Inhaltsverzeichnis

1 Einleitung

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

1.1 Problemfeld

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

1.2 Ziel der Arbeit

1.3 Aufbau der Arbeit

2 Grundlagen

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

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

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

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

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

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

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

2.1.1 Flux

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

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

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

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

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

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

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

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

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

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

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

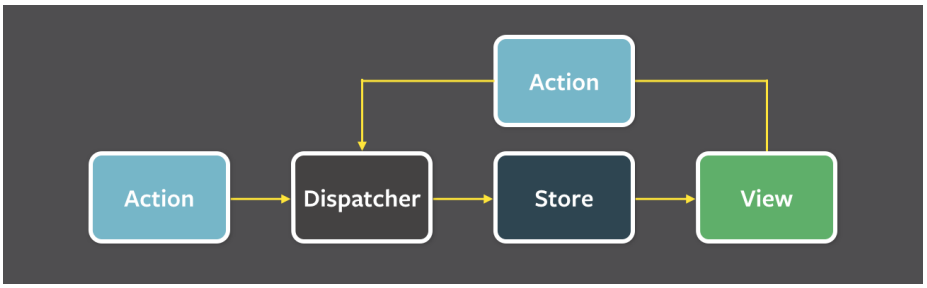

Abbildung 1: Datenfluss in der Flux Architektur

Anhand Abbildung 1 wird der unidirektionale Datenfluss deutlich erkennbar:

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

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

2.1.2 Redux

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

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

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

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

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

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

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

2.1.3 Elm

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

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

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

2.2 Funktionale Programmierung

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

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

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

3 Model-View-Intent

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

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

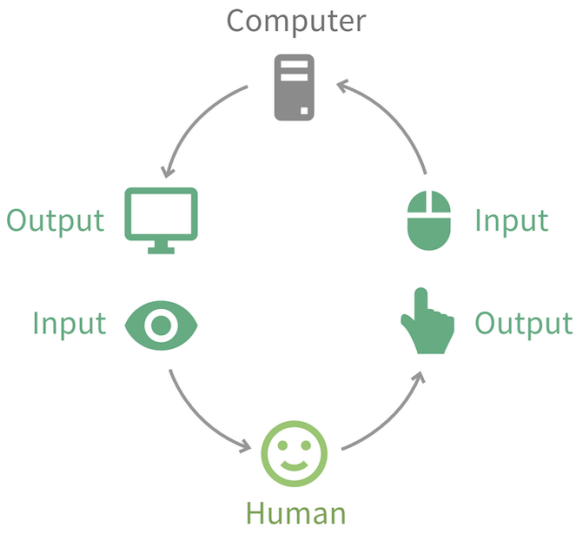

Abbildung 2: Nutzer und Computer als Input und Output

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

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

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

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

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

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

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

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

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

Listing 1: funktion

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

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

Listing 2: pseudo mvi implementation

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

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

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

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

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

Listing 3: pseudo mvi implementation

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

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

```
}

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

fun main(args : Array<String>) {

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

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

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

3.1 Reducer

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

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

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

3.2 Endlicher Automat

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

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

Abbildungsverzeichnis

Book References

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

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

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

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

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

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

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

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

[14] Sergi Mansilla. *Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code*. PRAGMATIC BOOKSHELF, 12. Jan. 2016, S. 103–110.

Artikel Referenzen

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

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

Online References

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

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

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