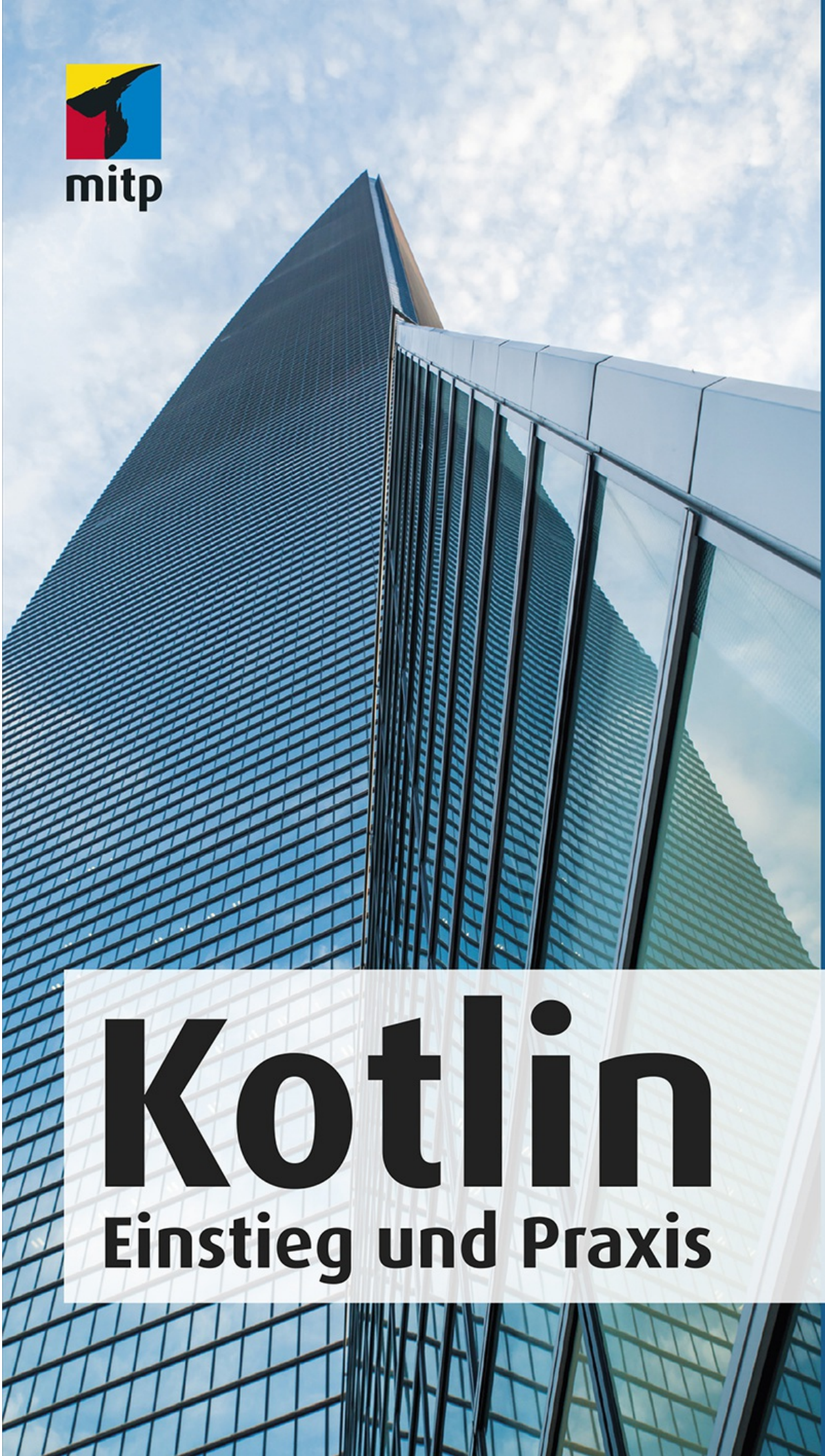

mitp

Karl
Szwillus

Kotlin

Einstieg und Praxis

1.2 Kotlin und Android

Hier kann man guten Gewissens sagen, dass Android zu dem Zeitpunkt, als die Entscheidung fiel, Kotlin als möglichen Ersatz in ersten Demos zu präsentieren, ein besonders dankbarer Kandidat für weitreichende Verbesserungen war.

Das lässt sich gut am Beispiel der Android *Activity* nachvollziehen. Diese Klasse ist ein sehr komplexes Stück Code, sie modelliert die Repräsentation eines »Screens« auf dem Smartphone und stellt den zentralen Anbindungspunkt für das Verhalten einer App dar. Es handelt sich um eine Klasse von 8.000 Zeilen mit etwa 40 Attributen, hinter denen zum Teil wieder komplexe Strukturen stecken. Teilweise setzen sie auch Nebenläufigkeit voraus oder werden durch das System regelmäßig ausgewertet oder beschrieben.

Und auch, wenn diese Klasse selbst in den Apps lediglich benutzt wurde, so schaffte sie mit ihren umfangreichen Schnittstellen die Grundlage für toxische Entwicklungsmuster.

In den ersten Versionen von Android gab es darüber hinaus wenig Vorgaben, nach welchen Mustern der Quellcode einer App zu strukturieren ist. Der übliche Aufbau begann mit der Initialisierung der App-Zustände in der Methode `public void onCreate(Bundle savedInstanceState)` in einer von `Activity` abgeleiteten Klasse. Diese Methode, vom Android-System aufgerufen, war und ist der kanonische Startpunkt einer App. Auch die Handhabung weiterer Komponenten folgte dann in dieser Klasse, herausgezogen lediglich die Views und ihre Backing Objects wie Adapter und Datenbanken.

Die gesamte Komplexität hing damit von Anfang an in jeder einzelnen Komponente einer App, die auf dieser Basis entwickelt wurde. Aufgrund der Funktionsfülle hatten auch die abgeleiteten Klassen eine Tendenz zu unkontrolliertem und schlecht kontrollierbarem Wachstum. Da nehme ich mich persönlich gar nicht aus.

Um nun eine einzelne Methode dieser Klasse sinnvoll per Unit-Test validieren zu können, bedurfte es nicht nur einer Instanz der entsprechenden `Activity`, sondern mindestens noch eines Aufrufs der Methode `onCreate()`. Im Normalfall benötigte dieser Aufruf eine komplette Systemumgebung (vulgo Smartphone oder Simulator), da diese Methode in der Vererbungshierarchie auf verschiedene Komponenten zugreift, die im Normalfall auf einem eingeschalteten Smartphone zur Verfügung stehen. In einem Unit-Test, dem kein Android-Gerät zur Verfügung steht, zog dies entweder aufwändiges Mocking von Komponenten nach sich – oder den Verzicht auf Unit-Tests.

Das Ergebnis waren zahlreiche ungetestete bis schlecht getestete Apps und damit eine deutlich gefühlte Unterlegenheit gegenüber Umgebungen[1], in denen die gute Testbarkeit des Codes eine lange geübte Selbstverständlichkeit darstellte.

Zunächst sind an dieser Stelle Anbieter von bekannten Apps eingesprungen, die durch das

Sponsoring von Open-Source-Entwicklungen für Abhilfe gesorgt haben. Sie führten Techniken und Strukturen wie die Architekturmodelle *MVP* und *MVVM* für Android ein und vereinfachten die Organisation von Abhängigkeiten mit *Dependency Injection* oder auch konkrete Aufgaben wie Bilderhandling und Netzwerk-Kommunikation. Mit den entsprechenden Libraries von Square, Facebook und anderen hat wohl jeder Android-Entwickler mehr als einen kurzen Flirt verbracht. Immerhin sorgten sie für strukturierteren Code.

Google verbesserte Android jedoch in den letzten Jahren durch neue Architekturansätze und leichtere Komponenten auf dieser Seite stark und machte viele der oben genannten Komponenten zu Teilen des Frameworks. Das ist aber nur eine Seite der Medaille.

Denn der Code, der auf diesen komplexen Bestandteilen aufsetzt, soll sich auf seine eigenen Qualitäten und Aufgaben fokussieren. Wenn das mobile Betriebssystem schon so große Teile der Aufgaben abnimmt, dann kann der eigene Code möglichst reduziert und möglichst präzise sein. In der Zwischenzeit hat sich der Blick dafür auf das Paradigma der *funktionalen Entwicklung* gerichtet.

1.3 Vorteile von Kotlin

Auf ganz grundsätzliche Art können die Sprachkonstrukte aus der funktionalen Entwicklung die Testbarkeit erleichtern oder sogar forcieren. Und diese Aspekte führt Kotlin als grundlegenden Bestandteil seiner DNA mit sich.

Eine »reine« (*pure function*) Funktion, die von außen alle nötigen Daten zur Bearbeitung als Parameter erhält und dann ein Ergebnis zurückgibt, ohne Daten an anderer Stelle zu manipulieren, kann leicht getestet werden. Je mehr Querbeziehungen zu anderen Daten oder Objekten bestehen oder wenn komplexere Systemzustände in die Ausführung der Funktion hineinspielen, desto schwieriger wird es, den einen Fall zu isolieren. Wenn sich aber der eigene Code darauf verlassen kann, dass die standardisierten Bibliotheken gut getestet und vollständig sind, dann lohnt es sich, auch den eigenen Use-Case möglichst präzise aufzuschreiben.

Die Arbeit mit kurzen Funktionen, unveränderlichen Datenstrukturen und höheren Funktionen führt dazu, dass die eigenen Code-Bestandteile nicht nur kürzer, sondern auch besser testbar werden.

Statische Typisierung und Null-Sicherheit stehen ebenfalls auf der Seite des Forcierens von besserer Wartbarkeit, weil sie Entwicklern die Mittel für den laxen Umgang mit seinen Variablen nehmen. Gerade bei einem komplexen System wie dem Smartphone lag und liegt auch hier großes Potenzial zur Verbesserung.

Wenn die oben vorgestellten Techniken aus der funktionalen Entwicklung und bessere Tests vor allem zur Entwicklungs- und Compilezeit wirken, so bedeuten statische Typen und Null-Sicherheit vor allem bessere Sicherheit zur Laufzeit. Sie bringen Klarheit über den Zustand einer Variable im Verlauf ihrer Benutzung, auch wenn der Code im Einsatz ist. Das gilt umso mehr, wenn der Umstieg von einer dynamisch typisierten Sprache wie JavaScript passiert.

Das jedoch funktioniert nur dann gut, wenn vorhersagbar ist, welchen Wert eine Variable zu einem Zeitpunkt annehmen kann. Veränderliche Datenstrukturen bedeuten bei jeder Zuweisung Verantwortung des Entwicklers für die Zulässigkeit der Operation. In einem System mit unveränderlichen Strukturen kann diese Verantwortung an den Compiler übertragen werden.

Eine Funktion, deren Rückgabewert durch einen Test validiert wird und deren Ergebnis in eine neue Variable geschrieben wird, bedeutet eine testbare Kette von Operationen.

Davon können auch Apps profitieren, denn sämtlicher Quellcode, der auf das Framework Android aufbaut, liegt im Zuständigkeitsbereich des Entwicklers. Eine gute objektorientierte Modellierung, die konsequent Vererbung und Delegation nutzt und intern möglichst auf unveränderliche Datenstrukturen und einzelne Funktionen setzt, kann als aktueller Goldstandard in der App-Entwicklung gelten.

1.4 Ziele der Entwicklung von Kotlin

1.4.1 Übersichtlichkeit, präziser Ausdruck

> *The best code is no code at all.*
>
> – The Internet

Ein Hauptziel der Entwicklung von Kotlin war die Reduktion von Boilerplate-Code. Darunter versteht man Standard-Code oder auch überflüssige Zeichen, die man immer wieder aufschreibt, um grundlegende Funktionen zu ermöglichen. An sich nicht direkt schädlich, führt dieser Überfluss oft dazu, dass die eigentliche Funktion nicht im Mittelpunkt steht, sondern zunächst aus dem Standardcode herausgeschält werden muss.

Kotlin benötigt weniger Zeichen, um komplexe Zusammenhänge aufzuschreiben.

Es gibt verschiedene Strategien, um dieses Ziel zu erreichen, zum Beispiel die Verwendung von Konventionen, Bündelung von Funktionen in Bibliotheken oder das Weglassen von redundanten Zeichen. In Kotlin finden beinahe alle Prinzipien für diese Reduktion

Anwendung, zum Teil als optionale Möglichkeiten und zum Teil verbindlich.

Dazu gehören unter anderem:

- Vereinfachungsregeln für zahlreiche Aufrufe und Kurzformen
- Einzeilige Funktionen zur Rückgabe eines Wertes
- Datenklassen, deren Quellcode auf eine Zeile reduziert werden kann
- Kurzschreibweisen mit Lambdas und Standardfunktionen
- Extension-Funktionen statt Helper/Utility-Klassen
- Mapping-Funktionen der Collection-Klassen

Neben den Veränderungen im Quellcode der Anwendung unterstützen darüber hinaus die Sprachfeatures wie Null-Sicherheit und statische Typen die Reduktion des Codes, der zur Behandlung von Laufzeitproblemen bislang notwendig war.

Fairerweise sollte hier auch erwähnt werden, dass manche Konstrukte der funktionalen Programmierung den Quellcode auch verlängern können. Bei ganz einfachen Objektzuständen wird das deutlich. Als Beispiel soll die Modellierung einer einfachen Checkbox dienen, die lediglich zwischen »an« und »aus« unterscheidet. In der klassischen Objektorientierung legt man diesen Schalter als eine Klasse an, die den aktuellen Schaltzustand als Datenfeld hält und eine Methode `toggle` zum Anschalten oder Abschalten anbietet.

```
1  class Switch {
2     var isActive: Boolean = false
3
4     fun toggle() {
5        isActive = !isActive
6        paint()
7     }
8  }
```

Listing 1.1: Objektorientiertes Einschalten

Die Funktion `toggle` ist damit jedoch nicht unabhängig von Daten. Der Aufruf erfolgt ohne Parameter. Das bedeutet, die Funktion manipuliert Daten (den Schaltzustand) und arbeitet daher mit Seiteneffekten, statt mit Input und Output. Das ist in der Benutzung zunächst einmal praktisch und nach objektorientiertem Muster auch sauber und durchaus nicht untestbar.

Der Test jedoch erfordert die Erzeugung eines Objekts. Das ist immer dann einfach, wenn

ein Objekt für sich steht und keine weiteren Querbeziehungen hat. Man stelle sich aber vor, diese Checkbox benötigt wiederum einen Android-Kontext, wie es bei allen View-Objekten aus dem Android-Framework üblich ist.

Dann kann der Test auf die eigentlich simple `toggle`-Funktion nur noch implementiert werden, indem entweder der komplette Kontext von Android simuliert (*mocked*) wird, oder der Test direkt auf einem Android-Telefon ausgeführt wird.

In einer funktionalen Welt wäre das nicht gegeben. Die Checkbox wäre modelliert als eine Variable, zum Beispiel auch ein Objekt, das den Wert kapselt. Dazu benötigt es eine Funktion, der dieses Objekt übergeben werden kann, welches dann den Zustand ausliest und ein neues Objekt zurückgibt, das den gegenteiligen Zustand hat.

```
 1  fun toggleSwitch(switch: Boolean) {
 2     // Neuen Schalter mit der Negation des Werts von switch erzeugen
 3      return Boolean(!switch)
 4  }
 5  fun paintSwitch(switch: Boolean) {
 6  }
 7
 8  val switch = false
 9  val switchedOn = toggleSwitch(switch)
10  paintSwitch(switchedOn)
```

Listing 1.2: Funktionales Einschalten

In dieser Variante werden die Daten nicht manipuliert. Der abgeschaltete Schalter und sein aktiviertes Pendant sind technisch gesehen zwei verschiedene Variablen. Sie sind auch nicht abhängig von anderen Aufrufen.

Solange die Umsetzung von `toggleSwitch` nach einem Interface für schaltbare Objekte geschieht, wäre es darüber hinaus auch einfach, die Funktion zu erweitern und trotzdem eine gute testbare, funktionale Umsetzung zu haben. Was damit allerdings verloren geht, ist der Zusammenhang, den die Klasse Switch vorher geleistet hat.

Natürlich ist das ein Beispiel. Doch genau diese Überlegungen finden in vielen Softwareprojekten statt. In der funktionalen Programmierung sucht man immer die Strategie, die Veränderung zu einer Funktion und ihre Daten zu einem Parameter zu machen und dann ein Ergebnis im neuen Zustand zurückzugeben, der im nächsten Schritt weiter verwendet werden kann.

Die gleiche Logik wird angewendet, wenn Aufgaben bei einer Umsetzung von MVP (*Model View Presenter)* in einer App in einzelne Schritte zerlegt werden. Die Trennung von Datenmodell, Logik und darstellender Schicht sorgt dafür, dass mehr und kleinere Funktionen mit einer gerichteten Abhängigkeit entstehen. Zum Testen wird nicht mehr das