

Inhaltsverzeichnis

1	Design & Konzept	1
1.1	Grundlegende Designentscheidungen	1
1.1.1	Android als Plattform	1
1.1.2	Framework	1
1.1.3	Kotlin als Programmiersprache	1
1.2	Intent	3
1.3	View Model	6

1 Design & Konzept

In diesem Kapitel ...

1.1 Grundlegende Designentscheidungen

Bevor auf Entscheidungen eingegangen wird ...

1.1.1 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 es sich im Fall Android um einen Nachzügler handelt.

1.1.2 Framework

Warum Framework? Besonderheiten...

1.1.3 Kotlin als Programmiersprache

Die Applikationen in Android und das Android-SDK selbst sind bis vor wenigen Jahren fast ausschließlich in der Sprache Java entwickelt wurden. Seit der Google I/O 2017 gehört jedoch eine weitere Sprache zu den offiziell unterstützen: Kotlin. Sie wird von dem Unternehmen JetBrains entwickelt, die unter anderem die Entwicklungsumgebung IntelliJ für Java produzieren. Dieses bildet auch die Grundlage für Android Studio. Kotlin hat in den letzten Jahren an Bodenhaftung gewonnen und findet auch intern bei Google Verwendung.

Die Sprache wird als statisch typisierte, objektorientierte Programmiersprache bezeichnet und verfügt über eine hohe Interoperabilität zu Java. Dies bedeutet, dass innerhalb eines in Java geschriebenen Programms ohne viel Aufwand Kotlin genutzt werden kann. Dies ist ein wichtiger Faktor für die immer weiter ansteigende Beliebtheit, da es eine einfache Integration und bisherige Projekte gestattet. Kotlin bringt 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-Funktionen und bilden eine funktionale Grundlage
- Datenklassen machen den Umgang mit unveränderlichen Datenstrukturen einfach

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

Listing 1: Kotlin Beispiel

```

1 data class Example(
2     privat val defaultMessage: String = "Hello World"
3     privat val maybeNull: String? = null
4 ){
5
6     // Expression
7     fun isHelloWorld() = when(message){
8         "Hello World" -> true
9         else -> false
10    }
11
12    // "?" findet bei null verwendung
13    fun printIfNotNull() {
14        maybeNull?.run {
15            print(this)
16        }
17    }
18 }
19
20 // Erweiterungsfunktion und Funktion als Parameter
21 fun HelloWorld.extensionFunction(function: () -> String) {
22     val message = function()
23     println(message)
24 }
25
26 // kein new Schlüsselwort nötig
27 // keine Semikolon nötig
28 val example = Example()
29
30 // copy wird automatisch generiert bei einer "data" Klasse
31 val newExample = example.copy(defaultMessage = "New Message")
32
33 // ist der letzte Parameter eine Funktion, so kann auf Klammern
34 // verzichtet werden
35 newExample.extensionFunction { "Hello" }

```

Insgesamt ist anhand Listing 1 zu erkennen, dass Kotlin eine deutlich prägnantere und schlankere Syntax besitzt. Sie vermeidet damit einen großen Teil des mit Java verbundenen "Boilerplate-Codes" und kann für einen 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.

1.2 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 einer 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 2: Intent Klasse

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

Für diesen Fall eignet sich eine Klasse mit einem generischen Typ als Attribut, wie Listing 2 zeigt. Das «T» in der Klassen Deklaration dient dabei als Platzhalter für den eigentlichen Typ, z.B. für ein "Item" aus dem obigen Beispiel.

Die aufgezeigte Option weist 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 2 beheben werden.

Listing 3: Intent Klasse

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

Hierfür muss lediglich von Kotlin's Notation für null Typen Gebrauch gemacht werden: Das Fragezeichen. Um zu vermeiden, dass bei dem Erstellen einer Klasse ohne Inhalt stets null übergeben werden muss, wird ein Standardparameter verwendet. Listing 5 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 4: Intent Enum

```
1 enum class IntentDescription {  
2     GO_BACK, DISPLAY_ITEM_DETAILS  
3 }
```

Bei Ansatz Nummer Eins ist ein potenzieller Kandidat die Verwendung eines "enum". Diese kann durch Konstanten (Listing 4 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 5: Intent Klasse

```
1 class Intent<T> (  
2     val payload: T? = null,  
3     val description: IntentDescription  
4 )  
5  
6 // die Explizite Angabe von Attributnamen  
7 // bei weglassen von anderen Attributen ist  
8 // best practice  
9  
10 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 Fällen vertretbar sein, wird bei einer hohen Anzahl an Intentionen unübersichtlich. Zusätzlich sollte immer versucht werden "null" Werten aus dem Weg zu gehen, wenn nicht zwingend erforderlich. Dies verringert die Gefahr einer "NullPointerException" über den Weg zu laufen.

Ein besserer 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 Objektorientierten Programmierung. Es sagt aus, dass mehrere unterschiedlichen Werte ein einzigen Wert bilden können. Darunter fällt bspw. eine Klasse in Java oder Kotlin.

Das zweite Konstrukt, die Summe von Werten liegt vor, wenn anstatt von mehreren Werten 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 6 gehört unter anderem zu den Summen Typen.

Listing 6: Summen Typ

```
1 enum class Color(val rgb: Int) {  
2     RED(0xFF0000),  
3     GREEN(0x00FF00),  
4     BLUE(0x0000FF)  
5 }  
6  
7 val color: Color = Color.RED  
8  
9 print(color is RED) // true  
10 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 "PP-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 5 funktionaler und eleganter zum implementieren:

Listing 7: Intents als sealed class

```
1 sealed class Intent {  
2     // 'object' erzeugt ein Singleton  
3     // es ist keine Instanziierung möglich  
4     // Attribute folglich nicht gestattet  
5     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, dass in Kotlin sich dieses Konstrukt innerhalb einer Datei befinden muss.

1.3 View Model

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

Im ersten Schritt muss eine zentrale Funktion definiert werden, die als Parameter einen Intent erhält. Diese muss vom Entwickler überschrieben und implementiert werden. Zu diesem Zweck wird vom Konzept der Vererbung und dem oft damit verbundenen Polymorphismus (griechisch für Vielgestaltigkeit) Gebrauch gemacht. Erbt eine Klasse von einer anderen, so wird sie zu einer sogenannten Subklasse, während die andere eine Superklasse (auch Eltern- oder Basisklasse) darstellt. Sinn und Zweck der Vererbung ist die Wiederverwendbarkeit von Code und somit die Vermeidung von Redundanz. Es ermöglicht Code für Klassen, die gewisse Merkmale teilen, nur einmal schreiben zu müssen und diese dann davon ableiten zu lassen. Die Polymorphie liegt vor, wenn zwei Klassen denselben Methodennamen verwenden, aber die Implementierung der Methoden sich unterscheidet. Listing 8 verdeutlicht diesen Sachverhalt.

Listing 8: Vererbung & Polymorphismus

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

```

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

```

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

```

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

```

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

Abbildungsverzeichnis