Inhaltsverzeichnis

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 Entwickelung von Webseiten fand und es sich im Fall Android um einen Nachzügler handelt.

1.1.2 Framework

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

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

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

Listing 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 HelloWorld.extensionFunction(function: () -> String) {
 val message = function()
 println(message)
}

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

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

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

Insgesamt is anhand Listing 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 NNullpointer-Exceptionretten. Des weiteren besteht ein größerer Fokus auf dem Einsatz von Konzepten aus der funktionalen Programmierung, welche durch Erweiterungsfunktionen für bswp. Listen zum Einsatz kommen.

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 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 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 eigentlich Typ, z.B. für ein ”Item”aus dem obigen Beispiel.
Die aufgezeigte Option weißt jedoch gewisse Mängel auf:

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

Mangel Nr. 1 lässt sich mit einer einfachen Abwandlung von Listing 2 behoben werden.

Listing 3: Intent Klasse

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

Hierfür muss lediglich von Kotlins Notation für nullTypen gebraucht gemacht werden: Das Fragezeichen. Um zu vermeiden, das bei dem erstellen einer Klasse ohne Inhalt stets null”übergeben werden muss, wird ein Standardparameter verwendet. Listing 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

```
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 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

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

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

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

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

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

Das zweite Konstrukt, die Summe von Werten liegt vor, wenn anstatt von mehreren Werte zu einem entweder ein Typ oder ein anderer vorliegt. Es ist somit keine Kombination von Werten wie beim Produkt, sondern die Entscheidung für einen der Angegebenen. Das Enum wie in Listing 6 gehört unter anderem zu den Summen Typen.

Listing 6: Summen Typ

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

val color: Color = Color.RED

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

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

In Kotlin existiert für diesen speziellen Fall eine bestimmte Form der Klasse, die ihn ihrer Deklaration mit dem ”sealed”Schlüsselwort eingeleitet wird. Dies macht es möglich, Listing 5 funktionaler und eleganter zum implementieren:

Listing 7: 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.

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