster hat sich im Jahre 2015 ein neues hinzugesellt: „Model-View-Intent (MVI)“. Es besitzt bereits bekannte Herangehensweisen, setzt jedoch auch auf neue Ideen. Zu Beginn kam es ausschließlich in der Entwicklung von Webanwendungen zum Einsatz, bis es mit etwas Verzögerung auch in der (nativen) Android Entwicklung Einzug hielt. Wie es meist der Fall ist wenn neue Wege beschritten werden, ergibt sich viel Ungeklärtes. Dieses wächst wenn zusätzlich eine Wechsel der Plattform vorgenommenen, welche trotz Gemeinsamkeiten erhebliche Unterschiede und aufweist.

Ist ein Entwickler an einem Einsatz von MVI interessiert, so ergeben sie für ihn gewisse, wenn auch übliche Hindernisse: Wie lassen sie die jeweiligen Komponenten umsetzten? Wie gut lassen sich Implementierungen für eine Plattform auf eine andere, seine Übertragen? Inwieweit müssen Eigenheit beachtet werden?

1.2 Ziel der Arbeit

Mit der Arbeit wird das Ziel verfolgt, das Architekturmuster MVI zu untersuchen, besser zu verstehen und im Kontext Android ein Framework zu schaffen.

Es sollen Eigenheiten der Plattform, sowie allgemeine Probleme ausgemacht werden die für den Einsatz von MVI in Android relevant sind. Dazu müssen bereits bestehende Lösung evaluiert und vorhandene Problematiken aufgezeigt werden. Auf Basis der gewonnenen Erkenntnisse soll daraufhin ein kleines und dogmatisches Framework entwickelt werden. Die Absicht ist dem Anwender alle nötigen Komponenten zu Realisierung von MVI zur Verfügung zu stellen und eine klare Struktur vorzugeben. Dabei soll der Aufwand seitens des Nutzers möglichst gering gehalten werden.

1.3 Aufbau der Arbeit

Im ersten Schritt werden die nötigen Grundlagen für ein besserer Verständnis von MVI geklärt. Dazu gehören spezielle Paradigmen der Programmierung als auch vorausgegangene Konzepte und Bibliotheken.

Ist dies Vollbracht so wird im nächsten Schritt MVI mit all seinen Komponenten genau beschrieben. Es wird auch versucht, die Gründe für die Entstehung von MVI zu finden und zu erläutern.

Daraufhin werden die Funktionalen und Nichtfunktionalen Anforderung für das zu entwickelnde Framework aufgelistet und näher ausgeführt. Hierbei soll Erkennbar ein, worauf der Fokus liegt und was als Optional eingestuft wird.

Im weiteren Fortgang wird mit diesen Anforderung und den zuvor erworbenen Kenntnissen das Framework und seine individuellen Komponenten konzipiert. Jeder dieser wird ausführlich beleuchtet und ihre Funktion dargelegt. Auch die Zusammenhänge innerhalb des Frameworks werden aufgegriffen und erörtert.

Anschließend erfolgt die Implementation des Framework in Form eines Prototypen. Zuvor werden jedoch Grundlegende Entscheidungen und ihre Auswirkungen erläutert.

Als vorletzter Schritt werden die jeweiligen Anforderungen auf Basis des Prototypen ausgewertet. Es wird geschaut zu welchem Grad diese Erfüllt wurden und falls nicht, welche Gründe dies hat und wie es umgesetzt werden könnte. Außerdem sollen eventuelle Verbesserungen diskutiert werden.

Zum Schluss wird die Arbeit und ihr Ergebnis zusammengefasst und ein Ausblick gegeben.

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 und 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 eine „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.

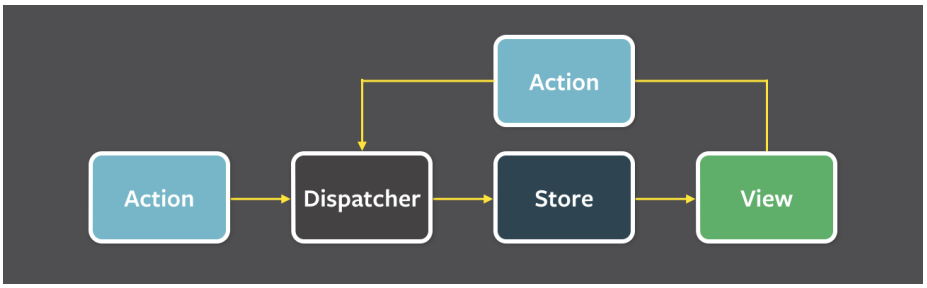

Abbildung 1: Datenfluss in der Flux Architektur

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: Dies repräsentiert den Zustand der Applikation als eine simple Datenstruktur.
View: Sie 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.

Update: Hierbei wird, ähnlich wie im „Reducer“, das Model manipuliert und dadurch der Zustand der Anwendung aktualisiert.

Es ist zu erkennen, dass diese Architektur den vorherigen sehr ähnlich ist. Sie genießt jedoch den Vorteil ein wesentlicher Bestandteil der Sprache zu sein. Damit ist es unter anderem möglich, die Sprache auf die Architektur anzupassen, statt andersherum.

2.2 Funktionale Programmierung

Im Verlaufe der Kapitel wurden bereits Begriffe wie eine reineFunktion („pure functions“) oder die Unveränderlichkeit 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, objektorientierten Programmierung wie sie in Java oder auch C# anzutreffen ist, machen Klassen, die Mutation derer und globale Variablen den Hauptbestandteil des Quellcodes aus. Dabei ist zu beachten, dass die funktionale Programmierung die Objektorientierte nicht zwingenden ausschließt, sondern lediglich in eingeschränkter Form nutzt. In der funktionalen Programmierung geht es, wie der Name bereits vermuten lässt, hauptsächliche um das Arbeiten mit Funktionen.

Dabei muss man verstehen, das bei einer imperativen Entwicklung Schritt für Schritt programmiert wird, wie etwas getan werden soll. Es handelt sich dabei oft um eine Serie von Mutationen in Kombination mit Abfragen von gewissen Konditionen. In der funktionellen hingegen wird nur zum Ausdruck gebracht, das etwas vollführt werden soll, aber nicht wie es am Ende vollbracht wird.

„Funktionale“ Funktionen sind nicht identisch mit den „unreinen“ Funktionen, die beispiel- oder typischerweise in Java genutzt werden - die, die einen Wert zurückgibt (oder nicht). Stattdessen ist eine funktionale Programmierfunktion wie eine mathematische Funktion, die eine Ausgabe erzeugt, die ausschließlich von ihren Argumenten abhängig ist. Jedes Mal, wenn diese mit den gleichen Argumenten aufgerufen wird, erhält man immer das gleiche Ergebnis. Erreicht wir dies durch Funktionen, die Frei von Seiteneffekten sind. Das bedeutet, dass keine globale Variable mutiert oder bspw. auf die Konsole ausgegeben wird. Ebenfalls sind Operation die auf die Peripherie des Gerätes zugreifen (I/O) theoretisch nicht erlaubt. Somit ist ein Seiteneffekt ein extern beobachtbarer Effekt, den eine Funktion zusätzlich zur Rückgabe eines Wertes ausführt. Im Prinzip sorgen sie für inkonsistente Abläufe bzw. Resultate innerhalb eines Programms. Natürlich ist die Entwicklung einer Anwendung ohne Seiteneffekte unmöglich, daher geht es vielmehr darum, sie zu minimieren und gesondert zu behandeln. Sie sollten meist am Ende einer Berechnung erfolgen und nicht im Laufe dieser.

Für die Einhaltung dieser Prinzipien stellt eine funktionale Sprache meist unterschiedliche Werkzeuge zur Verfügung. Dazu gehören Funktionen höherer Ordnung, bei der eine Variable eine Funktionen verkörpern kann. Die tief verankerte Fähigkeit unveränderliche Datenstrukturen zu erstellen. Das arbeiten mit Klassenhierarchien und algebraischen Datentypen, sowie dem oft dazugehörigen Musterabgleich. Oder die Bereitstellung von sogenannten „Monads“, welche unter anderem dafür dienen Seiteneffekte zu isolieren und erkennbar zu machen.

Ein großer Vorteil der funktionalen Programmierung ist, das vieles ein deterministisches Verhalten aufweist. Dies macht es einfacher einen Überblick zu bekommen und das Programm zu verstehen. Zugleich macht die Komposition von Funktionen das Programm modularer. Ein weiterer nicht zu unterschätzender Vorteil entspringt der Unveränderlichkeit, durch welches die Anwendung Threadsicherheit erlangt.

Insgesamt kann gesagt werden, dass das Anwenden von funktionalen Konzepten auch in Sprachen wie Java angestrebt werden sollte. Selbst wenn dies nicht in dem Umfang stattfinden kann, wie es in „echten“ funktionalen Sprachen möglich ist.
Zuletzt sollte noch erwähnt werden, dass am Ende immer imperativer Code steht. Funktionale Programmierung stellt eine Abstraktion dar und ist intern imperativ.

2.3 Reaktive Programmierung

Heutige Anwendungen müssen viele Ereignisse in scheinbar willkürlicher Reihenfolge verarbeiten: Ein Klick eines Nutzer, eine Fehlermeldung, Speichern im Dateisystem, Kommunikation mit dem Netzwerk, aktualisieren der Benutzeroberfläche usw. Auf jedes dieser Ereignisse muss das Programm reagieren, und dabei immer Ansprechbar bleiben. Hierfür ist es Notwendig den Code asynchron zu schreiben, d.h. ohne das andere Prozeduren blockiert werden.

Ein bekannte Methode stellt hierbei die Verwendungen von sogenannten Rückruffunktion („Callbacks“) dar, welches ein gern genutztes Mittel in der imperativen Welt ist. Leider führt diese Variante meist zu einer „Callback Hell“ und tief verschachtelten, unübersichtlichen Code. Dazu existieren vielfältige Verbesserungen, die alle eins Gemeinsam haben: sie reagieren (auf etwas).

Hierzu besteht ein weiteres Paradigma: die reaktive Programmierung. Sie setzt auf Konzepte der funktionalen Programmierung, bevorzugt einen deklarativen Ansatz und ist damit eine weitere Abstraktion. Ihr anliegen ist es, asynchronen Code basierend auf unterschiedlichen, zeitlich abhängigen Ereignissen einfach zu handhaben.
Ein Beispiel ist das Tippen auf einen Touchscreen; etwas, dass zu beliebigen Zeitpunkten passieren kann. Anstatt in einer Schleife zu fragen „Hat der Nutzer den Bildschirm berührt?“ hinterlegt man beim System etwas das auf dieses Ereignis horcht. Registriert das System eine solche Aktion leitet es dies Information an alle „Zuhörer“ weiter. Über

diese Vorgehensweise können die unterschiedlichsten Datenquellen angesprochen werden. In der reaktiven Programmierung ist es möglich mehrere solcher Ereignisse zu Kombinieren und mit ihnen zu arbeiten.

Zum Einsatz kommt in der reaktiven Programmierung das Beobachter-Muster (Englisch Observer Pattern). In diesem wird eine Änderung an einem Objekt an alle die von ihm Abhängig sind propagiert. Dabei unterscheidet man zwischen einem „Observable“ und dem „Observer“. Ersterer erzeugt ein oder mehrere Ereignisse und bietet Operatoren um mit ihnen zu arbeiten. Daraus ergibt sich ein Fluss an Daten, der von jemanden aufgenommen und beobachtet werden möchte - dafür ist der „Observer“ zuständig. Dieser schließt eine Abonnement auf dem „Observable“ ab und erhält die gewünschten Daten. In Pseudocode kann dies wie folgt aussehen:

Listing 1: Pseudocode reaktive Programmierung

```
1 Observable.produce("Hello")
2   .delay(1000)
3   .map { text -> text + " World" }
4   .subscribe(SomeObserver())
```

Am Beginn von Listing 1 wird eine Wert erzeugt, als „Observable“ transferiert und um 1 Sekunde verzögert. Genau wie bei der funktionalen Programmierung ist dabei die Implementation von „delay“ zweitrangig. Es geht nur darum auszudrücken was gemacht werden soll und nicht wie. Dies hat zur Folge, dass der Code ausdrucksstark wird. Als nächstes wird mit einer anonymen Funktionen höherer Ordnung der Wert manipuliert. Zum Schluss hinterlegt sich der „Observer“ als Abonnent und erwartet das Resultat. Abgesehen von der „subscribe“ Funktion gibt jede andere ein neues „Observable“ zurück. Das liegt daran, dass das „Observable“ als unveränderliche Datenstruktur realisiert wird.

Reaktive Programmierung macht es einfach, auf mehrere (gleichzeitige) Ereignisse zu reagieren und asynchron als auch parallel mit ihnen in deklarativer Form zu arbeiten.

3 Model-View-Intent

In diesem Kapitel wird MVI unter die Lupe genommen und genauer beschrieben.

3.1 Vorausgegangene Architekturmuster

Hier werden in kurzer Ausführung die bekanntesten Architekturmuster neben Model-View-Intent beschrieben.

3.1.1 Model-View-Controller

Mode-View-Controller oder auch MVC ist eines der ältesten Architekturmuster und wurde im Jahre 1979 von Trygve Reenskaug veröffentlicht. [12, 13] Es besteht aus drei Komponenten mit folgenden Funktionen:

- **Model**: Es beschreibt den Zustand der Anwendung durch das Festhalten von Daten und aktualisiert informiert die „View“ über Veränderungen.
- **View**: Sie ist für die Darstellung des UI zuständig und aktualisiert sich bei Änderungen im Model.
- **Controller**: Er kontrolliert die beiden anderen Komponenten und enthält möglicherweise weitere (Business) Logik. Dafür nimmt Ereignisse aus der „View“ bzw. vom Nutzer entgegen und manipuliert daraufhin das Model oder die „View“ direkt.

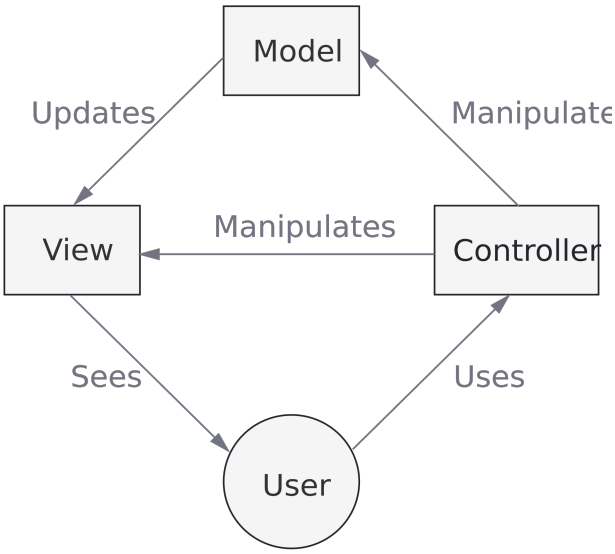

Abbildung 2: Model-View-Controller

https://cycle.js.org/model-view-intent.html

3.1.2 Mode-View-Presenter

Das nächste im Bunde ist Model-View-Presenter (MVP) aus dem Jahre 1996. [14, 15] Es kann als die erste Iteration von MVC betrachtet werden, birgt dabei allerdings ein paar wesentliche Unterschiede. Genau wie sei Vorreiter wird es aus drei Komponenten zusammengesetzt:

- **Model**: Es beinhaltet die Business Objekte und Logik die für die Ansicht von Relevanz sind. Sie kann zuständig sein für Zugriffe auf die Datenbank, das Dateisystem und Netzwerk (oft Rest API). Es hat keine direkten Zugriff auf den „Presenter“ oder due „View“.
- **View**: Sie ist ausschließlich für die Darstellung des UI und für die Interaktion mit dem Nutzer zuständig. Sie hat keine direkten Zugriff auf den „Presenter“ oder das „Model“.
- **Presenter**: Er beinhaltet die Business Logik der Anwendung und steht mit der „View“ und dem „Model“ im Austausch. Als Bindeglied steuert er die Abläufe zwischen den Beiden.

Der gravierende Unterschied zu MVC besteht darin, dass das „Model“ nichts von der „View“ weiß und andersrum genauso. Der „Presenter“ kontrolliert die „View“, welche jegliche Ereignisse an den „Presenter“ weiterleitet. Erreicht wird diese durch die Verwendung von Interfaces bzw. Verträgen zwischen den Schichten. Damit erhält keine der Komponenten die konkrete Implementation dieser, sonder eine Abstraktion. Dies führt zu einer verbesserten Modularität, da die Komponenten zu jeder Zeit ausgetauscht werden können, solange der Vertrag „erfüllt“ wird.

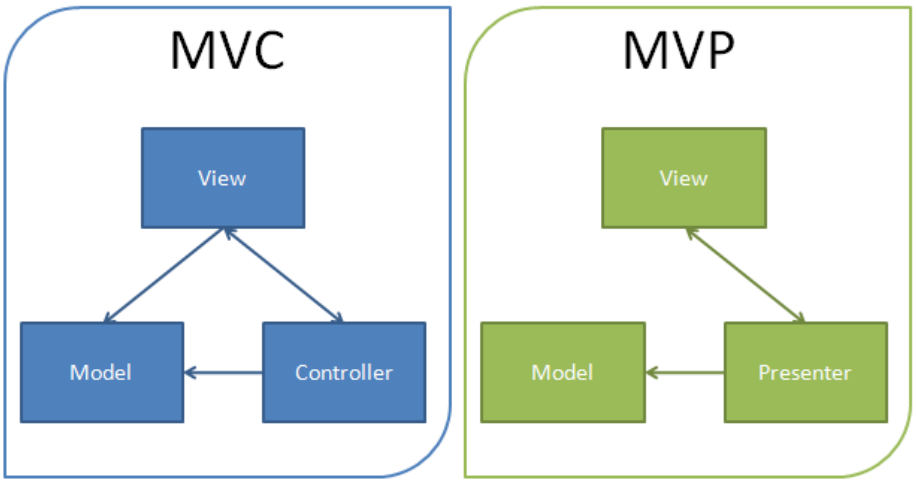

Abbildung 3: MVC vs. MVP

Source: https://i.imgur.com/xbeB5.png

3.2 Historie & Grundlegendes

Model-View-Intent (MVI) ist eine weiteres Entwurfsmuster, welches der Feder von André (Medeiros) Staltz entstammt. Er stellte dieses auf einer Javascript Konferenz im Jahre 2015 vor [16]. Es gehört damit zu den jüngsten seiner Art. Seine ursprüngliche Anwendung fand es in dem ebenfalls von André Staltz geschaffenen Framework „Cycle-Js“, hat seit der Artikelreihe von Hannes Dorfmann in 2016 [17] 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 bspw.. 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).

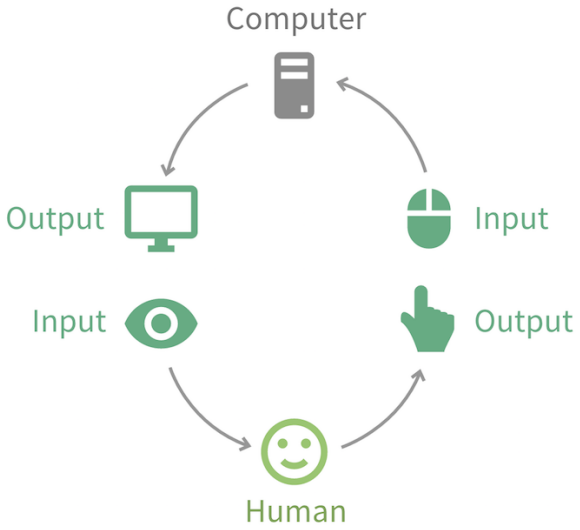

Abbildung 4: Nutzer und Computer als Input und Output

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.

3.3 Model, View & Intent

Fast noch wichtiger als der reaktive Ansatz ist zu verstehen, wie der Kreislauf in Figur 4 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 2: funktion

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

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

Listing 3: 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 4: 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 -> {
```

```
10      val newText = event.text.trim() // <-- business logic
11      model.copy(text = newText) // <-- immtuable data structure
12    }
13  }
14 }
15
16 fun view(model: Observable<Model>){
17  // we subscribe to the model to listen for changes
18  model.subscribe { model ->
19    textView.text = model.text
20  }
21 }
22
23 fun main(args : Array<String>) {
24
25  // this listens to arbitary text changes
26  val textChanges: Observable<String> = textView.changes()
27
28  view(model(intent(textChanges)))
29 }
```

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

3.4 Reducer

Schaut man sich die Model-Funktion in Beispiel 4 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“. Er ist für genau den oben aufgeführten Prozess zuständig und der Platz, an dem das Model verändert wird.

3.5 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.

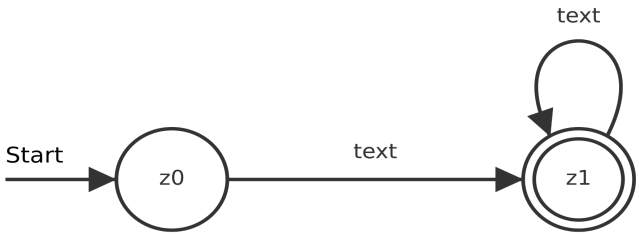

Abbildung 5: Endlicher Automat

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 5 zu präsentieren.

4 Anforderungsanalyse

In diesem Kapitel werden beide Seiten von Anforderungen beschrieben: Funktional und Nichtfunktional.

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 Bachelorarbeit wird sich im diesen Teil ausschließlich auf „Muss“-Anforderungen 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 & Funktional
Das Framework soll auf verstärkt reaktiven und funktionalen Konzepten aufbauen.

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.

ID	Anforderung	Typ
01	Identifizieren von „Intent“, „Action“, „State“ und „Result“	fa
02	Trennung von Business und Ansicht Logik	fa
03	Überprüfung des Zustands als unveränderliche Datenstruktur	fa
04	Bereitstellung eines „Reducers“	fa
05	Handhabung von Seiteneffekten	fa
06	Verwalten des Zustands	fa
07	Funktion für die Aktualisierung der Ansicht	fa
08	Speichern des Zustands	fa
09	Wiederherstellung des Zustands	fa
10	Asynchrone Ausführung	fa
11	Dogmatisches Framework	na
12	Unidirektionaler Datenfluss	nfa
13	Kotlin spezifische Implementierung	nfa
14	Reaktiv & Funktional	nfa

4.4 Begutachtung bestehender MVI-Frameworks/Bibliotheken

In diesem Kapitel wird ein Blick auf bereits bestehende MVI-Frameworks/Bibliotheken geworfen.

4.4.1 MVICore

Hierbei handelt es sich um ein auf MVI ausgerichtet Framework dessen Entwicklung seit 2018 auf Github stattfindet. [18] Es bietet eine vollständige Abdeckung der von MVI verlangten Komponenten und stellt darüber hinaus weitere Eigenschaften wie „Time Travel Debuggig“ und „Middlewares“ zur Verfügung.

Ähnlich wie im Konzept existiert hier eine Klasse, die den Zustand verwaltet: der 'Store'. Es besteht hierfür aber keine Überprüfung auf Unveränderlichkeit.

Eine erweitere Form des 'Store' ist das 'Feature' Interface. Es sind verschiedene „Feature“

Implementationen für unterschiedliche Anforderungen vorhanden. Zu gibt es bspw. das „ActorReducerFeature“ welches für asynchrone Logik gedacht ist und das arbeiten mit dem Typ „Observable“ erlaubt. Darüber erfolgt die Realisierung des Unidirektionalen Datenflusses.

Ein erstes großes Plus ist das Vorhandensein der automatischen Beendigung von Abonnements die im Zuge der Nutzung von 'Observable' entstehen. Die bedeutet, der Entwickler muss keine Sorge bezüglich eines Speicherlecks haben. Möglich ist dies durch die Bindung des 'Features' an den Lebenszyklus der „View“ Komponente. Hierfür existiert eine „Binder“ Klasse.

Ein weiter und wichtiger Pluspunkt ist die automatische Speicher- und Wiederherstellung des Zustands. Damit ist garantiert, dass nach Drehung des Geräts sich der Nutzer in dem selben Zustand wie zuvor befindet.

Des weiteren wird durch „Middleware“ eine Möglichkeit geboten, „Cross-Cutting Concerns“ wie z.B. das Loggen als generische Plugins zu implementieren und einem oder mehreren 'Feature' Klassen zuzuweisen.

Insgesamt übersteigt dieses Framework den Funktionsumfang des hier niedergeschrieben Konzepts und erfüllt soweit alle Anforderungen.

5 Design & Konzept

In diesem Kapitel wird sich der Erstellung eines Konzepts für das MVI Framework gewidmet.

5.1 Übersicht der Komponenten im Klassendiagramm

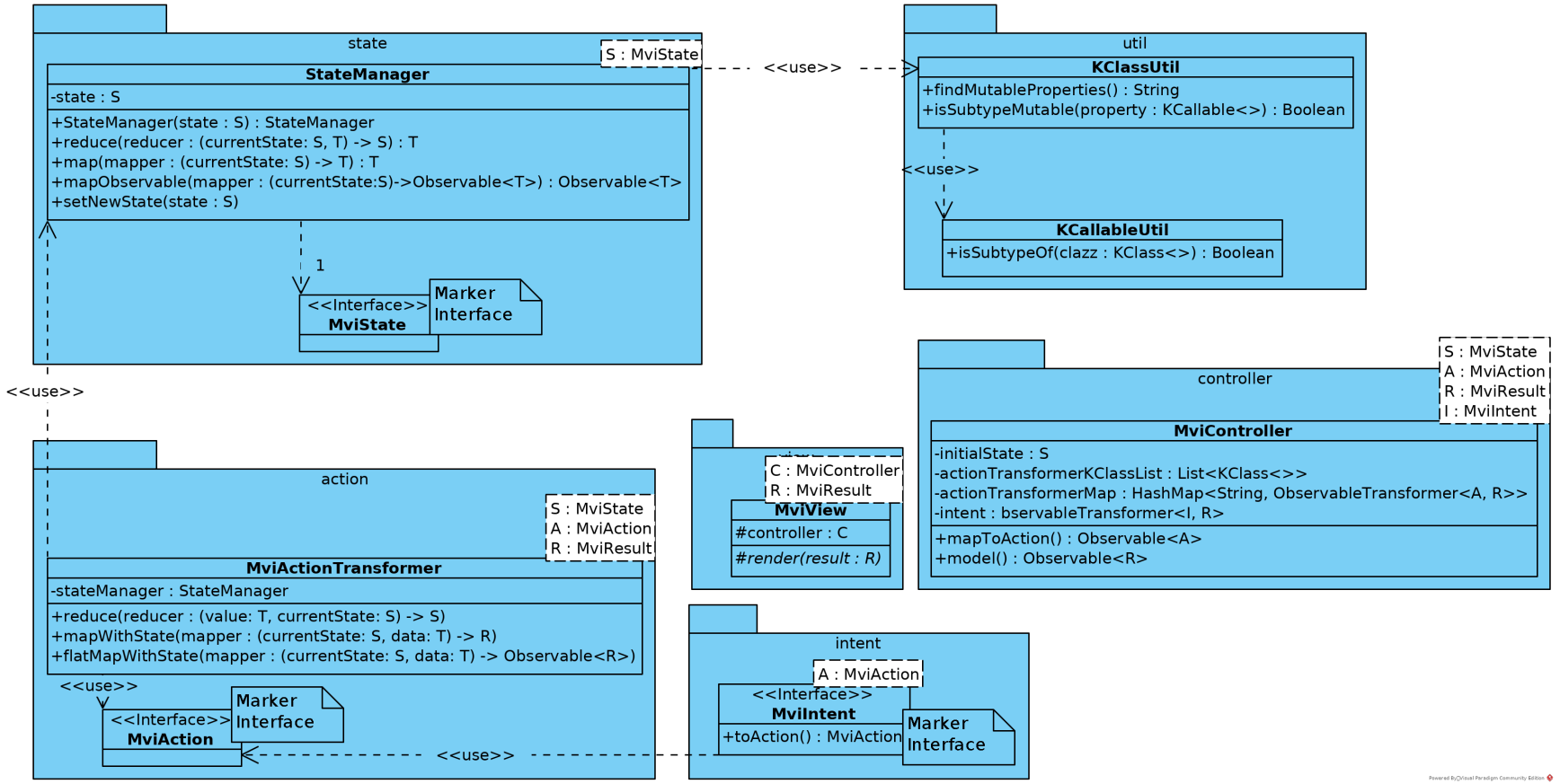

Abbildung 6: Komponenten im Klassendiagramm

5.2 Zustand (State) und seine Verwaltung

In den Ausführungen von MVI wird der Zustand als eine Anhäufung von Werten betrachtet. Es bildet den Kern von MVI und muss im Normalfall vom Entwickler selbst verwaltet werden. Unter anderem muss garantiert sein, dass ein Zugriff und eine Modifikation des Zustands nur an einer Stelle erfolgen kann. Im Rahmen des Framworks wird eine Komponente genutzt, die diese Aufgaben für den Entwickler übernimmt und den Zustand „managed“: Der „StateManager“.

Dieser erwartet den initialen Zustand, der vom Entwickler an das Framework übergeben wird. Der Zustand wird dabei als generischer Parameter definiert und muss dem Interface „MviState“ entsprechen. Die Variable „state“ selbst ist eine der wenigen, die eine Mutation zulassen muss, wie später deutlich wird.
Nachdem der Zustand überreicht wurde, wird geprüft, inwieweit dieser der Anforderung der Unveränderlichkeit entspricht. Dieser Vorgang wird auf zwei Klassen aufgeteilt: „KClassUtil“ und „KCallableUtil“. Erstere durchläuft alle Attribute des Zustands und verifiziert, dass keinem dieser ein neuer Wert zugewiesen werden kann. Die letztere schaut, ob auch der Subtyp, z.B eine Liste, nicht direkt modifiziert werden kann. Sollte diese nicht gegeben sein, so wird eine Fehlermeldung ausgegeben und der Prozess gestoppt.

Ist dies jedoch erfolgreich, so wartet der „StateManager“ mit Funktionalität auf, welche es ermöglicht einen neuen Zustand zu hinterlegen, sowie mit ihm sicher zu arbeiten. Ein neue Setzung des Wertes wird dabei durch die Kombinationen der Funktionen „reduce“ und „setNewState“ erreicht.

Erstere erwartet eine Funktion als Parameter, „reducer“, welche wiederum den derzeitigen Zustand übergeben bekommt. Diese wird aufgerufen und produziert einen neuen Zustand, der durch „setNewState“ als aktueller Zustand gesetzt wird. Dies geschieht allerdings nur unter der Prämisse, dass sich mindestens ein Wert innerhalb der Datenstruktur verändert hat.
Als Zusatz stellt „setNewState“ sicher, dass es auch bei gleichzeitigen Zugriffen von mehreren „Threads“ zu keiner sogenannten unbeabsichtigten Wettlaufsituation (Race Condition) und damit inkonsistenten Daten kommt.
An dieser Stelle wird erkennbar, weshalb die „state“ Variable zwingend veränderlich sein muss. Wäre diese nicht der Fall, so könnte ihr kein neuer Zustand zugewiesen werden.

Die Methode „map“ nimmt genau wie „reduce“ eine Funktionen entgegen, welche den aktuellen Zustand erhält. In dieser kann der Entwickler auf die Attribute des Zustands zurückgreifen, jedoch keine neuen bestimmen.

Diese Komponente ist nicht direkt für den Nutzer zugänglich und wird ausschließlich intern im Framework genutzt. Es verhindert, dass der Zustand an einer beliebigen und nicht vorgesehenen Stelle verändert werden kann.

5.3 Intent, Action und Result

Die im Titel genannten Komponenten dienen der Übermittlung und Beschreibung von Informationen innerhalb des „Kreislaufs“ vom MVI. Diese müssen vom Entwickler selbst gestellt werden, erhalten dabei jedoch Vorgaben vom Framework.

So muss zu jedem Intent eine Action existieren. Hierfür wartet das Framework mit dem Interface „MviIntent“ auf, das genanntes erzwingt und die Erfüllung dieser Anforderung sicherstellt. Realisiert wird dies durch die Funktion „toAction“, die der Entwickler später implementieren muss und eine „Action“ zurück gibt.

Ähnlich wie auf einen Intent eine Action erfolgt, zieht eine Action ein Result nach sich. Auch das gibt das Framework durch ein Interface namens „MviAction“ vor und muss vom Entwickler angewandt werden. Das Result findet seine Anwendung später in der View und wird ebenfalls mit einem Interface versehen.

Sämtliche der hier aufgeführten Strukturen sollten mit ihrem Namen die vorgesehene Intention signalisieren. Zusätzlich müssen sie in der Lage sein weitere Nutzdaten (Payload) aufzunehmen, die mit ihnen in Zusammenhang stehen und für den weiteren Verlauft essentiell sind.

5.4 Reaktiver unidirektionaler Datenfluss

Für die Konzeptionierung der nächsten Komponenten gilt vorher zu klären, wie der geforderte reaktive unidirektionale Datenfluss in das Framework integriert werden soll. Dabei müssen Daten synchron oder asynchron verarbeitet und die Option zur Nebenläufigkeit (Concurrency) geboten werden. Zu diesem Zweck bedarf es einem Tool, welches das bereits aufgeführte „Observer“ und „Iterator“ Pattern nutzt. In diesem Zusammenhang taucht oft das Objekt „Observable“ auf, das genau diese Anforderungen erfüllt.

5.5 Transformer und die Business-Logik

Ein elementaren Bestandsteil einer Anwendung macht die Businesslogik aus. Sie ist im Falle von MVI allein verantwortlich für das Abändern des Zustands und sollte strikt von anderen Komponenten getrennt sein. Hierfür stellt das Framework den „ActionTransformer“ zu Verfügung.

Wie der Name vermuten lässt, ist die „Action“ unter anderem ausschlaggebend für die Nutzung der Komponente. Jeder „Action“ wird dabei ein „MviActionTransformer“ zuteil, welche in der ersten der drei generischen Parameter, dem „A“, eingesetzt wird. Dieser Parameter setzt voraus, dass das eingefügte Objekt vom Interface „MviAction“ nutzen macht. Der zweite generische Parameter „R“ sieht ein Objekt vor, dass dem Interface „MviResutlt“ entstammt. Hier wird das zugehörige Result zur angegebenen Action eingefügt. Zuletzt muss der „MviActionTransformer“ wissen, mit welchen Zustand er zu tun haben wird. Dies wird ihm durch den letzten generischen Parameter „S“ mitgeteilt,

der als Zuordnung das „MviState“ Interface nutzt.

Der „MviActionTransformer“ ermöglicht die Interaktion mit dem Zustand durch eine Variation an Funktionen. Dafür benötigt er einen „StateManager“ welcher ihm bei seiner Erstellung übergeben werden muss. Die Funktionalität kann dabei - ähnlich wie beim „StateManger“ - in zwei Kategorien unterteilt werden:

1. Die, in der ein neuer Zustand erzeugt wird und
2. die, in der ausschließlich auf die Daten zugegriffen wird

Zu ersten Kategorie gehört lediglich die „reduce“ Methode, welche auf die gleichnamige Methode im „StateManager“ zugreift. Auch sie bekommt eine Funktion als Parameter, jedoch mit dem Unterschied, dass diese zusätzlich den derzeitigen Wert in Bearbeitung enthält. Dies kann beispielsweise ein Item sein, dass von einem Intent an die Action weitergegeben wurde. Auf Basis dieser zwei Werte kann der Entwickler einen neuen Zustand bestimmen, welcher intern über den „StateManager“ gesetzt wird. Diese Operation findet auf der in Abschnitt 2.3 präsentierten „Observable“ Klasse statt und legt diese auch mit „T“ als Rückgabewert fest.

In die zweite Kategorie fallen die Methoden „mapWithState“ und „flatMapWithState“, die beide die gleiche Funktion wie sie auch bei „reduce“ verwendet wird erhalten. Ihnen ist es nicht gestattet eine neuen Zustand herzustellen, sie können lediglich mit ihm arbeiten. Die „flatMapWithState“ Methode grenzt sich dadurch ab, das sie erlaubt andere asynchrone Funktionalität auszuführen, die ebenfalls einen Wert vom Typ „Observable“ zurückgibt und dabei Seiteneffekte berücksichtigt.
Alle hier aufgezählten Methoden operieren auf den anfangs genannten generischen Parametern.

Somit ist im Quellcode klar ersichtlich, inwieweit eine Abänderung der Zustands gewollt ist und wann dieser lediglich mitsamt seiner Daten für den weiteren Verlauf benötigt wird. Hinzu kommt hier auch die automatische Handhabung von Seiteneffekten und asynchroner Funktionalität. Dies garantiert, dass der Zustand keine unabsichtliche Modifikation erfährt und zu jedem Zeitpunkt nur einer auf ihn zugreifen kann.

5.6 Controller als Bindeglied

Der Controller vereint die bisher beschriebenen Komponenten und „kontrolliert“ bzw. koordiniert anhand dieser den Aufruf der Business-Logik. Er stellt das Bindeglied zwischen der View, einer „Action“ und dem dazugehörigen „ActionTransormer“ dar und sorgt für den Aufruf von ebendiesem.

Dafür muss er über die vom Entwickler angedachte Form des Zustands, Intents, Action und Result informiert werden. Dies geschieht wie bei den vorherigen Komponenten durch die Angabe mehrerer generischer Parameter. Überdies erhält der Controller für

seine Konstruktion den initialen Zustand und eine Liste von Namen der zugehörigen „MviActionTransformer“ vom Entwickler.

In der initialen Phase verifiziert der „Controller“, dass alle „ActionTransformer“ von der hinterlegten „Action“ und dem „Result“ abstammen um späteren Konflikten vorzubeugen. Anschließend wird der „StateMananger“ mit dem überlieferten Zustand instanziiert. Im letzten Schritt erzeugt der „Controller“ alle „ActionTransformer“ die jeweils einen „StateManager“ zugewiesen bekommen und speichert diese in Form von Schlüssel (Name des Transformers) und dazugehörigem „ActionTransformer“ in dem Attribut „actionTransformerMap“ ab.

Im Controller selbst existieren zusätzlich zwei Funktionen und eine Variable, von denen nur letztere von außen zugänglich ist: „intent“. Sie stellt die in MVI beschriebene Intent-Funktion dar und dient als Einstieg in den unidirektionalen Kreislauf. Wird sie aufgerufen so führt sie die interne „mapToAction“ Methode aus, die mittels der im „MviIntent“ Interface definierten „toAction“ Methode, den „Intent“ in eine „Action“ umwandelt.

Die zweite und zugleich auch letzte Funktion ist die „model“ Methode. Sie nimmt die „Action“, ermittelt ihren Klassennamen und entnimmt auf Grundlage dessen den entsprechenden „ActionTransformer“ aus der „actionTransformerMap“. Die dort enthaltende Business Logik wird dann ausgeführt und das „Result“, verpackt in einem „Observable“, nach „oben“ durchgereicht.

5.7 View

Die „View“ stellt den obersten Teil des Frameworks dar und besitzt im Kern zwei Eigenschaften: Sie reicht die „Intents“ an ihren „Controller“ weiter und verarbeitet das von ihm erhaltene „Result“. Für beide werden erneut generische Parameter definiert. Einmal einer welcher aussagt, dass das eingefügte Objekt vom „MviController“ abstammen und ein zweiter, der dem „MviResult“ Interface entsprechen muss.

Hierfür etabliert das Framework die „MviView“ Klasse, von der die „View“ abstammen muss. Sie stellt die oben geschilderten Anforderungen sicher, indem sie den Entwickler zwingt, ein Attribut zu hinterlegen und eine Funktionen zu implementieren. Bei ersterem handelt es sich um das Objekt, dass der Entwickler als „MviController“ ersonnen hat, bei letzterem um eine Methode, die ein „Result“ als Parameter erwartet. In dieser soll vom Entwickler die Logik stehen, welche für das Aktualisieren des User Interface (UI) zuständig ist.

5.8 Anleitung zur korrekten Nutzung

Um die Handhabung des Frameworks und die für den Entwickler vorgesehenen Komponenten besser nachzuvollziehen, wird im folgenden eine kurze Schritt für Schritt Anlei-

tung dargelegt:

1. Anlegen der Klasse für den Zustand im Verbund mit „MviState“
2. Anlegen sämtlicher Actions im Verbund mit „MviAction“
3. Anlegen sämtlicher Intents im Verbund mit „MviIntent“ und obigen „Actions“
4. Anlegen sämtlicher Results im Verbund mit „MviResult“
5. Implementierung der Business Logik auf Basis der „MviActionTransformer“ Klasse
6. Implementierung des Controllers auf Basis der „MviController“ Klasse
7. Implementierung der View auf Basis der „MviView“ Klasse

Soll eine weitere „Action“ hinzugefügt werden, so muss diese lediglich in der jeweiligen Klasse mit ihrem „Result“ hinterlegt werden. Danach erfolgt die Erstellung des dazugehörigen „MviActionTransformer“ samt Business Logik. Zuletzt wird dieser der „KClass<*>“ Liste die der „MviController“ erwartet beigefügt.

Insgesamt ist der anfängliche Aufwand wie oft üblich etwas höher, beschränkt sich danach aber für jeden Zusatz auf das zuletzt beschriebene Vorgehen.

6 Prototypische Implementierung

In diesem Kapitel findet auf Basis des zuvor angefertigten Konzepts eine prototypische Implementierung statt. Zu Beginn wird dabei auf Entscheidungen eingegangen, die globalen Einfluss auf die Umsetzungen haben.

6.1 Grundlegende Entscheidungen

In diesem Teil werden Themen besprochen, die Auswirkungen auf den weiteren Verlauf der Implementation haben.

6.1.1 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 Entwicklung von Webseiten fand und erst später seinen Einzug in Android erhielt.

6.1.2 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 unterstützten: Kotlin. Sie wird von dem Unternehmen Jetbrains [1] 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 für bisherige Projekte gestattet. Kotlin bringt eine 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-Objekte“ und bilden eine funktionale Grundlage
- Datenklassen machen den Umgang mit unveränderliche Datenstrukturen einfach

[1] https://www.jetbrains.com/

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

Listing 5: 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 Exampaple.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 5 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 „Nullpointer-Exception“ retten. Des weiteren besteht ein größerer Fokus auf dem Einsatz von Konzepten aus der funktionalen Programmierung, welche durch Erweiterungsfunktionen für bspw.. Listen zum Einsatz kommen.

6.2 Funktionale reaktive Programmierung mit RxJava und RxKotlin

Die Implementierung des Obervers und Iterator Patterns mit dazugehörigen Operatoren, die Einhaltung von funktionalen Paradigmen und die Berücksichtigung von asynchroner als auch paralleler Ausführung von Funktionalität (und deren Synchronisierung) ist eine äußerst komplexe Herausforderung, die viel Erfahrung und Zeit erfordert. Deshalb wird in diesem Fall auf eine bereits bestehende Implementierung zurückgegriffen, die in der Android Entwicklung oft zur Anwendung kommt: RxJava.

Diese Bibliothek ist eine Implementierung der „Reactive Extension“ API, welche für viele Plattformen und Programmiersprachen verfügbar ist. Sie hat ihrem Ursprung in „.NET/C#“ aus dem Hause Microsoft. Sie arbeitet genau wie die Java Stream API auf der Basis von Datenströmen und stellt dafür ein Fülle an Operatoren zur Verfügung.

Zusätzlich zu „RxJava“ existiert eine DSL („Domain Specific Language“) die Kotlin agnostisch ist: „RxKotlin“. Anhand Listing 6 soll der Umgang mit dieser Bibliothek in einem für den weiteren Verlaufen relevanten Umfang dargelegt werden.

Listing 6: RxJava/Kotlin Beispiel

```
1 // Erstellung eines Observables
2 val observable = Observable.just("Hello")
3
4 observavble
5  .map { text: String ->
6   text + "World"
7  }.subscribe(::println)
```

6.3 Zustand (State) und der StateManager

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 der Zustand kopiert

und während diesem optionalem 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 13 aufzeigt, gestaltet das den Umgang mit unveränderlichen Klassen verhältnismäßig einfach.

Listing 7: 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.

6.3.1 Reflexion (Introspektion)

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

1. Debugger
2. Interpreter
3. Objektserialisierung
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 8: 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 9: 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, durchweg 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 10: Override Annotation

```
1
2 class MyClassJava {
3
4 @Override
5 public String toString() {...}
6 }
7
8 class MyClassKotlin {
9
```

```
// Hier ist "override" teil der Deklaration
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.

6.3.2 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 11: 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 12: 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.

6.3.3 Überprüfung durch Reflexion

Um die Wartung von generiertem Code, den Aufwand der Implementierung und damit einhergehenden Zeitaufwand zu umgehen, fällt die Entscheidung auf Reflexion.

Im ersten Schritt muss verifiziert werden, dass es sich bei der Klasse für den Zustand um eine „data“ Klasse handelt. Hierfür bietet die Kotlin Reflexion API ein Attribut:

Listing 13: Kotlin „isData“ Attribut

```
data class DataClass

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

Sollte diese Auswertung negativ ausfallen, so wird eine „IllegalStateException“ geworfen und dem Entwickler mitteilt, dass ohne eine „data“ Klasse nicht fortgefahren werden kann.

Ist die Überprüfung jedoch erfolgreich, so muss im weiteren Vorgehen 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. In einer „Schleife“
   - 3.1. Prüfen, inwiefern das Attribut „var“ als Zugriffsmodifikator verwendet wird
     Bei Gebrauch zusätzlich den Typen auf Unveränderlichkeit prüfen
   - 3.2. Typen auf Unveränderlichkeit prüfen
   - 3.3. Eine Fehlernachricht generieren, mit den veränderlichen Attributsnamen
4. Eine Fehlernachricht mit allen veränderlichen Attributen oder einen leeren Text zurückgeben

Die Funktion zu Überprüfung wird dabei als Erweiterungsfunktion 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.

Innerhalb der Erweiterungsfunktion erfolgt der Zugriff auf die Liste aller Attribute der Klasse (Listing 14).

Listing 14: Erweiterungsfunktion „KClass“ mit Liste aller Attribute

```
internal fun KClass<*>.findMutableProperties(): String =
 members // Liste mit allen Attributen
 // ...
```

Zu beachten ist dabei, dass bei der „data“ Klasse für jedes Attribut ein zusätzliches generiert wird, welches mit dem Wort „component“ beginnt und am Ende eine Zahl von eins bis n (= Anzahl der Attribute) stehen hat. Durch dieses kann das Verfahren der destrukturierenden Zuweisung angewandt werden. Hierbei können aus dem Objekt Daten extrahiert und in (mehreren) Variablen abgelegt werden. Dies ist z.B. nützlich, wenn eine Funktion zwei Werte zurückgeben, oder eine „Map“ mit Schlüssel und zugehörigem Wert gleichzeitig durchlaufen werden soll. Listing 15 zeigt beide Anwendungsfälle.

Listing 15: Destrukturierende Zuweisung

```
data class Person(
 val forename: String,
 val surname: String
)

fun fetchPerson() = Person("forename", "surname")

val (forename, surname) = fetchPerson()

for ((key, value) in map) {
 // ....
}
```

Damit keine doppelte Überprüfung eines Attributs durch die „componentN“ Variable erfolgt, müssen diese vorher aus der Liste entfernt werden. Zu diesem Zweck wird eine Filter Methode angewandt, die den Namen jeden Attributs daraufhin untersucht. Entspricht es dem Suchmuster, so wird es - entsprechend Listing 16 - aus der Liste gelöscht.

Listing 16: Filtern

```
members.filter { property ->
 property.name.startsWith("component").not()
}
```

im weiteren Verlauf herausgefiltert.

Daraufhin soll geschaut werden, inwiefern eine Variable vorliegt, welcher ein neuer Wert zugewiesen werden kann. Dies ist der Fall, wenn dem Namen der Variable das Schlüsselwort „var“ vorausgeht. Mit der Reflexion API wird dies über die „KMutableProperty<*>“ Klasse signalisiert. Diesem schließt sich die Überprüfung auf Unveränderlichkeit des Subtyps an. Darunter fällt beispielsweise eine „MutableList“ in Kotlin, welcher Elemente direkt hinzugefügt oder entfernt werden können. Für jeden Fund wird eine Fehlermeldung erzeugt, die angibt welches Attribut betroffen ist. Diese wird an die vorherige Fehlermeldung angeheftet und zum Schluss im Gesamten als „String“ zurückgegeben. Hierfür muss die Liste in einer Schleife durchlaufen werden und in jedem Durchlauf die bisherigen Fehlermeldung in Form eines „String“ bereitstellen.

Eine Möglichkeit wäre die Standard „for“ Schleife in Kombinationen mit einer veränderlichen String Variable. Dies ist jedoch nicht dem Ziel der Fokussierung auf funktionale Konzepte vereinbar und ist daher keine Option. Zur Hilfe kommt die „fold“ Methode: Sie summiert den Wert, welcher mit einem Anfangswert festgelegt wird, und führt auf Basis einer Funkion und dem letzten Wert eine Operationen aus, die immer als Rückgabewert den Typ des Anfangswert hat. Zuletzt gibt sie den summierten Wert zurück.

In diesem Fall ist der anfängliche Wert einer leerer „String“, an den Fehlermeldungen angehängt werden können. Dabei wird nicht der String selbst verändert, sonder es wird eine Kopie erzeugt. „String“ ist von Haus aus „Unveränderlich“. Darauf folgt die Funktionen, die die oben aufgeführten Schritte umsetzt. Ihr erster Parameter ist „errorMessage“ und entspricht dem Typ des Anfangswert (hier: String). Der zweite, „property“, ist vom Typ „KCallable<*>“ und bezeichnet einen Wert aus der Liste „members“. Es ist ein Attribut des Zustands das überprüft werden soll.

Im Funktionskörper befindet sich eine „when“ Expression die ist zu erst abfragt, inwiefern es sich um eine „KMutableProperty“ Klasse handelt. Trifft dies zum, so wird eine temporäre Fehlerbeschreibung erstellt und mitsamt des Attributs an die nächste Funktion weitergereicht. Diese schaut mittels „property.isSubtypeOf(MutableList::class)“ bspw. ob es sich um eine „MutableList“ handelt. „isSubtypeOf“ entstammt nach Listing 17 der „KCallableUtil“ Klasse und nimmt die Überprüfung des Subtypen vor.

Listing 17: KCallableUti

```
1 internal fun KCallable<*>.isSubtypeOf(clazz: KClass<*>)
2  : Boolean = returnType.isSubtypeOf(clazz.starProjectedType)
```

Der Zugriffsmodifikator „internal“ beschränkt den Anwendungsbereich der Funkion auf das Framework selbst und kann nicht vom Entwickler in Anspruch genommen werden.

Finde auch diese Funktionen ein veränderliches Attribut, so wird die überreichte tem-

poräre Fehlerbeschreibung um eine weitere erweitert und der „fold“ Methode zurückgegeben. Dieser wird dann wieder im nächsten Durchlauf bereitgestellt bis die Liste abgearbeitet wurde.

Listing 18: fold

```
1
2 .fold("") { errorMessage: String, property: KCallable<*> ->
3  when (property) {
4   is KMutableProperty<*> -> {
5    val tmpErrorMessage = "$errorMessage\n${property.name}
6     is not allowed to be var"
7    isSubtypeMutable(property, tmpErrorMessage)
8   }
9
10  else -> isSubtypeMutable(property, errorMessage)
11 }
12}
13
14fun isSubtypeMutable(property: KCallable<*>,
15       errorMessage: String)
16  = when {
17   property.isSubtypeOf(MutableList::class) ->
18    "$errorMessage\n${property.name} is
19     not allowed to be mutable"
20
21   property.isSubtypeOf(Function::class) ->
22    "$errorMessage\n${property.name}
23    is not allowed to be a function"
24
25   else -> errorMessage
26}
```

Aufgrund der Tatsache, dass der Subtyp in beiden Fällen im „when“ Ausdruck verifiziert wird, wurde diese Logik in eine Methode mit dem Name „isSubtypeMutable“ ausgelagert.

6.3.4 StateManager

Die oben aufgeführte Implementierung für die Überprüfung der Klasse für den Zustand findet ihren Platz in der in der in Kapitel 5.2 vorgestellten „StateManager“ Klasse.

Bevor die Klasse implementiert werden kann, muss das „MviState“ Marker Interface wie in Listing 19 definiert werden.

Listing 19: „MviState“ Interface

```
interface MviState
```

Nun erfolgt die Nutzung von diesem in der Definition des „StateManager“ und seinem Konstruktor mit dem „state“ Attribut:

Listing 20: „StateManager“ mit „MviState“

```
class StateManager<S : MviState>(
 private var state: S
) {
 // ....
}
```

Dabei beschreibt das „S“ den generischen Part und der Doppelpunkt Notation sagt aus, dass das Objekt, welches für „S“ eingesetzt wird „MviState“ implementieren muss. Die „state“ Variable ist aufgrund des vorangestellten „private“ nur innerhalb Klasse selbst sichtbar. Um die geforderte Eigenschaft der Mutation zu entsprechen, wird „var“ verwendet.

Im „init“ Block des „StateManager“ kommt die „findMutableProperties“ Erweiterungsmethode dann zum Zuge. Ist der erhaltende „String“ nicht leer, so wird eine „IllegalStateException“, mit dem „String“ als Inhalt, geworfen und das Programm beendet.

Für die Umsetzung der „setNewState“ Methode ist zu beachten, dass diese die Eigenschaft der „Threadsicherheit“ erfüllen muss. Um das zu erreichen, wird auf die „synchronized“ Funktion gesetzt, welche ein Objekt als „Lock“ nutzt. Jeglicher Code der sich innerhalb dieses Blocks befindet, kann nur von einem „Thread“ zurzeit ausgeführt werden. Bevor der „Thread“ diesen Bereich betritt, muss er vom zugehörigen Objekt einen „Lock“ anfordern. Sollte dieser in Verwendung sein, so muss der „Thread“ warten bis der „Lock“ wieder frei ist.

Um keine unnötige Zuweisung durchführen zu müssen, wird in einer „if“ Abfrage der neue mit dem aktuellen Zustand verglichen. Nur wenn ein Unterschied entdeckt besteht findet eine Zuweisung statt. Mit Listing 21

Listing 21: „setNewState“ Methode

```
private fun setNewState(newState: S) {

 // "this" ist die aktuelle Instanz des "StateManager"
 synchronized(this) {
  if((state == newState).not()) state = newState
```

```
6   }
7 }
```

kann darauf die „reduce“ Methode in den „StateManager“ integriert werden. Sie macht von Kotlins „First-Class-Objekt“ Eigenschaft Gebrauch, die es erlaubt, Funktionen als Übergabeparameter zu definieren. Listing 22 macht dies Sichtbar.

Listing 22: „reduce“ Methode

```
1 fun reduce(reducer: (currentState: S) -> S) {
2  setNewState(reducer(state))
3 }
```

Die letzte Methode die zu Implementieren gilt ist „map“. Im Gegensatz zu „reduce“ benötigt sie eine zweiten generischen Parameter „<R>“, welcher im Zuge der Methoden Definition hinterlegt wird. Ihn zeichnet aus, das ihm mit „T : Any“ eine „Upper-bounded wildcard“ zugrunde liegt. Dies bestimmt, das jedes Objekt von „Any“ („Object“ in Java) abstammen muss und, viel wichtiger, nicht „null“ sein darf. Genau wie „reduce“ wird auch hier eine Funktion als Parameter genutzt. Letztendlich wird durch „R“ der Rückgabewert festgelegt, wie Listing 23 beweist.

Listing 23: „map“ Methode

```
1 fun <R : Any> map(mapper: (currentState: S) -> R): R {
2  return mapper(state)
3 }
```

Damit erfüllt diese Komponente alle Anforderung und ist vollends implementiert.

6.4 Intent, Action und das Result als Interfaces

Bevor die nächst größeren Klassen in Angriff genommen werden können, müssen ergänzend „Marker“ Interfaces für die im Titel stehenden Komponenten erstellt werden. Das einzige Interface, welches von einer einfachen „Interface“ Definition abweicht ist „MviIntent“. Genau wie in Kapitel 6.3.4 wird ein generischer Typ mit einem „Upper Bound“ benutzt. In diesem Fall wird „MviAction“ angegeben.

Listing 24: „Marker“ Interfaces

```
1 interface MviAction
2
3 interface MviIntent<A : MviAction> {
4
5  fun toAction(): A
6 }
7
8 interface MviResult
```

6.5 MviActionTransformer

Für diese Komponente wird etwas gesucht, dass sich in den Kreislauf auf Basis von „RxJava“ und „Observables“ einfügt, einen Wert erwartet und einen (anderen) zurückliefert. Außerdem soll es dafür dienen, die Busniess Logik „sichtbar“ zu trennen. Das Ziel ist daher die Transformation von Werten mittels Komposition von Funktionen.

Hierfür bietet sich ein Interface namens „ObservableTransformer“ an, das die „RxJava“ Bibliothek zur bereitstellt. Dieses wird wie folgt definiert:

Listing 25: ObservableTransformer

```
public interface ObservableTransformer<Upstream, Downstream> {
 /**
 * Applies a function to the upstream Observable and
 * returns an ObservableSource with
 * optionally different element type.
 * @param upstream the upstream Observable instance
 * @return the transformed ObservableSource instance
 */
 @NonNull
 ObservableSource<Downstream> apply(
  @NonNull Observable<Upstream> upstream);
}
```

„Upstream“ steht für den Wert, der am Anfang vorliegt und „Downstream“ für den, der am Ende herauskommen muss. Listing 26 demonstriert, wie es die beschriebenen Anforderungen erfüllt.

Listing 26: ObservableTransformer Anwendung

```

val intToStringTransformer =
 ObservableTransfromer<Int, String> { upstream ->
  upstream.map { integer ->
   integer.toString()
  }
 }

Observable.fromIterable([1,2,3])
 .compose(intToStringTransformer)
 .subscribe(::println)
```

Die „intToStringTransformer“ Variable findet ihren Einsatz im Verbund mit dem „compose“ Operator und verwandelt jeden „Integer“ in einen „String“. Hierzu ruft dieser die „apply“ Methode im „ObservableTransfer“ auf.

Von diesem „ObservableTransformer“ erbt die „MviActionTransformer“ Klasse und muss dementsprechenden die generischen Parameter übernehmen. Hinzu kommt, das der benötigte „StateManger“ für seine Vollständigkeit ebenfalls einem bedarf. In der Klassen Definition von Listing 27

Listing 27: MviActionTransformer Definition

```
abstract class MviActionTransformer<S : MviState, A : MviAction,
 R : MviResult>
```

wird vom dem zudem vom „abstract“ Schlüsselwort Gebrauch gemacht. Dies führt dazu, dass von dieser Klasse keine Instanz erstellt werden kann, da sie als Basisklassen herhalten soll. Somit ist auch keine Implementation der „apply“ Methode von Nöten. Dies wird erst in den vom „MviActionTransformer“ abstammenden Klassen verlangt.

Im nächsten Schritt wird genau wie im „StateManager“ zu Beginn die Methode implementiert, welche den derzeitigen Zustand verändern darf. Im Unterschied zur der dort ansässigen muss dieser hier dem reaktiven Aspekt Genüge tun, indem als Rückgabetyp ein „Observable“ verwendet wird. Im ersten Anlauf ergeben sich dafür zwei Optionen:

1. Eine Funktion, die einen generischen Parameter (Wert in Arbeit) sowie eine „reducer“ Funktion erhält und intern ein „Observable“ generiert
2. Eine Funktion, die eine „reducer“ Funktion erhält und den bereits vorgestellten „ObservableTransformer“ nutzt

Für Nummer eins kann folgende Implementation in Listing 28 herhalten:

Listing 28: 1. Versuch

```
protected fun <T : Any> reduceState(
 value: T,
 reducer: (currentState: S) -> S
): Observable<T> {

 return Observable.fromCallable {
  stateManager.reduce { currentState ->
   reducer(currentState)
  }
  value
 }
}
```

Hier wird mithilfe von „Observable.fromCallable“ ein „Observable“ erzeugt, in dessen Funktionsrumpf die „reduce“ Methode vom „StateManager“ mit der bereitgestellten „reducer“ Funktion aufgerufen wird. Damit der derzeitige Wert im weiteren Verlauft

nicht verloren geht, wird dieser am Ende zurückbegeben.

Bei Nummer zwei kann der „value“ Parameter gespart werden, da dieser über „upstream“ mitgeliefert wird:

Listing 29: 2. Versuch

```
protected fun <T : Any> reduceState(
 reducer: (currentState: S) -> S) =

 ObservableTransformer<T, T> { upstream: Observable<T> ->
  upstream.map { value ->
   stateManager.reduce { currentState ->
    reducer(currentState)
   }

   value
  }
}
```

Beide Version erfüllen ihren Zweck, wirken allerdings wie Listing **??** zeigt etwas umständlich in ihrer Anwendung, trotz der Verwendung von Anonymen bzw. Lambda-Funktionen.

Listing 30: 2. Versuch

```
upstream.map { TestMviResult.TestResult }
// 1
.flatMap { reduceState(it) { currentState ->
  currentState
} }
// 2
.compose(reduceState { currentState ->
 currentState
})
```

Ein andere Alternative stellt die Entwicklung eines Operators eigens für „RxJava“ dar, jedoch gestaltet sich dies als durchaus komplex. [19] Damit dem aus den Weg gegangen und auch die Einfachheit der Methode verbessert werden kann, ist der Einsatz einer Erweiterungsfunktion naheliegend. Bevor sich diesem allerdings gewidmet wird, muss ein Konzept der Kotlin Programmiersprache behandelt werden: „inline“ Funktionen.

Infolge der Tatsache, das Kotlin zu JVM Bytecode kompiliert wird und Java von Haus aus keine Funktionen höherer Ordnung unterstützt, muss für jedes dieser eine Klasse bzw. ein Objekt erstellt werden. Besonders wen, dies innerhalb einer Schleife passiert, geht es einher mit der zusätzlichen Zuweisung von (Arbeits-)Speicher und kostet damit

auch Performanz hinsichtlich des Leistungsverhalten. Um dem Vorzubeugen existiert die Möglichkeit, den Inhalt einer Funktionen an die Stelle zu kopieren an der sie aufgerufen wird. Für diesen Zweck stellt Kotlin das Schlüsselwort „inline“ bereit, dass wie in Listing 31 in der Methodendefinition platziert wird.

Listing 31: inline

```
fun main() {
 multiplyByTwo(5) { println("Result is: $it") }
}

inline fun multiplyByTwo(
 num: Int,
 lambda: (result: Int) -> Unit)
: Int {
 val result = num * 2
 lambda(result)
 return result
}
```

Daraus macht der Compiler:

Listing 32: inline kompiliert

```
public static final void main(@NotNull String[] args) {
 //...
 int num$iv = 5;
 int result$iv = num$iv * 2;
 String var4 = "Result is: " + result$iv;
 System.out.println(var4);
}
```

Listing 33 zeigt, dass die „multiplyByTwo“ Methode nicht aufgerufen wird.

Eine Gefahr die in Betracht gezogen werden muss ist, dass sich innerhalb der übergebenen, anonymen Funktionen ein „return“ Statement befinden kann. Dies mag dazu führen, dass die umschließende (Aufrufer-) Funktion eventuell früher verlassen wird als gewollt. Listing **??** verdeutlicht die Problematik.

Listing 33: inline mit „return“

```
// Kotlin
fun main() {
 println("Start")

 multiplyByTwo(5) {
```

```
    println("Result is: $it")
    return
  }

  println("Ende")
}

// Java
public static final void main(@NotNull String[] args) {
  String var1 = "Start";
  System.out.println(var1);
  int num$iv = 5;
  int result$iv = num$iv * 2;
  String var4 = "Result is: " + result$iv;
  System.out.println(var4);
}
```

Wie zu erkennen ist nach „System.out.println(var4);“ Schluss und „println('Ende');“ wurde nicht übernommen. Auch in dieser Angelegenheit kann Kotlin mit einem weiteren Schlüsselwort aushelfen: „crossinline“. Dies besagt ganz einfach, dass ein „return“ Statement in der übergebenen Funktion nicht gestattet ist.

Unter mithilfe der soeben vorgestellten Konzepten und Schlüsselwörter lässt sich die „reduceState“ Methode nach Listing 34 effizient und elegant umsetzen.

Listing 34: „reduceState“ Methode

```
typealias Reducer<S, T> = (value: T, currentState: S) -> S

protected inline fun <T : Any> Observable<T>.reduceState(
  crossinline reducer: Reducer<S, T>
): Observable<T> =
  map { value: T ->
    stateManager.reduce { currentState ->
      reducer(value, currentState)
    }

    value
}

someObservable
  .reduceState { newValue, currentState ->
    currentState.copy(value = newValue)
  }
```

```
18  .map { newValue -> \\... }
```

Wie zusehen ist, wird die durch „currentState.copy(...)“ erzeugte Zustands Klasse mit dem neuen Wert nicht an die nächste Methode weitergereicht. Dies sorgt dafür, dass den vordefinierten Methoden ein Zugriff auf den Zustand vorbehalten ist.

Als Ergänzung wird ein weitere Eigenschaft von Kotlin herangezogenen: Die Möglichkeit „Aliase“ unter Einsatz vom Schlüsselwort „typealias“ zu bestimmen. Diese sind besonders hilfreich wenn verschachtelte Typen („Observable<Result<List<String»>“) oder Funktionen als Parameter vorliegen und häufiger gebraucht werden.

Ein Vorteil dieser Lösung ist, dass die Erweiterungsfunktion nur innerhalb einer Klasse sichtbar ist, die von „MviActionTransformer“ abstammt. Unter diesen Voraussetzungen findet auch die Implementierung der anderen Methoden „mapWithState“ und „flatMapWithState“ statt. Am Ende lässt sich damit ein wie in Listing 35 Beispielhafte Implementierung erzielen.

Listing 35: Beispiel Implementation

```
1  class SomeTransformer(
2   stateManager: StateManager<SomeState>
3  ) : MviActionTransformer<
4   SomeState,
5   SomeAction,
6   SomeResult>(stateManager) {
7
8   override fun apply(upstream: Observable<SomeAction>)
9   : ObservableSource<SomeResult> =
10
11  upstream
12    .flatMapWithState { currentState: SomeState,
13       data: SomeAction ->
14
15      doSomethingWithState(currentState)
16    }
17    .reduceState { newValue: SomeNewValue,
18         currentState: SomeState ->
19
20      currentState.copy(value = newValue)
21    }
22    .map { SomeResult }
23 }
```

6.6 MviController

Ähnlich wie der „MviTransformer“ erhält auch diese Komponente mehrere generische Parameter, sowie den geforderten initialen Zustand und eine Liste mit allen zugehörigen „MviActionTransormer“ Klassen. Beide sind hierbei durch „val“ an ihren anfänglichen Wert gebunden. Dies verhindert jedoch nicht, das Daten innerhalb einer Klasse verändert werden können. So kann in Java beispielsweise einer Liste ein Wert hinzugefügt oder aus dieser entfernt werden.

Um auch in diesem Fall die Unveränderlichkeit zu wahren, ist der Griff zu Drittanbieter-Software (3rd Party Libraries zu Englisch) eine gern genutzte Option. Mit Kotlin ist dies aber nicht zwingend nötig, denn es unterscheidet zwischen veränderlichen und unveränderlichen Listen. Ersteres wird durch „MutableList“ und zweites durch „List“ erreicht. Hierbei ist allerdings wichtig zu wissen, dass beide keine Implementierungen sondern lediglich ein Interface darstellen, in dem Methoden für Schreiboperation präsent sind oder nicht. Dafür wird eine Liste auf Basis der Java „Collection“ erzeugt und in das jeweilige Interface (implizit) umgewandelt. Dies hat zur Folge, dass ein „List“ zu jederzeit in eine „MutableList“ verwandelt („casted“) werden kann:

Listing 36: „List“ und „MutableList“

```
1 val list = listOf(1,2,3)
2 val mutableList = mutableListOf(1,2,3)
3
4 list.add(4) // nicht vorhanden
5 mutableList.add(4) // vorhanden
6
7 (list as MutableList).add(4) // durch den "cast" möglich
```

Trotz dieser Problematik ist dies für das weitere Vorgehen nicht von großer Relevanz und der Konstruktor kann wie in Listing 37 niedergeschrieben werden.

Listing 37: Konsruktor

```
1 abstract class MviController<S : MviState, I : MviIntent<A>,
2   A : MviAction, R : MviResult>(
3  private val initialState: S,
4  private val actionTransformerKClassList: List<KClass<*>>)
```

Um die „actionTransformerMap“ in Form einer „HashMap<String, ObservableTransformer<A, R»“ zu füllen, muss ersten Schritt aus der „KClass“ eine Instanz vom Typ „ObservableTransformer“ erstellt werden. Hierzu kann auf ein Attribut namens „primaryConstructor“ zurückgegriffen werden, welches - wie der Name vermuten lässt - den primären Konstruktor als „KFunction“ bereitstellt. Der Tatsache geschuldet, das der Rückgabewert „null“ sein kann, muss diesem ein doppeltes Ausrufezeichen angestellt

werden um dem Compiler mitzuteilen, dass dem nicht so ist.

„KFunction“ wiederum birgt eine Funktion, die es erlaubt den Konstruktor aufzurufen. Infolge der Tatsache das dieser Argumente beinhalten kann bietet die Funktion „call“ einen Parameter mit dem Schlüsselwort „vararg“ und dem Typ „Any“. Damit können beliebig viele Argumente mit unterschiedlichen Werten übergeben werden. Überdies muss darauf geachtet werden, dass die Reihenfolge genau der entsprechen muss, wie sie im Konstruktor des zu erstellenden Objekts anzutreffen ist. Zuletzt erfolgt über einen expliziten „cast“ die Umwandlung von „Any“ zum „ObservableTransformer“.

Hier macht sich der angesprochene Verlust der Typsicherheit bei der Verwendung von Reflexion bemerkbar: Wird die Reihenfolge nicht beachtet, so erfolgt kein Hinweis des Compiler - stattdessen kommt es zu einem Fehler zur Laufzeit des Programms. Und wird nach „as“ nicht der korrekte Typ angegeben, so führt dies ebenfalls zu einem Laufzeitfehler. Nach Listing 38

Listing 38: Konsruktor

```
val transformer = kClass
    .primaryConstructor!!
    .call(StateManager(initialState))
     as ObservableTransformer<A, R>
```

ist der „ObservableTransformer“ erzeugt wurden, welcher als Wert in der „HashMap“ platz nehmen wird.

Als nächstes gilt es den Schlüssel zum gerade erstellten „ObservableTransformer“ in Form des Namens seiner „MviAction“ abstammenden Klasse zu finden. Dieser lässt sich aus der Liste der Supertypen der jeweiligen „MviController“ Klasse entnehmen. Dafür gibt es das Attribut „superTypes“, welches diese als Liste in der Reihenfolge in der sie angegeben wurden enthält. Der „MviController“ befindet sich an erster Stelle und kann daher über den Index Null angesprochen werden. An diesem selbst liegt das Argument der „MviAction“ an zweiter Position und kann mittels des Attribut „arguments“ über den Index eins angesprochen werden. Zuletzt ist mit dem „type“ Attribut und der Methode „toString“ der Klassenname der „MviAction“ verfügbar. Listing 39 zeigt die durch die Beschreibung erhaltene Implementation.

Listing 39: „MviAction“ Name

```
val actionName = kClass
  .supertypes[0]
  .arguments[1]
  .type
  .toString()
```

Da es sich um eine Liste von „MviActionTransfromer“ handelt, muss dies für jedes Ele-

ment vorgenommen werden. Um dem funktionalem Paradigma treu zu bleiben wird auf die „Collections“ API zurückgegriffen die, ähnlich der Java „Stream“ API, Methoden anbieten welche auf Basis von anonymen Funktion arbeiten. Gesucht wird eine Methode, die über die Liste iteriert und einen neuen Wert zurückgeben kann. Dafür eignet sich die „map“ Methode. In ihr wird der Code aus Listing 38 und 39 angewandt, wobei die Klasse und der Name als Paar zurückkommen. Ist die ganze Liste durchlaufen wurden, so ist das Ergebnis eine Liste von Paaren mit „ObservableTransformer“ und „String“ (Name der „MviAction“) als Inhalt.

Im letzten Schritt muss aus der Liste eine „HashMap“ werden. Um den Fluss mit an aneinandergereihten Funktionen aufrecht zu erhalten und keine temporären Variablen anlegen zu müssen, wird erneut in Kotlins Trickkiste gegriffen. Im Zusammenspiel aus Funktionen höherer Ordnung und Erweiterungsfunktionen lassen sich sogenannte Funktionsliterale mit Empfängern bilden. Listing 40 gibt ein Beispiel.

Listing 40: Funktionsliteral mit Empfänger

```
buildList<Int> {
 add(1)
 add(2)
 removeAt(0)
}

fun <T> buildList(block: MutableList<T>.() -> Unit): List<T> {
 val mutableList: MutableList<T> = mutableListOf()
 mutableList.block()
 return mutableList.toList()
}
```

In dieser Methode wird „MutableList<T>“ als Empfängern festgelegt und eine temporäre, anonyme Funktionen erweitert. Die in der „buildList<Int>“ anonymen Funktion spezifizierten Operation werden im Folge des Aufrufs auf der Liste angewandt. Anders ausgedrückt handelt es sich hierbei schlicht um temporäre Erdweiterfunktionen die für den Zeitraum des Aufrufs der Funktionen existieren.

Kotlin geht noch einen Schritt weiter und bietet dies Art von Funktionen auf generischen Typen an, genannt „scope functions“. Eine Methode ist „apply“, die obiges Beispiel weiter vereinfacht:

Listing 41: apply Funktion

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
 block()
 return this
}

```

```
mutableList<Int>().apply{
 // ...
}
```

Hiermit lässt sich der Rest Implementieren und eine „HashMap“ erzeugen:

Listing 42: apply Funktion

```
val actionTransformerMap = actionTransformerKClassList
 .map { \\ ...}
 .run {
   HashMap<String, ObservableTransformer<A, R>>()
   .apply {
    putAll(this@run.asSequence())
 }
}
```

Mit der „actionTransformerMap“ im Rücken erfolgt die Umsetzung der Model-Funktion. Sie wird als eine auf dem „Observable<A>“ fungierende Erweiterungsfunktion definiert und entnimmt der „actionTransformerMap“ auf Basis des vorliegenden „MviAction“ Namens ein „ObservableTransformer“. Dieser wird mit „compose“ innerhalb der „flatMap“ Methode ausgeführt.

Damit lassen auch die letzten beiden Methoden implementieren, wobei Listing 43 das Endergebnis darstellt:

Listing 43: Endergebnis

```
val intent = ObservableTransformer<I, R> {
 upstream: Observable<I> ->
 upstream
  .mapToAction()
  .model()
}

private fun Observable<I>.mapToAction(): Observable<A> =
 map { intent -> intent.toAction() }

private fun Observable<A>.model(): Observable<R> =
 flatMap { action ->
  compose(actionTransformerMap[action::class.qualifiedName])
 }
```

6.7 MviView

Die letzte Komponente kommt in den Klassen zum Einsatz, die für Darstellung des UI zuständig sind: „Activty“, „Fragment“ und „Custom Views“.

Listing 44: MviView

```
interface MviView<C : MviController<*, *, *, *>, R : MviResult>{

 val controller: C

 fun render(result: R)
}
```

Bevor ein Beispiel für eine Anwendung gegeben werden kann, muss geklärt werden, wie man „Intent“, „Action“ und „Result“ in Kotlin Implementiert.

6.8 Intent, Action, Result - Mögliche Anwendung

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 45: Intent Klasse

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

Für diesen Fall eignet sich eine Klasse mit einem generischen Typ als Attribut, wie Listing 45 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 „Intent“ transportiert die eigentliche Absicht nicht

Mangel Nr. 1 lässt sich mit einer einfachen Abwandlung von Listing 45 behoben werden.

Listing 46: Intent Klasse

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

Hierfür muss lediglich von Kotlins Notation für „null“-Typen 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 48 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 47: 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 47 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 48: 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 „payload“ 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 „NullPointer 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 49 gehört unter anderem zu den Summen Typen.

Listing 49: 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 „Produkt“ 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 48 funktionaler und eleganter zum implementieren:

Listing 50: Intents als sealded class

```
sealed class Intent {
 // "object" erzeugt ein Singleton
 // es ist keine Instanziierung möglich
 // Attribute folglich nicht gestattet
 object GoBack : Intent()
```

```
6   class DisplayItemDetails(item: Item): Intent()
7 }
```

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 und nicht über mehrere verstreut sein darf.

Der hier aufgeführte Ansatz kann dabei ebenfalls für alle „Actions“ und „Results“ verwendet werden.

7 Evaluation

In diesem Kapitel werden die einzelnen Anforderungen betrachtet und auf die Vollständigkeit ihrer Implementation untersucht.

7.1 Verwalten des Zustands

Durch den „StateManager“ Klasse ist diese Anforderung zu großen Teilen abgedeckt. Ein neuer Zustand kann ausschließlich indirekt unter sicheren Bedingungen gesetzt werden. Des weiteren steht Funktionalität für den Zugriff auf den Zustand zur Verfügung, ohne dabei eine Änderung am Original vornehmen zu können. Ein interessanter Zusatz wäre die Option den Zustand observieren lassen zu können und jedem Beobachter über eine Änderung zu informieren. Damit bestände die Möglichkeit, durch eine Modifikation am Zustand ohne größeren Aufwand anderweitige Logik ausführen zu lassen. Dadurch würde das Framework ebenfalls zusätzlich an Reaktivität gewinnen.

7.2 Überprüfung des Zustands als unveränderliche Datenstruktur

Mittels der angewendeten Reflexion ist dieser Punkt im Großen und Ganzen erfüllt. Ein bessere Lösung stellt jedoch die Generierung einer unveränderlichen Zustands Klasse (hier „data“ Klasse) mittels Annotation dar. Damit ginge man einer Überprüfung zur Laufzeit und dem damit einhergehenden Problemen (keine Typsicherheit etc.) aus dem Weg und stellt von Anfang an die Korrektheit des Zustands sicher.

7.3 Speichern des Zustands

Diese Anforderung ist im Prototypen nicht vorhanden. Für eine befriedigende Implementierung muss der Zustand sowohl Konfiguration Änderungen (Drehung der Ansicht etc.) als auch dass Beenden des Prozess der Anwendung überstehen. Für die Realisierung dieser Funktion ergeben sich unterschiedliche Ansätze:

1. Das persistente Speichern des Zustands bei jeder Änderung (bswp. mit sqlite)
2. Die Nutzung des „Jetpack“ „ViewModel“ in Kombination mit der „AbstractSavedStateVMFactory“ zum Speichern innerhalb eines „Bundles“
3. Das nutzen von „onSavedInstanceSate“ zum hinterlegen des Zustands in einem „Bundle“ und einer der jeweiligen Methoden zur Wiederherstellung

Nummer eins ist auf den erstem Blick mit dem geringsten Aufwand verbunden, besitzt jedoch zwei „Unannehmlichkeiten“. Abgesehen davon, dass jede Änderungen eine Schreib Operation auf dem Dateisystem bedeutet, muss der gespeicherte Zustand auch wieder zum richtigen Zeitpunkt gelöscht werden. Passiert dies nicht, so könnte sich der Nutzer innerhalb einer unerwarteten Ansicht wiederfinden.

Bei Nummer zwei besteht die Abhängigkeit zu Android spezifischen Implementierungen mit ihren potentiellen Eigenheiten. Dies stellt Möglicherweise eine Einschränkung in

der Flexibilität dar und kann unter Umständen dazu führen, das Tests unter erschwerten Bedingungen durchgeführt werden müssen. Auf der anderen Seite wird ein Großteil der Arbeit vom Framwork übernommen und ist damit bereits automatisiert. Dabei bleibt zu evaluieren, inwieweit die zugrundeliegende Implementation die Anforderungen erfüllt, oder ob eine manuelle Umsetzung zu bevorzugen ist.

Der letzte Ansatz ist zwischen Nummer eins und zwei anzusiedeln. Er birgt einen höheren Aufwand als Nummer 2, ist dabei aber vermutlich Vielseitiger. Auch hier muss abgewogen werden, wie diese Variante gegenüber Nummer 2 langfristig abschneidet.

Für zwei und drei muss der Zustand von „Parcelable“ abstammen, um im „Bundle“ abgelegt werden zu können. Dies wäre ein weiterer Einsatzort für Code Generierung, welcher die Implementierung vornehmen könnten.

7.4 Wiederherstellung des Zustands

Infolge der fehlenden Persistenz des Zustands findet auch keine Rekonstruktion von diesem statt. Die dafür zu Verfügung stehenden Ansätze sind Identisch zu den in Sektion 7.3 aufgeführten.

7.5 Bereitstellung eines Reducers

Dieser wird innerhalb des „StateManager“ und später durch den „MviActionTransformer“ dem Entwickler offeriert. Mithilfe eines Alias wird der eigentlichen Funktion als Parameter der Name „Reducer“ gegeben. Damit ist die Anforderung zwar erfüllt, bietet aber dennoch Potential für eine bessere Lösung.

7.6 Trennung von Business und Ansicht Logik

Durch die Kombination aus „MviActionTransformer“, „MviController“ und „MviView“ erfolgt die Implementierung und Ausführung der Business Logik strikt von der Ansicht Logik getrennt. Zudem ist der Zugriff auf den Zustand nur innerhalb des „MviActionTransformer“ gestattet, in welchem auch die Business Logik residiert.

Das Konzept bietet eine guten Ansatz, verliert durch die Anwendung von Reflexion jedoch an Robustheit. Auch in diesem Fall kann über den Einsatz von Code Generierung nachgedacht werden. Des weiteren wird nicht überprüft, inwieweit „Action“ und „Result“ im „MviActionTransformer“ übereinstimmen.

7.7 Handhabung von Seiteneffekten

Seiteneffekte werden im Framework nicht als solche gekennzeichnet oder gar besonders behandelt, sondern stellen die Standardmäßige Erwartung der Ausführung dar. In vielen Fällen findet eine Operation asynchron statt, in der Zugleich meist eine Form von I/O involviert ist. Das Framework nimmt dafür die „Observable“ Klasse und den Operator

„flatMap“ zur Hand.

Hiermit ist die Anforderung indirekt erfüllt, könnte durch eine spezielle Behandlung von Seiteneffekten jedoch verbessert werden.

7.8 Identifizieren von Intent, Action, State und Result

Die im Titel genannten Komponenten werden lediglich durch Interfaces definiert, welche abgesehen von „Intent“ keine weitere Funktionen vorgeben. Dadurch ist auch diese Anforderung für sich umgesetzt, bietet aber Platz für Verbesserungen.

7.9 Asynchrone Ausführung

Hierfür wird „RxJava“ und die zu der Bibliothek gehörenden Operatoren genutzt. Diese erfüllt den Zweck, weißt aufgrund ihrer Komplexität aber eine hohe Lernkurve auf. Koltin enthält eines in die Sprache integriertes Konzept, welches die asynchrone Programmierung synchron erscheinen lässt: „Coroutines“. [20] Die ist in seiner Anwendung wesentlich einfacher und kann daher als Alternative herangezogen werden.

7.10 Dogmatisches Framework

Durch die Komponenten von denen die Klassen des Entwicklers teilweise abstammen, die Überprüfung der Zustands Klasse und das vordefinierte Vorgehen durch den „MviController“ ist das Framework durchaus als „dogmatisch“ zu beschreiben.

7.11 Unidirektionaler Datenfluss

Dieser wird durch die Verwendung des „Observable“, dem „MviActionTransformer“ und der vorgefertigten Funktionen innerhalb des „MviController“ geboten.

7.12 Reaktiv & Funktional

Das Framework macht Gebrauch von reaktiven und funktionalen Konzepten: Observer Pattern, Komposition von Funktionen, unveränderliche Datenstrukturen, algebraische Datentypen, Musterabgleich sowie Funktionen höherer Ordnung. Es erfüllt damit grob die Idee dieser Paradigmen.

7.13 Kotlin spezifische Implementierung

Das Framework ist in Kotlin geschrieben und nutzt eine Vielzahl der bereitgestellten Konzepte, darunter Erweiterungsfunktionen, „inline“ für Funktionen und natürlich Funktionen höherer Ordnung. Diese Art der Implementierung wäre in Java nicht möglich gewesen und ist damit Kotlin spezifisch.

7.14 Zusätzliche Verbesserungen

Im Framework muss der Entwickler selber die Abonnements die durch das Abonnieren eines „Observable“ entstehen an der richtigen Stelle beenden. Passiert dies nicht, so kommt es zu Speicherlecks.

Ähnlich wie in dem im Abschnitt 4.4.1 vorgestellten Framework sollten „Middlewares“ für bspw. das Loggen von Daten eingeführt werden.

Zusätzlich ist es derzeit Möglich für eine „Action“ mehrere „MviActionTransformer“ zu erstellen oder diese doppelt oder gar mehrfach im „MviActionTransformer“ zu hinterlegen. Dies sollte verhindert werden um unnötigen Problemen aus dem Weg zu gehen, z.B. der Ausführung von Code doppelt.

8 Fazit & Ausblick

Hier wird Resümee gezogen und in die Zukunft geschaut.

8.1 Fazit

Das Framwork besitzt ein paar gute Ansätze, schwächelt aber in der Umsetzung. Hierfür ist vor allem ein schlechtes Zeitmanagement und Trägheit des Verfassers schuld. Die Grundidee von MVI kommt im der Implementierung zur Geltung und der Entwickler welcher auf dieses Framework zurückgreift erhält einen gewissem Grad an Unterstützung bei der Anwendung vom MVI.

Die größte Schwäche ist der Gebrauch von Reflexion. Sie mag in bestimmten Fällen Unabdingbar sein, kann in dieser Implementation allerdings an vielen Stellen durch Code Generierung substituiert oder gar komplett eliminiert werden. Ein Bespielt ist hierfür die Erstellung einer unveränderlichen Zustands Klasse auf Basis von Annotationen durch das Frameowork selbst. Damit fiele die Notwendigkeit für eine Überprüfung von diesem weg.

Kurz um: Idee nett, Umsetzung mangelhaft.

8.2 Ausblick

Eine Weiterentwicklung und Nutzung des Framework ist durchaus denkbar. Hierfür müssen einige Verbesserungen wie die angesprochene Nutzung von Code Generation an Stelle von Reflexion vorgenommen werden. Besonders das Ersetzen von RxJava mit den in Koltin integierten Coroutines ist von großem Interesse. Auch wenn dies keine Nutzung mehr in Java erlauben würde, ist es mit der steigenden Popularität von Kotlin vereinbar. Am Ende bleibt die Frage, inwieweit die Weiterentwicklung bei der Anwesenheit von anderen MVI Frameworks sinnvoll ist.

Abbildungsverzeichnis

Literatur

Bücher Referenzen

[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.

[20] Nishant Srivastava Filip Babić. *Kotlin Coroutines by Tutorials (First Edition): Mastering coroutines in Kotlin and Android*. Razeware LLC, 26. Apr. 2019.

Report Referenzen

[12] Trygve Reenskaug. *THING-MODEL-VIEW-EDITOR - an Example from a planningsystem*. The original MVC report. 12. Mai 1979. url: http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf (besucht am 23. 05. 2019).

[13] Trygve Reenskaug. *MODELS - VIEWS - CONTROLLERS*. 10. Dez. 1979. url: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf (besucht am 23. 05. 2019).

[14] Mike Potel. *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java*. Taligent, Inc., 1996. url: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

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 Referenzen

[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).

[15] Martin Fowler. *Model-View-Presenter (MVP)*. 18. Juli 2006. url: https://martinfowler.com/eaaDev/uiArchs.html#Model-view-presentermvp (besucht am 23. 05. 2019).

[16] 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).

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

[18] badoo. *MVI framework with events, time-travel, and more*. 25. Feb. 2018. url: https://github.com/badoo/MVICore (besucht am 23. 08. 2019).

[19] David Karnok. *Writing operators for 2.0*. 8. Mai 2018. url: https://github.com/ReactiveX/RxJava/wiki/Writing-operators-for-2.0 (besucht am 21. 08. 2019).