Inhaltsverzeichnis

1 Design & Konzept

In diesem Kapitel ...

1.0.1 Framework

Warum Framwork? Besonderheiten...

1.1 Übersicht der Komponenten im Klassendiagramm

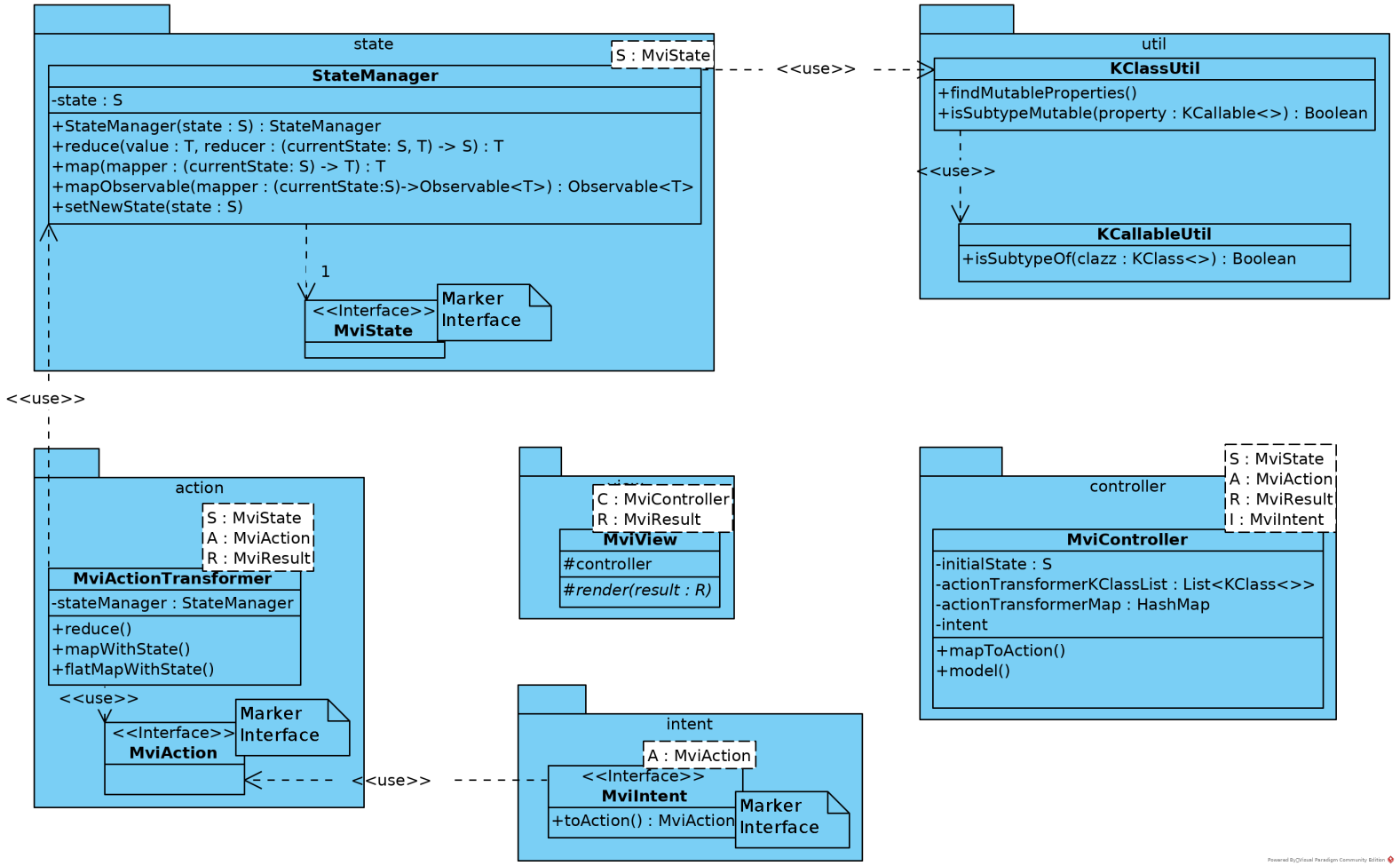

Abbildung 1: Komponenten im Klassendiagramm

1.2 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. Sollte diese nicht gegeben sein, so wird eine Fehlermeldung ausgegeben und der Prozess gestoppt.

Ist dies 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 wir ausschließlich intern im Framework genutzt. Es verhindert, dass der Zustand an einer beliebigen und nicht vorgesehenen Stelle verändert werden kann.

1.3 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 'mapToAction', 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 Essential sind.

1.4 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.

1.5 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 'ActionTransfomrer' zuteil, welcher über der ersten der zwei generischen Parameter, dem 'A', definiert wird. Dieser Parameter setzt voraus, dass das eingesetzte 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.

Der 'ActionTransformer' ermöglicht die Interaktion mit dem Zustand durch eine Variationen an Funktionen. Dafür benötigt er einen 'StateManager' welcher im 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 Kapitel 1.4 präsentierten 'Observable' Klasse statt und legt diese auch 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.

1.6 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 'ActionTransformer' 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 Klassenamen 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.

1.7 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.

1.8 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 Anleitung 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 'ActionTransformer' Klasse
6. Implementierung des Controllers auf Basis der 'MviController' Klasse
7. Implementierung der View auf Basis der 'MviView' Klasse

2 Prototypische Implementierung

In diesem Kapitel ...

2.1 Grundlegende Entscheidungen

Bevor auf Entscheidungen eingegangen wird ...

2.1.1 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.

2.1.2 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
- Erweiterungsfunktionen
- Kovarianz und Kontravarianz werden explizit anwendet

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

- Standardwerte für Parameter

Listing 1: 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 1 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 bswp. Listen zum Einsatz kommen.

2.2 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' die Kotlin agnostisch ist: RxKotlin. Anhand Listing 2 soll der Umgang mit dieser Bibliothek in einem für den weiteren Verlaufen relevanten Umfang dargelegt werden.

Listing 2: '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)
```

2.3 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 9 aufzeigt, gestaltet das den Umgang mit unveränderlichen Klassen verhältnismäßig einfach.

Listing 3: data class

```

data class MyClass(val text: String, val anotherText: String)

val myClass = MyClass("text", "anotherText")

// 'text' wird verändert, 'anotherText' nicht
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.

2.3.1 Reflexion (Introspektion)

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

1. Debugger
2. Interpreter
3. 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 4: Java Reflexion

```

val myClass: Class<MyClass> = MyClass::class.java

// gibt Methoden wie 'toString' aus
myClass.methods.forEach(::println)
```

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

Listing 5: Kotlin Reflexion

```

val myClass: KClass<MyClass> = MyClass::class

// gibt den Namen der Klasse aus
println(myClass.qualifiedName)

// 'false', da 'text' nicht Konstant ist
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 6: Override Annotation

```

class MyClassJava {

@Override
public String toString() {...}
}

class MyClassKotlin {

// 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.

2.3.2 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 7: 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 8: 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.

2.3.3 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 9: Kotlin 'isData' Attribut

```
1 data class DataClass
2
3 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 10).

Listing 10: 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 11 zeigt beide Anwendungsfälle.

Listing 11: 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 von jedem Attributs daraufhin untersucht. Entspricht es dem Suchmuster, so wird es - entsprechend Listing 12 - aus der Liste gelöscht.

Listing 12: Filtern

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

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.

Die beide Funktionen 'findMutableProperties' und 'isSubtypeMutable' werden gemeinsam in der Klasse 'KClassUtil' untergebracht.

2.3.4 StateManager

Die oben aufgeführte Implementierung für die Überprüfung der Klasse für den Zustand findet ihren Platz in der in der in Kapitel 1.2 vorgestellten 'StateManager' Klasse.

Bevor die Klasse implementiert werden kann, muss das 'MviState' Marker Interface wie in Listing 13 definiert werden.

Listing 13: 'MviState' Interface

```
interface MviState
```

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

Listing 14: '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.

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 15

Listing 15: 'setNewState' Methode

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

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

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 16 macht dies Sichtbar.

Listing 16: 'reduce' Methode

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

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 17 beweist.

Listing 17: 'map' Methode

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

Damit erfüllt diese Komponente alle Anforderung und ist vollends implementiert.

2.4 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 2.3.4 wird ein generischer Typ mit einem 'Upper Bound' benutzt. In diesem Fall wird 'MviAction' angegeben.

Listing 18: 'Marker' Interfaces

```
interface MviAction

interface MviIntent<A : MviAction> {

 fun toAction(): A
}

interface MviResult
```

2.5 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 19: 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 20 demonstriert, wie es die beschriebenen Anforderungen erfüllt.

Listing 20: 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 21

Listing 21: 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 22 herhalten:

Listing 22: 1. Versuch

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

 return Observable.fromCallable {
  stateManager.reduce { currentState ->
```

```
8       reducer(currentState)
9     }
10    value
11  }
12 }
```

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 23: 2. Versuch

```
1  protected fun <T : Any> reduceState(
2   reducer: (currentState: S) -> S) =
3
4   ObservableTransformer<T, T> { upstream: Observable<T> ->
5    upstream.map { value ->
6     stateManager.reduce { currentState ->
7      reducer(currentState)
8     }
9
10    value
11   }
12 }
```

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

Listing 24: 2. Versuch

```
1 upstream.map { TestMviResult.TestResult }
2 // 1
3 .flatMap { reduceState(it) { currentState ->
4   currentState
5 } }
6 // 2
7 .compose(reduceState { currentState ->
8  currentState
9 })
```

Ein andere Alternative stellt die Entwicklung eines Operators eigens für 'RxJava' dar, jedoch gestaltet sich dies als durchaus komplex. [1] 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 25 in der Methodendefinition platziert wird.

Listing 25: 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 26: 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 27 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 27: 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 28 effizient und elegant umsetzen.

Listing 28: '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)
  }
  .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 29 Beispielhafte Implementierung erzielen.

Listing 29: Beispiel Implementation

```
class SomeTransformer(
 stateManager: StateManager<SomeState>
) : MviActionTransformer<
 SomeState,
 SomeAction,
 SomeResult>(stateManager) {

 override fun apply(upstream: Observable<SomeAction>)
 : ObservableSource<SomeResult> =

 upstream
  .flatMapWithState { currentState: SomeState,
     data: SomeAction ->

    doSomethingWithState(currentState)
  }
  .reduceState { newValue: SomeNewValue,
       currentState: SomeState ->

```

```
20      currentState.copy(value = newValue)
21    }
22    .map { SomeResult }
23  }
```

2.6 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 30: '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 31 niedergeschrieben werden.

Listing 31: 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 32

Listing 32: Konsruktor

```
1 val transformer = kClass
2     .primaryConstructor!!
3     .call(StateManager(initialState))
4      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 33 zeigt die durch die Beschreibung erhaltene Implementation.

Listing 33: 'MviAction' Name

```
1 val actionName = kClass
```

```
  .supertypes[0]
  .arguments[1]
  .type
  .toString()
```

Da es sich um eine Liste von 'MviActionTransfromer' handelt, muss dies für jedes Element 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 32 und 33 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 34 gibt ein Beispiel.

Listing 34: 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 35: 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 36: 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 37 das Endergebnis darstellt:

Listing 37: 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> =
```

```
12  flatMap {action ->
13   compose(actionTransformerMap[action::class.qualifiedName])
14  }
```

2.7 Intent

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

Listing 38: Intent Klasse

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

Für diesen Fall eignet sich eine Klasse mit einem generischen Typ als Attribut, wie Listing 38 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 38 behoben werden.

Listing 39: Intent Klasse

```
1 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 41 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 40: 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 40 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 41: 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 42 gehört unter anderem zu den Summen Typen.

Listing 42: 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 41 funktionaler und eleganter zum implementieren:

Listing 43: Intents als sealded class

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

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

mehrere Instanzen haben, die überdies einen Zustand enthalten kann. Zu beachten gilt außerdem, das in Kotlin sich dieses Konstrukt innerhalb einer Datei befinden muss.

Abbildungsverzeichnis

Online References

[1] 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).