Projektarbeit

Alexander Lange

28. Februar 2022

Zusammenfassung

Dieser Bericht erläutert die Umgestaltung der Prüfplan-Applikation der Fachhochschule Bielefeld. Es wurden dabei einige Änderungen sowohl im Frontend als auch im Backend vorgenommen. Ziel war es zum einen, die Nutzererfahrung durch eine schönere Applikation zu verbessern und zum andern die Weiterarbeit durch einen besser strukturierten und dokumentierten Code zu verbessern. Dabei wurde viel Wert auf eine gute Programmlogik und überschaubare Klassen und Methoden gelegt, sowie die Trennung von verschiedenen Aufgaben mit Hilfe des Model-View-ViewModel-Patterns.

Inhaltsverzeichnis

Einleitung

Die Digitalisierung der Welt schreitet voran. Das Smartphone ist mittlerweile zu einem Grundgegenstand eines jeden Menschen geworden. Tag für Tag werden neue Apps entwickelt, die einem das Leben einfacher machen sollen. Im Zuge dessen entwickelt die Fachhochschule Bielefeld eine eigene App, mit der sich jeder Student seinen eigenen Prüfungsplan zusammen basteln kann. Im Zentrum dabei steht eine Übersicht über alle anstehenden Prüfungen, von denen sich der Student die für Ihn relevanten auswählen kann, und bei bedarf werden diese auch automatisch mit dem Kalender synchronisiert.

Ziel dieser Arbeit war es, die bereits bestehende App zu verbessern. Zum einen sollte die Programmiersprache von Java auf Kotlin geändert werden, zum anderen sollte die Benutzerfreundlichkeit und die Weiterarbeit angenehmer gestaltet werden.

Damit eine App benutzerfreundlich ist, muss auf einige Sachen geachtet werden. Dazu zählen:

- Übersichtlichkeit
- Personalisierbarkeit
- Verständlichkeit

Um diese Punkte zu erfüllen, wurden die einzelnen Seiten neu aufgebaut und unnötige Sachen wurden entfernt. Zudem wurden einige neue Einstellungsmöglichkeiten hinzugefügt, mit denen der Nutzer die App für sich persönlich anpassen kann. Dazu zählen unter anderem verschiedene Farbthemen und ein Darkmode.

Um die Weiterarbeit zu erleichtern, wurde der Programmcode neu strukturiert.Es wurde das sogenannte Model-View-ViewModel-Pattern umgesetzt, wodurch verschiedene Anwendungsbereiche wie Datenzugriffe, App Logik und User Interface von einander abgekoppelt werden, und es wurde eine umfangreiche Dokumentation erstellt.

In den folgenden Kapitel gibt es zuerst eine kleine Einführung in die Programmiersprache Kotlin, anschließen werden die Veränderungen im Backend und im Frontend näher erläutert. Zum Schluss gibt es noch ein paar kurze Anmerkungen und Regeln zur Weiterarbeit an dem Projekt, damit die App auch weiterhin gut strukturiert bleibt.

Kapitel 1

Kotlin

Kotlin ist eine Programmiersprache, die 2016 von dem Unternehmen Jetbrains in der Version 1.0 veröffentlicht wurde und 2017 von Google zur offiziellen Programmiersprache für Android erklärt wurde.[8] [5]

Genauso wie Java werden Kotlinprogramme in der JVM (Java-Virtual-Machine) ausgeführt. Dies sorgt für gute Kompatibilität dieser beiden Sprachen und macht so den Umstieg deutlich einfacher. Ein entscheidender Vorteil ist zum Beispiel, das Javaklassen in Kotlinklassen eingebunden werden können. Dies erspart viel Arbeit, da keine neuen Klassen programmiert werden müssen und viel mit dem alten Wissen weiter gemacht werden kann. Zusätzlich gibt es auch Tools, zum Beispiel von Android Studio, die es einem ermöglichen, bestehenden Javacode automatisch in Kotlincode umzuwandeln. Dies funktioniert zum einen für die Dateien im eigenen Projekt, aber auch mit einkopiertem Code aus externen Quellen.

1.1 Vorteile gegenüber Java

Kotlin bietet aber auch einige Vorteile gegenüber Java. Dazu gehören

- Nullpointer Sicherheit
- Weniger Codezeilen (dadurch allerdings erschwerte Lesbarkeit)

[2]. Die Nullpointer Sicherheit wird in Kapitel 1.2.2 näher erläutert. Die Codeverkürzung wird dadurch erzielt, das vieles unnötige weggelassen werden können. Dazu gehört zum Beispiel das Semikolon und die Getter-und Setter Methoden. Zudem gibt es viele Möglichkeiten zur Vereinfachung, wie zum Beispiel SAM aus Kapitel 1.10.2 oder die Nullüberprüfungen aus Kapitel 1.2.2.

1.2 Variablen

Kotlin ist eine statisch-typisierte Programmiersprache. Das bedeutet, das der Datentyp von Variablen im Gegensatz zu dynamisch-typisierten Sprachen nicht der Laufzeit, sondern bereits bei der Kompilierung festgelegt wird. Daher muss bei der Deklarierung von Variablen der Datentyp festgelegt werden wie es in Kapitel 1.2.1 erklärt ist, aber auch Verkürzungstechniken wie

1.2.1 Definition

Variablen in Kotlin werden über die Schlüsselwörter *var* und *val* definiert. Dabei steht *var* für eine Variable, der zur Laufzeit neue Werte zugewiesen werden können. Variablen die mit *val* definiert wurden, erhalten nur einmal bei der Erzeugung einen Wert und können danach nur noch gelesen werden. Der Code-Ausschnitt in Abbildung 1.1 zeigt verschiedene Deklarations-und Initialisierungsmöglichkeiten von Variablen in Kotlin. Datentypen werden, wie in Zeile 1,2 zu sehen, mit der Erweiterung *:Typ* festgelegt. Wenn der Variable bei der Deklaration sofort ein Wert zugewiesen wird, kann die Typisierung auch weggelassen werden, da der Compiler aus der Zuweisung den Datentyp automatisch bestimmt.

```
1    var a:Int = 3
2    val  b = 2
3    a = 2
```

Codebeispiel 1.1: Variablen

1.2.2 Null-Sicherheit

Eine Variable kann als Null-Sicher definiert werden. Das bedeutet, dass diese Variable den Wert *null* annehmen kann, ohne das beim Zugriff eine Nullpointer-Exception geworfen wird. Um dies zu ermöglichen, wird hinter den Datentyp ein Fragezeichen angehängt, wie es im Codeausschnitt 1.2 zu sehen ist. Der Zugriff auf eine Nullable-Variable muss allerdings immer mit einer Nullüberprüfung stattfinden. Dafür gibt es verschiedene Möglichkeiten. Die einfachste Möglichkeit ist in Zeile 4 zu sehen. Durch den Operator *?.* wird eine Nullüberprüfung durchgeführt und die Methode *Split()* wird nur ausgeführt, wenn *a* einen Wert besitzt. Sollte mit dem Ergebnis der Methode weiter gearbeitet werden, so muss für jede zusammenhängende Methode ebenfalls eine Nullüberprüfung durchgeführt werden, wie es in Zeile 6 zu sehen ist. Mit dem Operator *?:* wird in dem Fall eine Alternative angegeben, sollte die Variable *b* null enthalten. Mit dem *!!*-Operator in Zeile 5 wird die Nullable-Variable *a* für den Ausdruck in eine Not-Nullable-Variable umgewandelt. Wenn dies getan wird, muss allerdings vorher sichergestellt werden, das die Variable zu keiner Zeit den Wert *null* annehmen kann oder es muss eine manuelle Nullüberprüfung durchgeführt werden. Für den Fall, das eine Methode nur ausgeführt werden soll, wenn der Eingabeparameter nicht *null* ist, gibt es die *let*-Operation aus Zeile 7. Die Methode in den geschweiften Klammern wird nur ausgeführt wenn die aufrufende Variable einen Wert besitzt. Auf die Variable wird dann in den Klammern mit dem Schlüsselwort *it* zugegriffen.

```
1    var a:String? = null
2    print(a)
3    a = "Hello World!"
4    val b = a?.Split(" ")
5    val c = a!!.Split(" ")
6    print(b?:"Leeres Array")
7    b?.let{print(it)}
```

Codebeispiel 1.2: Umgang mit Nullable-Variablen

1.3 Methoden

Methoden in Kotlin, werden mit dem Schlüsselwort *fun* generiert. Der folgende Methodenname ist frei wählbar. In den Klammern werden Parameter festgelegt und nach den Klammern kann mit dem Ausdruck *:Datentyp* ein Rückgabe-Datentyp für die Methode festgelegt werden. Im Beispiel 1.3 sind ein paar Varianten aufgeführt. In Zeile 1 ist eine Methode ohne Rückgabewert und ohne Parameter zu sehen. In Zeile 10 ist eine Methode zu sehen, die zwei mögliche Eingabeparameter und einen Rückgabeparameter besitzt. Der zweite Eingabeparameter kann allerdings auch weggelassen werden da für ihn ein Default-Wert festgelegt wurde. In Zeile 15 ist eine Methode mit einer variablen Anzahl an Eingabeparameter definiert. Das heißt der Methode können beliebig viele Werte von einem Datentyp übergeben werden und die Methode kann dann über diese Werte iterieren. Feste und variable Parameter können auch kombiniert werden, wie es in Zeile 23 der Fall ist.

```
1  fun main() {
2      val a = add(1,2)
3      val b = add(1)
4      print(a.toString())
5      val c = add(1,2,3,4,5)
6      print(b.toString())
7      addAndPrint(1,2)
8  }
9
10 fun add(wert1:Int,wert2:Int=0):Int{
11      val sum = wert1+wert2
12      return sum
13 }
14
15 fun add(vararg werte:Int):Int{
16      var sum = 0
17      for(num in werte){
18        sum += num
19      }
20      return sum
21 }
22
23 fun add2(wert1:Int,vararg weitere:Int):Int{
24      var sum = wert1
25      for(zahl in weitere){
26        sum += zahl
27      }
28      return sum
29 }
```

Codebeispiel 1.3: Methoden

1.4 Datenstrukturen

Wenn mehrere Daten in einer Liste abgespeichert werden sollen, so wird eine Datenstruktur benötigt. In dieser sind beliebig viele Daten eines Datentypes miteinander verkettet und können über einen Index abgerufen werden. Im folgenden sind drei wichtige Datenstrukturen erläutert.

- Array

- Set
- Map
- List

1.4.1 Array

Ein Array ist eine statische Datenstruktur, das heißt es ist in seiner Größe unveränderlich. Erzeugt werden kann ein Array über die Methode *arrayOf(vararg values:T)* wobei als Parameter die zu füllenden Werte übergeben werden. Soll das Array zu beginn keine Werte enthalten, so lässt es sich mittels der Methode *arrayOfNulls<T>(size:Int)* erzeugen. Dabei muss mit *T* der Datentyp und *size* die Größe des Arrays festgelegt werden. Der resultierende Datentyp des Array ist dann allerdings unabhängig der Vorgabe Nullable (Array<T?>). Für einige Basisklassen wie

- Int
- Long
- Float
- Double
- Boolean

sind bereits Arrays vorhanden, diese können dann mit Methoden wie *intArrayOf(vararg elements:Int)* erzeugt werden. Um auf ein Element des Arrays zuzugreifen, wird entweder der **[]**-Operator oder die *.get(index:Int)*-Methode verwendet[7]. Sollen mehrere Elemente abgerufen werden, so kann über das Array mit einer For-Schleife iteriert werden. Diese wird in Abschnitt 1.5 genauer erläutert. Der Codeauschnitt 1.4 zeigt ein Beispiel für den Umgang mit Arrays.

```
1  val a = intArrayOf(1,2,3,4,5)
2      val b = arrayOf<String>("eins","zwei","drei")
3      val c:Array<Double?> = arrayOfNulls(3)
4      c[0] = 1.0
5      c[1] = 1.1
6      c[2] = 1.2
7      for(i in a){
8          print(i)
9      }
10     for(i in b){
11         print(i)
12     }
13     for(i in c){
14         print(i)
15     }
```

Codebeispiel 1.4: ArrayList

1.4.2 ArrayList

Wenn ein veränderbares Array benötigt wird, kann eine *ArrayList* verwendet werden. Die Erzeugung und Abfrage einer *ArrayList* verläuft gleich wie ein normales Array, allerdings ist es möglich, mit Hilfe von Methoden wie

- add
- addAll
- remove

das Array zu manipulieren, also Elemente hinzufügen und löschen, ohne auf eine feste Größe angewiesen zu sein.

1.4.3 Set

Ein Set ist ebenfalls eine Struktur von Daten. Die Besonderheit ist allerdings, das in einem Set jedes Element höchstens einmal existiert. Auf Grund dessen sind mit Sets mathematische Mengenoperationen wie zum Beispiel

- Schnittmenge - intersect()
- Vereinigungsmenge - union()
- Differenzmenge - minus()

möglich[7]. Man unterscheidet bei Sets zwischen *mutable*-und *not-mutable* Sets. Ein *mutableSet* ist von der Größe veränderbar, ein *not-mutable*-Set nicht. Erzeugt werden Sets genau wie Arrays mit Methoden wie *setOf()* beziehungsweise *mutableSetOf()*.

1.4.4 Map

In einer map werden Schlüsselwertpaare abgespeichert. Das heißt zu einem Schlüssel eines Datentyps wird ein Wert eines beliebigen Datentyps zugeordnet. Der Schlüssel ist dabei eindeutig, das heißt er existiert nur einmal. Der zugehörige Wert kann auch mehrfach in der Map vorkommen. Die Erzeugung funktioniert mit der Methode *mapOf()* beziehungsweise *mutableMapOf()* wobei als Parameter Schlüssel-Wert-Paare der Form *(schlüssel to wert)* übergeben werden. Die Abfrage verläuft über den Schlüssel, welcher als Index verwendet wird. Ein Beispiel für die Erzeugung und Abfrage ist im Codeausschnitt 1.5 beispielhaft gezeigt.

```
1 val a:Map<String,Int> = mapOf("1" to 1,"2" to 2,"3" to 3)
2 print(a["1"])
3 print(a["2"])
4 print(a["3"])
```

Codebeispiel 1.5: Map

1.4.5 List

Die Liste wurde in der App am häufigsten verwendet.Es handelt sich dabei um eine simple Liste von Daten, ähnlich wie die ArrayList. Auch hier wird zwischen *MutableList* und *List* unterschieden.

1.5 Schleifen

Um mehrere Elemente einer Datenstruktur abzufragen, wird eine Schleife benötigt. In Kotlin gibt es viele Möglichkeiten, eine Schleife zu implementieren. Die wichtigsten sind in den folgenden Abschnitten aufgeführt.

1.5.1 While

Eine While-Schleife durchläuft einen Codeabschnitt so lange, bis eine bestimmte Bedingung nicht mehr erfüllt wird, oder manuell aus der Schleife ausgetreten wird. Ein Beispiel für eine While-Schleife ist im Codeausschnitt 1.6 zu sehen.

```
1 var zaehler = 0
2 while(zaehler<10)
3 {
4     print(zaehler)
5     zaehler++
6 }
```

Codebeispiel 1.6: While-Schleife

Mit dem Schlüsselwort *break* kann die Schleife schon vorzeitig beendet werden, mit dem Schlüsselwort *continue* wird sofort mit der nächsten Iteration fortgefahren. Dies gilt auch für die anderen Schleifen.

1.5.2 For

Mit einer For-Schleife wird über eine Sammlung von Daten iteriert. Dies kann zum Beispiel ein Zahlenbereich sein, es kann sich aber auch um eine Datenstruktur irgendeines Datentyps handeln. Der Codeausschnitt 1.7 zeigt eine For-Schleife über die Zahlen von 0 bis 9. Der Zahlenbereich wird mit dem Schlüsselwort *until* aufgebaut, wobei die 10 in diesem Fall nicht mit eingeschlossen ist. Die Variable i ist in dem Fall die Iterationsvariable, sie nimmt nacheinander jeden Wert des Zahlenbereichs an.

```
1 for(i in 0 until 10){
2         print(i)
3     }
```

Codebeispiel 1.7: For-Schleife über einen Zahlenbereich

Der Codeausschnitt 1.8 zeigt eine Iteration über eine Datenstruktur. Die Variable item nimmt in diesem Fall alle Werte innerhalb der Datenstruktur nacheinander an.

```
1 val a = arrayOf("Eins","Zwei","Drei","Vier","Fuenf")
2 for(item in a){
3     print(item)
4 }
```

Codebeispiel 1.8: For-Schleife über eine Datenstruktur

1.5.3 Foreach

Die *forEach*-Schleife ist eine Alternative zur For-Schleife. Sie wird von einer Datenstruktur als Methode aufgerufen und führt einen bestimmten Codeblock für jedes Element der Datenstruktur aus. Auf das Element aus der Datenstruktur kann wie im Beispiel 1.9 in Zeile 3 zu sehen über das Schlüsselwort *it* oder wie in Zeile 2 über eine selbst benannte Variable zugegriffen werden.

```
1 val a = arrayOf("Eins","Zwei","Drei","Vier","Fuenf")
2     a.forEach{i -> print(i)}
3     a.forEach{print(it)}
```

Codebeispiel 1.9: ForEach-Schleife über eine Datenstruktur

1.6 Verzweigungen

Natürlich gibt es auch in Kotlin Verzweigungen, wo eine Bedingung geprüft wird, und je nach Ergebnis ein anderer Codepfad durchlaufen wird.

1.6.1 If-Verzweigung

Die einfachste Möglickeit ist die If-Verzweigung. Diese überprüft eine Bedingung auf Wahr(true) oder Falsch(false) und wählt je nach Ergebnis einen von zwei Pfaden. Ein Beispiel ist im Codeausschnitt 1.10 gezeigt. Eine Besonderheit in Kotlin ist, dass wie in Zeile 8 zu sehen, eine If-Verzweigung in einen Ausdruck mit eingebaut werden kann. Die Variable c nimmt dann abhängig der Variablen a und b einen anderen Wert an.

```
1 val a = 4
2 val b = 2
3 if(a<b){
4     print(a)
5     }else{
6     print(b)
7 }
8 val c = if(a<b) a else b
9 print(c)
```

Codebeispiel 1.10: If-Verzweigung

When

Eine Erweiterung der If-Verzweigung ist die When-Verzweigung. Diese ist die Kotlin alternative zu Javas Switch-Case und vergleicht einen Wert mit einer beliebigen Anzahl anderer Werte. Bei Gleichheit wird dann der entsprechende Code ausgeführt. Die Implementierung der When-Verzweigung ist im Codeausschnitt 1.11 an einem Beispiel gezeigt. Auf der linken Seite des Pfeiles -> steht jeweils der Wert, der mit dem Wert a verglichen werden soll, auf der rechten Seite steht der Code, der bei Gleichheit ausgeführt werden soll. Mit dem Schlüsselwort *else* wird beschrieben was passiert, wenn keiner der Fälle zu trifft. Dieser Zweig kann aber auch weggelassen werden. [7]

```
val a = 3
    when(a){
        1 -> print("eins")
        2 -> print("zwei")
        3 -> print("drei")
        4 -> print("vier")
        5 -> print("fuenf")
        else -> print("anderer Wert")
    }
```

Codebeispiel 1.11: When-Verzweigung

1.7 Klassen

Kotlin ist eine Objektorientierte Programmiersprache. Klassen, oder auch Objekte, spielen daher eine entscheiden Rolle bei der Programmierung.

1.7.1 Definition

Eine normale Kotlin-Klasse wird mit dem Schlüsselwort *class* erzeugt. Die Sichtbarkeit der Klasse ist durch das vorangestellte Schlüsselwort festgelegt.

- public - Auf die Klasse kann aus jeder anderen Klasse zugegriffen werden
- protected - Auf die Klasse kann nur innerhalb der Elternklasse, oder allen Klassen, die von der Elternklasse erben, zugegriffen werden
- private - Auf die Klasse kann nur von der Elternklasse zugegriffen werden

Innerhalb einer Klassen können verschiedene Parameter und Methoden definiert werden. Für beide kann die Sichtbarkeit ebenfalls eingeschränkt werden. Jeder Parameter und jede Methode ist per default *public*.

```
public class TestKlasse{

private var ersterParameter:String? = null

val zweiterParameter = 1

private fun ersteMethode(){
}

fun zweiteMethode(){
}
}
```

Codebeispiel 1.12: Einfache Klasse

1.7.2 Konstruktoren

Um von einer Klasse ein Objekt zu erzeugen, muss ein Konstruktor der Klasse aufgerufen werden. Wurde kein eigener Konstruktor definiert, so wird automatisch ein leerer Konstruktor ohne Parameter und ohne Code erzeugt. Wenn ein speziellerer Konstruktor benötigt wird, gibt es verschiedene Möglichkeiten, diesen zu erzeugen.

Primärkontruktor

Der Primärkonstruktor steht, wie in Beispiel 1.13 zu sehen, direkt hinter dem Klassennamen. Diese Parameter müssen dann allerdings direkt an globale Parameter der Klasse übergeben werden. Sie können selber nicht innerhalb von Klassenmethoden verwendet werden.[7]

```
1 class Test(a:String,b:Int){
2     private val a = a
3     private val b = b
4 }
```

Codebeispiel 1.13: Primärkonstruktor

Alternativ kann den Variablen im Primärkonstruktor, wie in Beispiel 1.14 zu sehen, eine Deklaration vorangestellt werden, wodurch der Parameter als globale Variable der Klasse gesehen wird.

```
1 class Test(private val a:String,var b:Int){
2
3 }
```

Codebeispiel 1.14: Primärkonstruktor mit integrierten Variablen

Initialisierungsblock

Wenn für die Zuweisung der Primärkonstruktorparameter eine komplexere Logik als die direkte Zuweisung benötigt wird, ist es möglich, die Zuweisungen in einen Initialisierungsblock zu verschieben. Dieser wird mit dem Schlüsselwort *init*, gefolgt von einem Codeblock implementiert. Ausschnitt 1.15 zeigt ein Beispiel für eine mögliche Implementierung.[7]

```
1  class Test(aInput:String,bInput:Int){
2
3      private var a:String
4      private var b:Int
5
6      init{
7          a = if(aInput=="")"unbekannt" else aInput
8          b = if(bInput<0) 0 else bInput
9      }
10 }
```

Codebeispiel 1.15: Initialisierungsblock

Sekundärkontruktoren

Wenn es für eine Klasse mehrere verschiedene Konstruktoren geben sollen, zum Beispiel weil manche Parameter auf Grund von Defaultwerten keine Initialisierung benötigen, so können Sekundärkonstruktoren verwendet werden. Ein Sekundärkonstruktor wird mit dem Schlüsselwort *constructor*,gefolgt von einer Parameterliste und einem Methodenblock definiert. Sekundärkonstruktoren können sich auch gegenseitig aufrufen, dafür wird der Methodenblock weggelassen, und hinter die Parameterliste der Ausdruck *:this(param1,param2 ...)* angehängt. Dabei werden für die Parameter (*param1,param2 ...*) die passenden Werte für

den aufgerufenen Konstruktor eingesetzt. Dies können zum Beispiel Parameter des aufrufenden Konstruktors oder Defaultwerte sein. Um eindeutig zu bleiben, muss jeder Sekundärkonstruktor eine Parameterliste, mit unterschiedlichen Datentypen besitzen. Dies ist in Beispiel 1.16 zu sehen.

```
class Test{

    private var a:String
    private var b:Int

    constructor(pa:String,pb:Int){
        a = if(pa=="")"unbekannt" else pa
        b = if(pb<0)0 else pb
    }

    constructor(pa:String):this(pa,0)
    constructor(pb:Int):this("",pb)
    constructor():this("",0)
}
```

Codebeispiel 1.16: Sekundärkonstruktoren

Soll ein Primärkonstruktor mit weiteren Sekundärkonstruktoren kombiniert werden, so muss jeder Sekundärkonstruktor den Primärkonstruktor aufrufen, entweder direkt oder über einen anderen Sekundärkonstruktor.[7]

1.7.3 Vererbung

Genau wie in anderen objektorientierten Programmiersprachen ist es auch in Kotlin mögich, dass Objekte von anderen Objekten erben. Dabei übernimmt die erbende Klasse (Kindklasse) alle öffentlichen (public) oder beschützten (protected) Attribute und Methoden der Basisklasse (Elternklasse). Damit von einer Elternklasse geerbt werden kann, muss diese mit dem Schlüsselwort *open* definiert worden sein. Zusätzlich muss jede Methode und jedes Attribut der Elternklasse mit dem Schlüsselwort *open* versehen werden, wenn die Kindklasse diese überschreiben können soll. Die Vererbung findet mit dem *:*-Operator statt. In Beispiel 1.17 erbt die Klasse B von der Klasse A und überschreibt sowohl das Attribut *text* als auch die Methode *print()*.

```
 open class A{
    open protected val text:String = "Hello World!"
    open fun print(){
        println(text)
    }
}

class B:A(){
    override val text = "World Hello!"
    override fun print(){
        println(text.reversed())
    }
}
```

Codebeispiel 1.17: Vererbung

1.8 Erweiterungsmethoden

Da es allerdings schnell unüberschaubar werden kann, wenn jedes mal, wo zum Beispiel eine neue Methode für die Elternklasse benötigt wird, eine neue Kindklasse implementiert werden muss, gibt es in Kotlin eine elegante Lösung. Mit so genannten *Erweiterungsmethoden* kann einer bereits bestehenden Klasse eine neue Methode hinzugefügt werden, ohne eine Kindklasse erstellen zu müssen. Das ist besonders nützlich für Klassen aus fremden Paketen wie zum Beispiel der Kotlin Standardbibliothek, auf die nur lesend zugegriffen werden kann.Die Methoden müssen ausserhalb einer Klasse stehen und werden wie in Beispiel 1.18 in Zeile 6 implementiert. Dabei ist zu beachten, dass innerhalb der Methode ausschließlich nur auf die öffentlichen Attribute und Methoden der Klasse zugegriffen werden kann. Private oder Beschütze bleiben weiterhin verborgen.

```
1 fun main() {
2         val i:Int = 8
3     println(i.half())
4 }
5
6 fun Int.half():Int{
7     return this/2
8 }
```

Codebeispiel 1.18: Erweiterungsmethoden

1.9 Object und Companion

In Java gibt es neben normalen Klassen auch noch statische Klassen. Diese sind im gesamten Projekt nur ein einziges mal vorhanden und können daher ohne vorherige Instanziierung verwendet werden. Dies ist nützlich um zum Beispiel Hilfsmethoden zur Verfügung zu stellen, für die keine Instanz eines Objektes benötigt wird. Die Alternative dafür in Kotlin sind Klassen, die an Stelle des Schlüsselwortes *class* mit dem Schlüsselwort *object* definiert werden. Beispiel 1.19 zeigt wie eine statische Klasse in Kotlin implementiert und verwendet wird.

```
1 fun main() {
2     Printer.print()
3     Printer.text = Printer.text.reversed()
4     Printer.print()
5 }
6
7 object Printer{
8     var text = "Hello World!"
9
10    fun print(){
11        println(text)
12    }
13 }
```

Codebeispiel 1.19: Statische Klasse

Wenn allerdings eine nicht-statische Klasse mit statischen Methoden und Attributen ausgestattet werden soll, so kann innerhalb der Klasse ein Companion-Object definiert werden. Variablen und Methoden innerhalb des Companion-Objects werden als statisch erkannt und sind daher für alle Instanzen der Klasse

identisch. Das Companion-Object wird wie in Beispiel 1.20 mit den Schlüsselwörtern *companion object* definiert. Im Beispiel wurde ein Counter für die Klasse implementiert, der mitzählt wie oft ein Objekt der Klasse erzeugt wurde. Da das Attribut counter im Companionobjekt enthalten ist, ist sein Wert für alle Klassen der selbe.

```
1  class Klasse{
2      constructor(){
3        counter++
4      }
5      companion object{
6          var counter:Int = 0
7
8          fun printCounter(){
9              println(counter.toString())
10         }
11     }
12 }
```

Codebeispiel 1.20: Companion Objekt

1.10 Interfaces

Ein Problem, was die Vererbung in der objektorientierten Programmierung mit sich bringt, ist die Tatsache, dass eine Kindklasse immer nur von höchstens einer Elternklasse erben kann. Wenn eine Klasse von mehreren Eltern erben soll, so ist dies nur über Interfaces möglich. Interfaces kommen dann zum Einsatz, wenn eine spezielle Klasse auf eine bestimmte Funktionalität beschränkt werden soll.Ein Beispiel wäre das speichern verschiedener Klassen in einer einzigen Liste, wenn alle eine Gemeinsamkeit teilen. Ein Beispiel ist in 1.21 zu sehen. Da wurden zwei Klassen implementiert, die jeweils zwei Interfaces implementieren. In Zeile 2 und 3 werden dann zwei Arrays erstellt, die jeweils Elemente von sowohl *Klasse1* als auch *Klasse2* enthalten. Die Klassen werden dann jeweils auf ihre Implementierung des Interfaces zurückgestuft. Weiter Funktionen der Klassen sind dann nicht mehr vom Array aus aufrufbar, sie können allerdings in den Interface-Methoden verwendet werden.

```
1  fun main() {
2      val a:Array<IA> = arrayOf(Klasse1(),Klasse2())
3      val b:Array<IB> = arrayOf(Klasse1(),Klasse2())
4      for(e in a){
5          e.printA()
6      }
7      for(e in b){
8          e.printB()
9      }
10 }
11
12 class Klasse1:IA,IB{
13     override val text = "Klasse1:IB"
14     override fun printA(){
15         println("Klasse1:IA")
16     }
17     override fun printB(){
18         println(text)
19     }
```

```
20 }
21
22 class Klasse2:IA,IB{
23     override val text = "Klasse2:IB"
24     override fun printA(){
25         println("Klasse2:IA")
26     }
27     override fun printB(){
28         println(text)
29     }
30 }
31
32 interface IA{
33     fun printA()
34 }
35
36 interface IB{
37     val text:String
38     fun printB()
39 }
```

Codebeispiel 1.21: Interfaces

Die Erstellung eines Interfaces funktioniert in Kotlin mit dem Schlüsselwort *interface* anstelle von *class*. Jedes Attribut und jede Methode eines Interfaces muss *public* sein, Attribute dürfen keine Werte im Interface zugewiesen bekommen und bei Methoden wird der Körper weggelassen. Wenn ein Interface von einer Klasse implementiert wird, muss jede Methode und jedes Attribut des Interfaces von der Klasse überschrieben werden.

1.10.1 Anonymes Interface

Neben der Implementierung in einer Klasse, kann ein Interface auch anonym verwendet werden. Dadurch muss nicht unbedingt eine Klasse erzeugt werden die das Interface implementiert, sondern das Interface wird zur Laufzeit im Code erzeugt. Dies wird zum Beispiel bei den *OnClickListener* der Buttons verwendet. Diese benötigen nämlich nur eine Implementation des OnClick-Interfaces und keine Klasse. Die Erzeugung eines anonymen Interfaces wird mit den Schlüsselwörtern *object:* gefolgt von dem Namen des Interfaces umgesetzt. Im darauffolgenden Block müssen dann alle notwendigen Funktionen des Interfaces überschrieben werden. Ein Beispiel für den Umgang mit anonymen Interfaces ist in Abbildung 1.22 gezeigt.

```
1  fun interface Print{
2      fun Print(text:String)
3  }
4
5  fun Execute(p:Print){
6      p.Print("Hello World")
7  }
8  fun main() {
9      Execute(object:Print{
10         override fun Print(text:String){
11             println(text)
12         }
13     })
14     Execute(object:Print{
15         override fun Print(text:String){
```

```
16                 println(text.reversed())
17             }
18         })
19 }
```

Codebeispiel 1.22: Anonyme Interfaces

1.10.2 Vereinfachung

Die Implementation eines anonymen Interfaces lässt sich allerdings noch vereinfachen, um den Code kürzer zu gestalten. Diese Operation wird in Kotlin *SAM* genannt. Dies ist dann möglich, wenn es im Interface nur eine einzige Methode gibt. Im ersten Schritt wird das Schlüsselwort *object:* weggelassen und die überschreibung der Methode durch einen Lambdaausdruck ersetzt. Dazu werden zuerst alle Eingabeparameter der Methode durch Komma getrennt deklariert und nach dem Lambda-Operator -> wird dann der Programmcode implementiert. Wenn das Interface in einer Funktion als Eingabeparameter verwendet wird, die ansonsten keine weiteren Parameter benötigt, kann das Interface ohne Namen direkt im Anschluss an den Funktionsnamen geschrieben werden. Dabei werden auch die runden Klammern der Funktion weggelassen. Dies ist in Abbildung 1.23 zu sehen.

```
1  fun interface Print{
2      fun Print(text:String)
3  }
4
5  fun Execute(p:Print){
6      p.Print("Hello World")
7  }
8
9  fun main() {
10     Execute{text->
11         println(text)
12     }
13     Execute{text->
14         println(text.reversed())
15     }
16 }
```

Codebeispiel 1.23: Anonyme Interfaces

Wenn die Methode ebenfalls nur höchstens einen einzigen Parameter besitzt, so kann dieser ebenfalls weggelassen werden, und auf den Parameter wird dann mit dem Schlüsselwort *it* zugegriffen. Dies ist in Abbildung 1.24 gezeigt.

```
1  fun interface Print{
2      fun Print(text:String)
3  }
4
5  fun Execute(p:Print){
6      p.Print("Hello World")
7  }
8
9  fun main() {
10     Execute{
11         println(it)
12     }
```

```
13      Execute{
14          println(it.reversed())
15      }
16  }
```

Codebeispiel 1.24: Zweite Vereinfachung des Anonymen Interfaces

```
1  fun interface Print{
2      fun Print(text:String)
3  }
4
5  fun Execute(p:Print){
6      p.Print("Hello World")
7  }
8
9  fun main() {
10     //Ohne SAM
11     Execute(object:Print{
12         override fun Print(text:String){
13             println(text)
14         }
15     })
16     //Mit SAM
17     Execute{
18         println(it.reversed())
19     }
20 }
```

Codebeispiel 1.25: Gegenüberstellung mit und ohne SAM

In Abbildung 1.25 sind die Implementierungen mit und ohne SAM gegenübergestellt. Dabei ist deutlich zu sehen, wie sehr die Verkürzung der Übersichtlichtkeit hilft.

1.11 Coroutines

Damit eine App benutzerfreundlich ist, ist eine wichtige Voraussetzung, dass der Nutzer zu jeder Zeit mit der App interagieren kann ohne auf eine Reaktion der App warten zu müssen. Wenn in einer App allerdings aufwendige Aufgaben gelöst werden müssen, kann dies auch mal länger dauern. Ein Beispiel wäre die Abfrage einer Datenbank, die Aufgrund der Netzverbindung und der eventuell großen Datenmengen längere Zeit in Anspruch nehmen kann.Würde diese Abfrage auf dem selben Thread wie die UI ausgeführt werden, so müsste der Nutzer solange warten, bis die Abfrage fertig ist, bis er mit der Bedienung der App fortfahren könnte. Um dieses Problem zu lösen, bietet Kotlin die Coroutines an. Diese lassen Code Asynchron zur UI ablaufen, wodurch diese nicht blockiert werden würde. Da das Thema zu Coroutines sehr umfangreich ist, wird hier nur der Teil erklärt, der auch in dem Projekt umgesetzt wurde.

1.11.1 Suspend

Damit eine Funktion in Kotlin asynchron zum Mainthread arbeiten kann, muss sie mit dem Schlüsselwort *suspend* versehen werden. Dies sorgt dafür, dass die Funktion gestartet,gestoppt und fortgesetzt werden kann[3]. Eine solche Funktion muss immer entweder in einem Coroutinescope, welches in Kapitel 1.11.2

näher erläutert wird, oder in einer weiteren Suspendfunction ausgeführt werden. Dies sorgt für Sicherheit, dass die Funktion den Mainthread nicht blockiert.

1.11.2 Coroutine Scope

Um eine Suspendfunction in einer synchronen Funktion aufzurufen, muss zuerst ein *Coroutinescope* definiert werden. Dieses kann ein globales Attribut einer Klasse sein, kann allerdings auch innerhalb einer Methode erzeugt werden.

Definition

Als globales Attribut kann das Scope wie in Beispiel 1.26 erstellt werden.

```
1 private val scope:CoroutineScope = CoroutineScope(CoroutineName
      ("Scope")+Dispatchers.IO)
```

Codebeispiel 1.26: Erzeugung eines Coroutinescopes

Dabei ist der Name des Scopes frei wählbar. Innerhalb des Konstruktors der Klasse *CoroutineScope()* werden bestimmte Eigenschaften für das Scope festgelegt. In diesem Fall wird dem Scope ein Name gegeben und ein Dispatcher zugeordnet.Die Attribute müssen mit einem + voneinander getrennt werden.

Dispatcher

Der Dispatcher gibt an, in welcher Umgebung das Scope ausgeführt wird. Die drei Möglichkeiten sind

- Main
- IO
- Default

Wenn der Main-Dispatcher gewählt werden würde, würde der Code innerhalb des Scopes auf dem selben Thread wie die UI ausgeführt werden. Der IO und der Default-Dispatcher laufen beide Asynchron zur UI, dabei ist der IO-Dispatcher für Netzwerk-oder Laufwerklastige Aufgaben, der Default-Dispatcher für CPU-lastige Aufgaben ausgelegt.

Aufruf

Um eine Coroutine in einem Scope zu starten, muss dies mit der Methode *launch* des Coroutinescopes eingeleitet werden. Jeglicher Code, der im Block dieser Methode ausgeführt wird, läuft asynchron auf dem vordefinierten Dispatcher. Ein Beispiel ist in Abbildung 1.27 zu sehen.

```
1 fun main() {
2     scope.launch{
3         System.out.println("Coroutine 1 started")
4         System.out.println("Calculation 1 started\n")
5         HeavyCalculation(100000000)
6         System.out.println("Calculation 1 ended\n")
7     }.invokeOnCompletion(){
8         System.out.println("Coroutine 1 finished\n")
9     }
```

```
10      scope.launch{
11          System.out.println("Coroutine 2 started")
12          System.out.println("Calculation 2 started\n")
13          HeavyCalculation(1000000)
14          System.out.println("Calculation 2 ended\n")
15      }.invokeOnCompletion(){
16          System.out.println("Coroutine 2 finished\n")
17      }
18  }
19
20  suspend fun HeavyCalculation(n:Long){
21      var result = 0L
22      for(i in 0 until n){
23          result += i
24      }
25      System.out.println(result)
26  }
```

Codebeispiel 1.27: Aufruf einer Coroutine

Die Reihenfolge der Ausgabe für Abbildung 1.27 ist:

- Coroutine 1 Started
- Calculation 1 Started
- Coroutine 2 Started
- Calculation 2 Started
- Result 2
- Calculation 2 Ended
- Coroutine 2 finished
- Result 1
- Calculation 1 Ended
- Coroutine 1 finished

Mit Hilfe der Funktion *invokeOnCompletion* kann Programmcode nach Beendigung der Corouine ausgeführt werden.

Kapitel 2

Backend

Um die App auch im Backend, also in der Hintergrundlogik zu verbessern, wurden Änderungen am Aufbau der App vorgenommen.

2.1 Model-View-Viewmodel

Es wurde das sogenannte *Model-View-Viewmodel (MVVM)*-Pattern implementiert, welches für eine bessere Aufteilung des Codes in UI, Datenbank und Hintergrundlogik sorgt. Das Pattern teilt die App in drei Aufgaben. Ein Teil kümmert sich um die Zugriffe auf Daten, also in diesem Fall die Room-Datenbank, die Restschnittstelle und die Sharedpreferences, ein anderer Teil kümmert sich ausschließlich um die UI, also sorgt dafür das dem Nutzer Daten schön angezeigt werden, die Navigation innerhalb der App funktioniert und Nutzereingaben und Auswahlmöglichkeiten erkannt werden. Der dritte Teil verbindent die ersten beiden miteinander. Stellt also Daten für die Ausgabe zur Verfügung und verarbeitet Nutzereingaben und Auswahlen. Der große Vorteil ist dabei, das Änderungen an der App einfacher durchgeführt werden können, da jeder Teil unabhängig von der Logik der anderen Teile arbeitet. Wenn zum Beispiel eine Änderung an der Datenbank vorgenommen wird, so müssen nur die entsprechenden Schnittstellen, also zum Beispiel die Room-Datenbank und das Datenbank-Repository angepasst werden. Die UI und die Viewmodels benötigen keine, oder nur kleine Änderungen.

2.1.1 Model

Der Model-Teil kümmert sich wie oben angesprochen um die Datenzugriffe. Dabei gibt es die direkten Schnittstellen zur Datenbank, also die UserDAO für die Room-Datenbank, die Retrofit-Schnittstelle für den Zugriff auf den externen Server und das SharedPreferences-Repository für die Zugriffe auf die SharedPreferences. Die UserDAO und Retrofit-Schnittstellen werden zudem in einem allgemeinen Datenbankrepository zusammengeführt.

Room-Datenbank

Die Room-Datenbank bietet dem Nutzer eine lokale Speichermöglichkeit, sodass die Daten vom extern Server auch ohne Internetverbindung weiter erhalten blei-

ben. Der Zugriff auf diese Datenbank erfolgt über das UserDAO-Interface. Dieses enthält diverse SQL-Abfragen nach dem SQLite-Standard für die Modifikation und Selektion der Datenbank. Jede Abfrage muss dabei mit dem Schlüsselwort *suspend* versehen werden. Dadurch werden diese zu Asynchronen Funktionen, müssen also aus einer Coroutine aufgerufen werden, wie es in Kapitel 1.11.1 erklärt wurde.

LiveData

Eine Ausnahme dabei bilden die LiveData-Abfragen. Diese benötigen keine *suspend*-Initialisierung sondern haben als Rückgabetyp ein Objekt der generischen Klasse LiveData. Der große Vorteil einer LiveData-Klasse ist, dass Ihnen ein sogenannter *Observer* angehängt werden kann. Dieser erkennt automatisch eine Veränderung in den Daten und stellt dann die neuen Daten zur weiterverarbeitung zur Verfügung. Der genauere Vorgang wird in Kapitel 3.1 näher erklärt. LiveData-Objekte sind die Hauptschnittstelle für eine Datenselektion aus der UI, da sie keine Coroutine benötigen.

Retrofit-Schnittstelle

Die Retrofit-Schnittstelle baut die Verbindung zum externen Server auf. Die Abfrage ähnelt dabei der Roomdatenbank aus Kapitel 2.1.1. Es wird ein Interface erstellt das gewisse vordefinierte Abfragen implementiert. Dabei wird allerdings nicht mit SQL-Befehlen gearbeitet, sondern es wird der URL-Pfad mit angegeben, der zu den Daten auf der Rest-Schnittstelle führt. Die *RetrofitHelper*-Klasse stellt dann den Zugriff über das Interface her. In ihr wird auch eine Base-URL festgelegt, die für alle Daten identisch ist, sowie ein *GSON*-Konverter, da die Daten, die vom Server kommen, als JSON versendet werden. Alle Abfragen müssen ebenfalls mit dem Schlüsselwort *suspend* versehen werden, um einen Asynchronen Zugriff zu ermöglichen.

Datenbank-Repository

Um die direkten Datenbankzugriffe vor weiteren Klassen abzuschirmen, wurde das Datenbank-Repository implementiert. Dieses hat Zugriff sowohl auf die lokale Room-Datenbank als auch auf die externe Datenbank via der Retrofitschnittstelle. Alle vorhandenen Funktionen der UserDAO-Klasse und der Retrofit-Schnittstelle werden vom Repository neu implementiert. Dabei wird die asynchronität erhalten, indem die Funktion *withContext* verwendet wird. Diese ist nichts anderes als ein neuer Weg, um eine *async*-Funktion zu schreiben, wo die Methode *await()* nicht aufgerufen werden muss [4]. Der Dispatcher wird dabei für jede Funktion auf den *IO-Dispatcher* gesetzt, um die Optimierung beim Datenzugriff zu verwenden.

SharedPreferences-Repository

Neben den Datenbanken werden auch noch weitere Daten in den SharedPreferences gespeichert. Dies sind einfache XML-Dateien, die zu einem Schlüsselbegriff einen Wert speichern können. Verwendet werden die SharedPreferences zum Beispiel, um die Appeinstellugnen des Nutzers lokal abzuspeichern. Da der Zugriff auf die Daten immer mit dem Schlüsselbegriff erfolgen muss, wurde ein

Repository implementiert, welches diese Aufgabe übernimmt, um einen Fehler bei der Programmierung durch falsche Schlüsselbegriffe zu vermeiden.

2.1.2 Viewmodel

Um die Daten aus den Repositories an die UI weiter zu geben, wurden sogenannte ViewModels erstellt. Diese verbinden die UI mit den Datenbanken und kontrollieren zudem die weitere Logik der App, die Unabhängig von der UI ist. Eine Aufgabe des ViewModels ist es, die Asynchronen Funktionen der Repositories zu synchronisieren, also die *suspend*-Funktionen in normale Funktionen umzuwandeln, damit diese aus der UI aufgerufen werden können, ohne auf eine Coroutine zurückgreifen zu müssen. Wie in Kapitel 3.1 erklärt, soll die UI keine Implementation einer Datenbank oder einer Coroutine beinhalten.

Datenmodifikation

Um die Datenmodifikation der UI zu ermöglichen, kann die entsprechende *suspend*-Funktion in dem *ViewModelScope* ausgeführt werden. Dies ist ein CoroutineScope, welches vom viewModel bereitgestellt wird. Da die Datenmodifikation keine Rückgabewerte benötigt ist dies ohne weiteres möglich.

Datenselektion

Bei der Datenselektion gibt es mit den ViewModels allerdings ein großes Problem. Um die *suspend*-Funktion aufzulösen muss die Rückgabe außerhalb des CoroutineScopes erfolgen. Dadurch passiert die Rückgabe allerdings zeitlich vor dem Erhalt der Daten durch das Repository. Aus diesem Grund müssen die erhaltenen Daten noch innerhalb des CoroutineScopes verarbeitet werden, da der Code dort weiterhin strukturell abgearbeitet wird. Eine Möglichkeit dafür ist zum Beispiel, wenn Daten vom externen Server abgefragt werden, diese direkt nach dem Erhalt in die Lokale Datenbank zu speichern. Dies ist zum Beispiel in Abbildung 2.1 zu sehen. Auf die neuen Daten kann dann mit Hilfe von LiveData-Variablen zugegriffen werden. Dafür können entweder die vordefinierten Abfragen aus dem Repository verwendet werden, es können aber auch eigene LiveData-Objekt erstellt werden, wenn Daten benötigt werden, die nicht direkt aus der Datenbank kommen sollen.

```
1 fun fetchCourses() {
2         viewModelScope.launch {
3             val courses = repository.fetchCourses()
4             if(courses!=null)
5             {
6                 insertCoursesJSON(courses)
7             }
8         }
9     }
```

Codebeispiel 2.1: Selektion von Daten im ViewModel

```
1     val liveCoursesForFaculty = MutableLiveData<List<Course
      >?>()
2
3     override fun setReturnFaculty(faculty: Faculty){
4         super.setReturnFaculty(faculty)
```

```
5           viewModelScope.launch {
6               val courses = getCoursesByFacultyid(faculty.fbid)//
                    Funktion suspend fun getCoursesByFacultyid(id:
                    Int) aus der Klasse BaseViewModel muss
                    innerhalb eines CoroutineScopes aufgerufen
                    werden.
7               liveCoursesForFaculty.postValue(courses)//Daten nur
                     innerhalb des CoroutineScopes konsistent
                    vorhanden.
8           }
9       }
```

Codebeispiel 2.2: Verwendung von MutableLiveData-Objekten

Um eigene LiveData-Objekte zu erzeugen, müssen diese als MutableLiveData-Variablen definiert werden, wie es in Beispiel 2.2 zu sehen ist. Der Zugriff erfolgt dann über die Methode *postValue*. Dadurch werden die Daten im LiveData-Objekt mit den neuen Daten ersetzt und die Observer werden über die neuen Daten benachrichtigt. Auf die Daten kann über das Attribut *value* auch direkt lesend zugegriffen werden, wenn die Daten benötigt werden, ohne das es eine Veränderung gegeben hat. Um die Applikation überschaubar zu halten, wurde für jede Activity und jedes Fragment ein eigenes ViewModel erstellt. Dadurch entsteht zwar eine gewisse Redundanz, da möglicherweise Funktionen mehrfach implementiert sind, es erhöht aber signifikant die Übersichtlichkeit, da die Vorhandenen Attribute und Methoden nur in einer Klasse verwendet werden. Trotzdem wurde noch ein BaseViewModel erstellt, von dem alle anderen ViewModels erben, welches Funktionen zur Verfügung stellt, die in allen Activities und Fragmenten relevant sind. Dies betrifft zum Beispiel den Zugriff auf die SharedPreferences.

ViewModelFactory

Um ein ViewModel in einer View zu instanziieren, wird eine ViewModelFactory verwendet. Die Implementierung ist in Abbildung 2.3 zu sehen.

```
1 class ViewModelFactory(application: Application):
      ViewModelProvider.AndroidViewModelFactory(application)
```

Codebeispiel 2.3: Implementierung der ViewModelFactory

Ein ViewModel soll keine direkte Verbindung zur UI beinhalten, daher werden keine View bezogenen Parameter übergeben. Weder als globale Variable, noch über Eingabeparameter. Die einzige Ausnahme ist der Applikationscontext, auf den mittels des Parameter *application* aus dem Konstruktor zugegriffen werden kann.

2.2 Weitere Funktionen

Neben der Implementierung des Model-View-ViewModel-Patterns, wurden für die App noch weitere Funktionalitäten zur Verfügung gestellt.

2.2.1 Updating-Service

Eine neue Funktion ist der Updating Service. Dieser erkennt, wenn eine neue Version der App im PlayStore verfügbar ist und leitet nach Zustimmung des Nutzers den Updatingprozess ein. Für den UpdatingService wurde der UpdateManager verwendet, welcher von Google zur Verfügung gestellt wird. Abbildung 2.4 zeigt die Initialisierung des UpdateManagers. Die Erzeugung ist über die *AppUpdateManagerFactory* möglich. Anschließend wird dem Manager ein Listener-Interface übergeben, welches beschreibt was passieren soll, wenn der Manager mit der Prüfung auf Updates abgeschlossen ist. In diesem Fall wird geprüft ob ein Update verfügbar ist und eine Flexible Aktualisierung möglich ist. Bei einem Flexiblen Update kann der Nutzer die App weiter verwenden, auch wenn er kein Update vornimmt. Die Alternative wäre *Immedate*, wo die App beendet wird, wenn der Nutzer sich weigert ein Update vor zu nehmen. Wenn ein Update möglich ist, wird mit der Methode *startUpdateFlowForResult* ein Dialog geöffnet, welcher den Nutzer auf das neue Update hinweist.

```
1  private fun initUpdateManager() {
2          updateManager = AppUpdateManagerFactory.create(this)
3          updateManager?.appUpdateInfo?.addOnSuccessListener {
4              if (it.updateAvailability() == UpdateAvailability.
                   UPDATE_AVAILABLE && it.isUpdateTypeAllowed(
5                              AppUpdateType.FLEXIBLE
6                      )
7              ) {
8                  updateManager?.startUpdateFlowForResult(
9                          it,
10                         AppUpdateType.FLEXIBLE,
11                         this,
12                         UPDATE_REQUEST_CODE
13                 )
14             }
15         }
16         updateManager?.registerListener(
               installStateUpdateListener)
17     }
```

Codebeispiel 2.4: Initialisierung des UpdateManagers

Nach einer Zustimmung des Benutzers wird das Update im Hintergrund heruntergeladen, und im Anschluss wird über den Listener in Abbildung 2.5 eine Snackbar am unteren Rand des Bildschirms angezeigt, über die der Benutzer die Installation beenden kann. Nach der Installation wird die App automatisch neu gestartet und das Update ist beendet.

```
1  private val installStateUpdateListener:
       InstallStateUpdatedListener =
2              InstallStateUpdatedListener {
3                  if (it.installStatus() == InstallStatus.
                       DOWNLOADED) {
4                      Snackbar.make(
5                              findViewById(android.R.id.content),
6                              "Update is ready",
7                              Snackbar.LENGTH_INDEFINITE
8                      ).setAction("Install") {
9                          updateManager?.completeUpdate()
10                     }.show()
11                 }
```

```
12              }
```

Codebeispiel 2.5: Listener nach Beendigung des herunterladens

2.2.2 Background-Worker

Ein weiteres neues Feature ist ein Hintergrundprozess, der automatisch in einem vorgegebenen Intervall prüft, ob der Prüfplan auf dem Server verändert wurde. Verwendet wurde dafür der *WorkManager* von Android. In Abbildung 2.6 ist zu sehen, wie dem WorkManager ein neuer Request hinzugefügt wird. Über die *Contraints* können Beschränkungen hinzugefügt werden. In diesem Fall können die Updates nur bei bestehender Internetverbindung durchgeführt werden. Über das Interval wird angegeben, in welchen Zeitabständen der Request ausgeführt wird. Die Daten dafür werden bei den Settings in den SharedPreferences gespeichert und sind über das Einstellungsmenü veränderbar. Bei der Erstellung eines Requests muss eine Workerklasse angegeben werden. In diesem Fall ist dies die Klasse *CheckForDatatbaseUpdateWorker*. Diese muss von der *Worker*-Klasse erben und die Methode *doWork* überschreiben. In dieser wird definiert, was passieren soll, wenn der Request ausgeführt wird. Die Implementierung dieser Methode ist in Abbildung 2.7 zu sehen. Es werden die Prüfplaneinträge vom Server geladen und der *TimeStamp*, welcher Auskunft über die letzte Veränderung gibt, wird mit den Daten aus der lokalen Datenbank verglichen. In diesem Fall, oder wenn schon die Anzahl der Einträge verschieden sind, wird eine Push-Benachrichtigung an den Benutzer gesendet. Dafür wird die *PushService*-Klasse verwendet, die erstellt wurde, um Benachrichtigungen zu senden. Der *updateWorkerName* kann frei gewählt werden und ist hier eine Konstante. Durch den Parameter *ExistingPeriodicWorkPolicy.KEEP* wird eingestellt, dass, sollte bereits ein Request mit dem selben Namen vorhanden sein, nur der vorhandene Beibehalten wird. So wird verhindert, dass mehrere Backgroundtask gleichzeitig aktiv sind, da nur einer benötigt wird.

```
1
2       val constraints = Constraints.Builder()
3                   .setRequiredNetworkType(NetworkType.CONNECTED)
4                   .build()
5           val interval = spRepository.getUpdateIntervalTimeHour()
                * 60 + spRepository.getUpdateIntervalTimeMinute().
                toLong()
6           val checkRequest = PeriodicWorkRequestBuilder<
                CheckForDatabaseUpdateWorker>(interval, TimeUnit.
                MINUTES)
7                   .setConstraints(constraints)
8                   .build()
9
10          context.let { WorkManager.getInstance(it).
                enqueueUniquePeriodicWork(updateWorkerName,
                ExistingPeriodicWorkPolicy.KEEP, checkRequest) }
11      }
```

Codebeispiel 2.6: Initialisierung eines Periodic Requests für den Workmanager

```
1 override fun doWork(): Result {
2           PushService.sendNotification(context,"The
                examinationplan has been updated!") //
```

```
3
4           iOScope.launch {
5               if(checkDatabase())
6               {
7                   PushService.sendNotification(context,"The
                        examinationplan has been updated!") //
8               }
9           }
10          return Result.success()
11      }
12
13      private suspend fun checkDatabase():Boolean{
14          return withContext(Dispatchers.IO){
15              val periode = spRepository.getCurrentPeriode()
16              val termin = spRepository.getCurrentTermin()
17              val examinYear = spRepository.getExamineYear()
18              val Ids = repository.getChoosenCourseIds(true)
19              val courseIds = JSONArray()
20              if (Ids != null) {
21                  for (id in Ids) {
22                      try {
23                          val idJson = JSONObject()
24                          idJson.put("ID", id)
25                          courseIds.put(idJson)
26                      } catch (e: JSONException) {
27                          e.printStackTrace()
28                      }
29                  }
30              }
31              if (periode == null || termin == null || examinYear
                    == null || courseIds.toString().isNullOrEmpty
                    ()) {
32                  return@withContext false
33              }
34              val remoteEntries = repository.fetchEntries(periode
                    , termin, examinYear, courseIds.toString())
35              val localEntries = repository.getAllEntries()
36              if(remoteEntries.isNullOrEmpty()||localEntries.
                    isNullOrEmpty())
37              {
38                  return@withContext false
39              }
40              if(remoteEntries.size != localEntries.size){
41                  return@withContext true
42              }
43              for(i in remoteEntries.indices){
44                  val remoteEntry = remoteEntries.get(i)
45                  val remoteId = remoteEntry.ID
46                  val localEntry = remoteId?.let { repository.
                        getEntryById(it) }
47                  if(remoteEntry.Timestamp!=localEntry?.timeStamp
                        )
48                  {
49                      return@withContext true
50                  }
51              }
52              return@withContext false
53          }
54      }
```

Codebeispiel 2.7: Implementierung der Methode *doWork*

2.3 Kalender Synchronisation

Die Möglichkeit, die gewählten Prüfungen automatisch in den Kalender eintragen zu lassen war zwar schon vorher implementiert, allerdings kam es dabei immer wieder zu Fehlern, weshalb eine neu Implementierung für die Kalenderkommunikation vorgenommen wurde. Dafür wurde das statische Objekt *CalendarIO* erstellt, welches diverse Funktionalitäten zur Kommunikation mit dem Kalender zur Verfügung stellt. Es ist Möglich, Prüfungen in den Kalender zu übertragen und wieder zu löschen.

2.3.1 ID

Um die Events später wieder identifizieren zu können wird jedem eine eigene Id mitgegeben. Diese wird dann in den Shared Preferences zusammen mit dem zugehörigen Eintrag abgespeichert. Die Id wird aus einer zufällig generierten UUID erzeugt und ist 64-Bit (Long) groß.

2.3.2 Einfügmethoden

Der Nutzer hat über die Einstellung die Möglichkeit, selber zu entscheiden auf welche Art die Einträge in den Kalender übertragen werden sollen.

Automatisch

Wenn *Automatisch* gewählt wurde werden alle Einträge direkt eingefügt, ohne eine das der Nutzer etwas mit bekommt. Dies gilt natürlich nur wenn die Synchronisation in den Einstellungen aktiviert wurde.

Manuell

Eine weiter Möglichkeit ist das manuelle Einfügen. Dabei wird der Prozess zur Erstellung eines neuen Eintrages des Kalenders aufgerufen und mit den Daten des Prüfplaneintrages voreingestellt. Dadurch kann der Benutzer selber entscheiden, wie die Informationen in den Eintrag eingebaut werden sollen.

Nachfragen

Die dritte Möglichkeit, *Ask*, fragt bei jedem Eintrag den Benutzer, ob der Eintrag Manuell oder Automatisch eingefügt werden soll.

2.3.3 Verschiedene Kalender

Des weiteren kann der Benutzer in den Einstellungen festlegen, in welchen Kalender die Einträge eingefügt werden sollen. Dazu werden alle auf dem Smartphone vorhandenen Kalender gesucht und die heraus gefiltert, deren Zugriffslevel es erlaubt, dort Einträge ein zu fügen. Dazu muss der Zugriffslevel größer oder gleich 700 sein. Als default Kalender wird der Primäre Kalender des Benutzers voreingestellt.

Kapitel 3

Frontend

Wenn das Backend feststeht, kann der View-Teil der Applikation erstellt werden. Dieser bildet die Schnittstelle zwischen des Benutzers und der Applikation. In der View werden Layouts erstellt, die die UI-Elemente dem Nutzer auf eine strukturierte Art und Weise präsentieren und es werden Designs für die Elemente erstellt, die eine schön aussehende App ausmachen. Zuletzt müssen die UI-Elemente noch mit einer Logik versehen werden, damit sie ihre Funktion erfüllen können.

3.1 View

Durch die Implementierung des Model-View-ViewModel-Patterns, mussten in der View ebenfalls Anpassungen vorgenommen werden. Alle Referenzen zu den Datenbanken, sowie zu den SharedPreferences konnten entfernt werden. Stattdessen wird mithilfe der LiveData-Objekte, wie in Kapitel 2.1.2 erklärt, auf die Daten der Datenbank zugegriffen werden. Dafür muss allerdings zuerst das ViewModel in der Activity oder dem Fragment implementiert werden. Hierzu dient der ViewModelProvider. Dieser erzeugt mit Hilfe der ViewModelFactory, die in Kapitel 2.1.2 erstellt wurde, ein Objekt des entsprechenden ViewModels für die View. Der Code für die Erzeugung ist in Abbildung 3.1 zu sehen.

```
1     private lateinit var viewModel: TermineViewModel
2
3 override fun onViewCreated(view: View, savedInstanceState:
      Bundle?) {
4         super.onViewCreated(view, savedInstanceState)
5         viewModel = ViewModelProvider(
6                 requireActivity(),
7                 ViewModelFactory(requireActivity().application)
8         )[TermineViewModel::class.java]
9         }
```

Codebeispiel 3.1: ViewModel-Erzeugung

Dabei kann durch ändern des Klassentypen, hier *TermineViewModel::class.java* das der Activity oder dem Fragment entsprechende ViewModel zugeordnet werden. Danach kann das ViewModel wie ein normales Objekt behandelt werden. Wenn auf Objekte der Datenbank zugegriffen werden soll, wie es zum Beispiel

bei den Recyclerviews der Fall ist, müssen Observer für die entsprechenden LiveData-Objekte erstellt werden.

```
1        recyclerViewExamAdapter = RecyclerViewExamAdapter(
             mutableListOf(),viewModel)
2        recyclerView4.adapter = recyclerViewExamAdapter
3        recyclerView4.visibility = RecyclerView.VISIBLE
4        viewModel.liveEntryList.observe(viewLifecycleOwner){
             entryList ->
5            entryList?.let { recyclerViewExamAdapter.
                 updateContent(Filter.validateList(it).
                 toMutableList()) }
6            termineFragment_swiperefres.isRefreshing = false
7        }
```

Codebeispiel 3.2: Zugriff auf Datenbankobjekte

Dieser Vorgang ist in Abbildung 3.2 beispielhaft dargestellt.Bei den LiveData-Objekten handelt es sich um sogenannte *lifecycle-aware*-Objekte. Diese haben Zugriff auf den aktuellen Lifecycle der Androidkomponenten und können daher Methoden wie *onCreate()* oder *onDestroy()* überschreiben. Dadurch können sie die Komponente aktualisieren wenn es eine Änderung in der Datenbank gibt[1]. Im Anhang B.1 und B.2 sind die unterschiedlichen Lifecycle für Activity und Fragment abgebildet. Wie zu erkennen ist, wird die *onCreateView()*-Methode eines Fragmentes jedes mal aufgerufen, wenn die View zerstört und neu erstellt wurde. Dies ist zum Beispiel der Fall, wenn zwischen mehreren Fragmenten Navigiert wird. Sollte ein Observer in der *onCreateView()*-Methode erstellt werden, so wird dieser jedes mal neu erzeugt, wenn von dem Fragment weg und wieder zurück navigiert wird. Um dies zu vermeiden, wird für den LifecycleOwner der *viewLifecycleOwner* verwendet. Dieser sorgt dafür, dass jeder Observer nur ein einziges mal erzeugt wird. Im Lifecycle einer Activity, wird die *onCreate()*-Methode nur ein einziges mal aufgerufen, auch wenn sich die View ändert, wie es in Anhang B.1 zu sehen ist. Daher kann dort für den Lifecycleowner die Activity verwendet werden[9].

3.2 Startseite

Um die Startseite benutzerfreundlicher zu gestalten wurde die *BottomNavigation* durch ein TabLayout mit integriertem ViewPager eingebaut.

3.2.1 TabLayout

Das Tablayout ist eine alternative Möglichkeit, um mehrere verschiedene Fenster zu verwalten und zwischen diesen zu Navigieren. Dabei wird zwischen den Seiten durch die aktuelle Position unterschieden und bei einer Änderung wird das entsprechende Fragment erzeugt. In diesem Fall wurden die beiden Fragmente fest für die jeweilige Position definiert, eine dynamische Definition wäre aber möglich. Der Code zur Initialisierung ist in Abbildung 3.3 zu sehen.

```
1        recyclerViewExamAdapter = RecyclerViewExamAdapter(
             mutableListOf(),viewModel)
2        recyclerView4.adapter = recyclerViewExamAdapter
3        recyclerView4.visibility = RecyclerView.VISIBLE
```

```
4           viewModel.liveEntryList.observe(viewLifecycleOwner){
                entryList ->
5               entryList?.let { recyclerViewExamAdapter.
                    updateContent(Filter.validateList(it).
                    toMutableList()) }
6               termineFragment_swiperefres.isRefreshing = false
7           }
```

Codebeispiel 3.3: Initialisierung des TabLayouts

3.2.2 ViewPager

Durch einbinden des ViewPagers hat der Benutzer die Möglichkeit, zwischen den verschiedenen Tabs durch ein wischen des Bildschirmes zu wechseln. Dem ViewPager muss ein adapter übergeben werden, welcher ähnlich wie das TabLayout anhand der Position ein neues Fragment erzeugt. Zusätzlich muss der ViewPager dem TabLayout noch als Parameter mit übergeben werden. Da das Wischen ebenfalls das SwipeToDelete-Feature aktivieren würde, wurde dieses aus der Applikation entfernt. Da es sich bei dem ViewPager um ein Layout handelt, welches über den die darunter liegende UI gelegt wird, muss beim Wechsel zu einem Fragment das nicht im ViewPager angezeigt werden soll, dieser deaktiviert, und im umgekehrten Fall wieder aktiviert werden.

3.3 Farben

Damit eine Applikation schön aussieht, benötigt es eine gute Auswahl an Farben, die die einzelnen Elemente annehmen. Dafür wurden für die App eigene XML-Attribute angelegt, die eine gewisse Struktur in den Farben vorgibt. Eine Auswahl der wichtigsten Attribute ist in der folgenden Liste zu sehen.

- colorPrimary - Die Primärfarbe für die App. Sie bestimmt das grundlegende Farbthema
- colorOnPrimary - Eine Farbe, die auf der Primärfarbe gut sichtbar ist. Überwiegend als Textfarben verwendet.
- colorPrimaryDark - Eine dunklere Variante der Primärfarbe um für ein wenig Abwechslung zu sorgen.
- colorOnPrimaryDark - Eine Farbe, die auf der dunklen Primärfarbe gut sichtbar ist. Überwiegend als Textfarben verwendet.
- colorPrimaryLight - Eine hellere Variante der Primärfarbe.
- colorOnPrimaryLight - Eine Farbe, die auf der helleren Primärfarbe gut sichtbar ist. Überwiegend als Textfarben verwendet.
- colorAccent - Eine Akzentfarbe, die einen Kontrast zur Primärfarbe bildet. Diese muss gut mit der Primärfarbe synergieren.
- colorOnPrimaryLight - Eine Farbe, die auf der Akzentfarbe gut sichtbar ist. Überwiegend als Textfarben verwendet.

- colorBackground - Eine Farbe für einen farblosen Hintergrund.
- colorOnPrimaryLight - Eine Farbe, die auf dem Hintergrund gut sichtbar ist. Überwiegend als Textfarben verwendet.

DarkMode

Ein neues Feature für die App ist die Möglichkeit, einen DarkMode zu aktivieren. Dadurch werden alternative Farben zur Darstellung verwendet, die ein eher dunkles Farbspektrum abbilden, wie dunkle Grautöne oder sogar Schwarz. Das Gesamtbild ist dadurch schonender für die Augen, außerdem kann durch die dunklen Farben zudem Energie gespart werden[6]. Android Studio bietet von sich aus eine Möglichkeit, den DarkMode,beziehungsweise Night-Mode, zu implementieren. Dafür wird zusätzlich zu der bereits vorhandenen colors.xml Datei eine weitere colors-night.xml Datei angelegt. In dieser können Fraben, die in der colors.xml Datei festgelegt wurden, überschrieben werden. Dabei müssen die Farbennamen jeweils übereinstimmen. Aktiviert wird der DarkMode in der App über die Methode *AppCompatDelegate.setDefaultNightMode()*, wie es in Abbildung 3.4 zu sehen ist.

```
1    AppCompatDelegate.setDefaultNightMode(if (darkMode)
         AppCompatDelegate.MODE_NIGHT_YES else AppCompatDelegate
         .MODE_NIGHT_NO)
```

Codebeispiel 3.4: Aktivieren des Darkmodes

3.4 Themen

Da jeder Mensch unterschiedliche Farben mag, wurde die Möglichkeit implementiert, mehrere Farbthemen zu erstellen, aus denen der Benutzer sich eines auswählen kann. Ein Thema wird in der styles.xml Datei erstellt und weist den in Abschnitt 3.3 beschriebenen Attributen bestimmte Farben zu. Die zugewiesenen Farben müssen vorerst allerdings in den colors.xml Dateien definiert werden, um wie in Abschnitt 3.3 beschrieben, den DarkMode zu ermöglichen. Nachdem ein Thema definiert wurde, muss die Id des Themas, die über das Attribut *R.style.ThemeName* erreichbar ist, in die *themelist* in der Klasse *Utils* eingetragen werden, damit sie für den Nutzer in den Einstellungen zur Auswahl steht. Alle XML-Dateien zur Erstellung neuer Themen sind im Anhang A dargestellt.

3.5 Filter

Um für den Nutzer die App-Bedienung zu erleichtern, wurden die verschiedenen Suchfunktionen durch einen allgemeinen Filter ersetzt. Dieser ist als *Alertdialog* von der Toolbar zu erreichen und gibt dem Nutzer die Möglichkeit, eine Liste von anstehenden Prüfungen nach folgenden Kriterien zu Filtern.

- Studiengang
- Modul
- Erstprüfer

- Datum
- Semester

Die Daten für die Filterwahl werden in einer eigenen Klasse *Filter* gespeichert. Jeder Wert kann auf Null gesetzt werden, was als ungefiltert interpretiert wird. Wenn ein Wert des Filters verändert wird, so wird ein *onFilterChangedCallback* ausgelöst. Dieser ist eine Liste von Methoden, die bei einer Filteränderung aufgerufen werden. Auf diese Weise können verschiedene Fragmente erkennen, wann der Filter geändert wurde und entsprechend eine Veränderung in der UI vornehmen. Diese Logik wird benötigt, da sowohl das Fragment für alle anstehenden Prüfungen (*TermineFragment.kt*) als auch das Fragment für die Favorisierten Prüfungen (*FavoritenFragment.kt*) ihre Listen über den selben Filter filtern sollen. Die Implementierung des Filters ist in Anhang C.1 dargestellt. Der Zugriff auf den *onFilterChangedCallback* ist in Abbildung 3.5 am Beispiel für die Klasse *TermineFragment.kt* gezeigt.

```
1
2 Filter.onFilterChangedListener.add {
3             viewModel.liveEntryList.value?.let { Filter.
                  validateList(it) }?.let {
                  recyclerViewExamAdapter.updateContent(it) }
4         }
```

Codebeispiel 3.5: Zugriff auf den OnFilterChangedCallback

3.6 Suchleiste

Zusätzlich zum Filter wurde eine Suchleiste in die Actionbar eingebaut, über die der Nutzer von jedem Fragment aus nach einem bestimmten Modul suchen kann und zum gewählten Modul weiter geleitet wird. Die Implementierung der Suchleiste ist in Abbildung 3.6 gezeigt. Es wird die Liste aller anstehenden Prüfungen in eine Stringliste umgewandelt, welche dann dem SearchView als AutoCompleteAdapter übergeben wird. Dadurch bekommt der Nutzer nach Eingabe der ersten paar Buchstaben Vorschläge für Prüfungen, die mit diesen Buchstaben beginnen. Nach Beendigung der Eingabe wird der Filter zurückgesetzt und der Wert für den Modulnamen auf die Eingabe gesetzt. Anschließend wird der Nutzer zur Terminenübersicht weitergeleitet, wo dann nur noch die Module mit dem gesuchten Namen angezeigt werden.

```
1
2 private fun initSearchView(menu: Menu) {
3         val search: SearchView = menu.findItem(R.id.
              menu_item_search).actionView as SearchView
4         val searchAutoComplete: SearchAutoComplete = search.
              findViewById(R.id.search_src_text)
5         viewModel.liveEntriesOrdered.observe(this) { entryList
              ->
6             val list: MutableList<String> = mutableListOf()
7
8             entryList?.forEach { entry -> list.add(entry.module
                  ?: "") }
9             searchAutoComplete.setAdapter(
10                    ArrayAdapter(
11                            this,
```

```
12                                  R.layout.simple_spinner_item,
13                                  list
14                          )
15                  )
16          }
17
18          searchAutoComplete.setOnItemClickListener { adapterView
                , view, i, l ->
19              search.setQuery(adapterView.getItemAtPosition(i).
                    toString(), true)
20          }
21          search.setOnQueryTextListener(object : SearchView.
                OnQueryTextListener {
22              override fun onQueryTextChange(text: String?):
                    Boolean {
23                  return true
24              }
25
26              override fun onQueryTextSubmit(text: String?):
                    Boolean {
27                  Filter.reset()
28                  Filter.modulName = if (text.isNullOrBlank())
                        null else text
29                  changeFragment("Suche", Terminefragment())
30                  return true
31              }
32          })
33      }
```

Codebeispiel 3.6: Implementierung der Suchleiste

Kapitel 4

Weiterarbeit

Damit die App auch weiterhin gut strukturiert bleibt, werden in diesem Kapitel ein paar Regeln erläutert, an die sich gehalten werden sollte um weiterhin eine gute Weiterarbeit zu ermöglichen.

4.1 Farbthemen

Um neue Themen zu erstellen, müssen zuerst die notwendigen Farben definiert werden. Ein Thema besteht aus 11 Attributen, denen jeweils eine Farbe zugeordnet werden muss. Diese Attribute sind in folgender Aufzählung dargestellt.

- colorPrimary
- colorOnPrimary
- colorPrimaryDark
- colorOnPrimaryDark
- colorPrimaryLight
- colorOnPrimaryLight
- colorAccent
- colorOnAccent
- colorBackground
- colorOnBackground
- actionMenuTextColor

Zusätzlich zu diesen 11 Attributen gibt es auch not weiter, welche allerdings für alle Themen identisch sind. Diese sind im *BaseTheme* festgelegt, ein neues Thema muss daher das *BaseTheme* als parent implementieren.
Um neue Farben zu definieren müssen diese in die beiden **colors.xml**-Dateien geschrieben werden. Dabei ist die eine für den Lightmode und die andere (night) für den Darkmode. Bei der Benennung kann sich gerne an den bestehenden orientiert werden. Falls dringend neue Attribute benötigt werden, können diese in

der **attr.xml**-Datei definiert werden. Diese müssten dann auch für jedes schon bestehende Thema initialisiert werden.
Innerhalb der UI darf ausschließlich auf die oben genannten Attribute zurückgegriffen werden. Dies geschieht über den Ausdruck *?attr/farbe*.

4.2 Strings

Wenn in der UI Text angezeigt werden soll, so ist dieser unbedingt in die **strings.xml**-Datei zu extrahieren. Dies ist notwendig um eventuell später mehrere Sprachen zur Verfügung zu stellen.

4.3 Bezeichnungen

Bei den Bezeichnungen für Klassen oder XML-Elemente kann gerne an dem bestehenden Schema festgehalten werden. Die Bezeichnungen sollten auf jeden Fall im Projekt eindeutig sein und grob beschreiben, was dahinter steckt.

4.4 MVVM

Um an dem Model-View-ViewModel-Pattern fast zu halten ist hier nochmal eine kurze Erklärung, was es zu beachten gilt.

4.4.1 Model

Bei dem Model handelt es sich um jegliche Zugriffe auf Daten. Dies können zum Beispiel die Datenbanken oder die Shared Preferences sein. In den sogenannten repositories werden die Datenzugriffe gebündelt. Es wird allerdings keine intelligente Logik implementiert sondern einfach nur ein simpler Zugriff. Die komplexere Logik, zum Beispiel das Hinzufügen von Kalendereinträgen bei Favorisierung eines Eintrages, ist Teil des ViewModels.

4.4.2 ViewModel

In dem ViewModel ist die Logik hinter der App implementiert. Zum einen werden hier die Model-Zugriffe neu definiert, aber auch andere Logik die nicht direkt etwas mit der UI zu tun hat gehört ins ViewModel. Die Zugriffe auf das Model können in diesem Fall auch komplexer aussehen, es darf allerdings ausschließlich über die Repositories auf das Model zugegriffen werden.

4.4.3 View

in die View gehört ausschließlich die Logik zur Initialisierung und Verwaltung der UI-Elemente. Verboten in der View sind Zugriffe auf das Model, sowie Coroutines. Für beides soll das ViewModel verwendet werden.

4.5 Dokumentation

Jede Methode,Klasse oder Parameter soll mit einem kurzen Kommentar versehen werden. Bei Methoden und Klassen sollte ebenfalls eine kurze Erklärung zu den Ein-und Ausgabeparametern, sowie die Appversion, in der die Methode oder Klasse erstellt wurde. Zusätzlich kann auch der Autor mit angegeben werden und es können auch weitere Verlinkungen hinzugefügt werden. Dabei kann sich an den bestehenden orientiert werden.

Fazit

Ziel dieser Arbeit war es, die bestehende Prüfungsplan App zu verbessern. Erzielt wurde dies durch eine besser strukturierte UI mit einheitlichen Farben, mehr Einstellungsmöglichkeiten zur Personalisierung, ein besser strukturierter Code mithilfe des Model-View-ViewModel-Patterns und einheitlicheren Klassen und Methoden sowie eine umfangreiche Dokumentation.

Es gibt allerdings noch einige Punkte, an denen weitergearbeitet werden kann. Wie jede andere App auch ist diese nicht Fehlerfrei, daher müssen die noch bestehenden Bugs gefunden und behoben werden. Funktionalitäten wie das Updaten der App oder die Suche nach Veränderungen im Hintergrund sind noch nicht vollständig getestet und bedürfen noch einer genaueren Untersuchung. Die lokale Room-Datenbank befindet sich bislang noch in der ersten Normalform, da wäre eine Verbesserung von Vorteil. Des weiteren sollte die App in mehreren Sprachen, mindestens Englisch, angeboten werden, und die Kommunikation mit dem Server muss noch auf *https* umgestellt werden. Für all dies wurde mit dieser Arbeit aber eine gute Grundlage erstellt um die angesprochenen Themen einfacher umsetzten zu können.

Literatur

[1] Android for Developers. *Handling Lifecycles with Lifecycle-Aware Components*. 2021. URL: https://developer.android.com/topic/libraries/architecture/lifecycle.

[2] DomainFactory. *Kotlin vs. Java – was eignet sich wofür?* 2019. URL: https://www.df.eu/blog/kotlin-vs-java-was-eignet-sich-wofur/.

[3] GeeksForGeeks. *Suspend Function In Kotlin Coroutines*. 2020. URL: https://www.geeksforgeeks.org/suspend-function-in-kotlin-coroutines/.

[4] GeeksForGeeks. *withContext in Kotlin Coroutines*. 2020. URL: https://www.geeksforgeeks.org/withcontext-in-kotlin-coroutines.

[5] Jobtensor. *Moderne App-Entwicklung mit Kotlin - Übersicht und Vergleich der neuen Android-Entwicklungssprache mit Java*. 2022. URL: https://jobtensor.com/Skill/Kotlin-vs-Java-Tutorial.

[6] Atharva Kulkarni. *Dark Mode UI: the definitive guide*. 2020. URL: https://uxdesign.cc/dark-mode-ui-design-the-definitive-guide-part-1-color-53dcfaea5129.

[7] Thomas Theis. *Einstieg in Kotlin, Apps entwickeln mit Android Studio*. first. Rheinwerk Verlag, 2019.

[8] Android Wiki. *Kotlin*. 2021. URL: https://www.droidwiki.org/wiki/Kotlin.

[9] http://jacob yo.net/. *VIEWLIFECYCLEOWNER VS LIFECYCLEOWNER*. 2020. URL: http://jacob-yo.net/viewlifecycleowner-vs-lifecycleowner/.

Anhang A

XML-Files

A.1 Attribute

Codebeispiel A.1: Test

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="Theme">
        <attr name="colorOnPrimaryDark" format="reference" />

        <attr name="colorPrimaryLight" format="reference" />
        <attr name="colorOnPrimaryLight" format="reference" />

        <attr name="colorOnAccent" format="reference" />

        <attr name="colorPrimaryText" format="reference" />
        <attr name="colorSecondaryText" format="reference" />

        <attr name="colorDivider" format="reference" />

        <attr name="colorBackground" format="reference" />

        <attr name="colorChecked" format="reference" />
        <attr name="colorUnchecked" format="reference" />
        <attr name="themeName" format="reference" />
    </declare-styleable>


    <declare-styleable name="Status">
        <attr name="defaultStatusColor" format="color"/>
        <attr name="frueherVorschlag" format="color"/>
        <attr name="inDiskussion" format="color"/>
        <attr name="veroeffentlicht" format="color"/>
        <attr name="veraltet" format="color"/>
        <attr name="zukuenftigePlanung" format="color"/>
    </declare-styleable>

    <declare-styleable name="Favorit">
        <attr name="isFavorit" format="color"/>
        <attr name="isNotFavorit" format="color"/>
    </declare-styleable>
</resources>
```

A.2 Farben

A.2.1 Lightmode

Codebeispiel A.2: Test

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Theme 1 -->
    <color name="Theme1.colorPrimary">#009688</color>
    <color name="Theme1.colorOnPrimary">#fff</color>

    <color name="Theme1.colorPrimaryDark">#00796B</color>
    <color name="Theme1.colorOnPrimaryDark">#fff</color>

    <color name="Theme1.colorPrimaryLight">#B2DFDB</color>
    <color name="Theme1.colorOnPrimaryLight">#000</color>

    <color name="Theme1.colorAccent">#00BCD4</color>
    <color name="Theme1.colorOnAccent">#fff</color>

    <color name="Theme1.colorPrimaryText">#212121</color>
    <color name="Theme1.colorSecondaryText">#757575</color>

    <color name="Theme1.colorDivider">#BDBDBD</color>

    <color name="Theme1.colorBackground">#fff</color>
    <color name="Theme1.colorOnBackground">#000</color>

    <!-- Theme 2 -->
    <color name="Theme2.colorPrimary">#3F51B5</color>
    <color name="Theme2.colorOnPrimary">#fff</color>

    <color name="Theme2.colorPrimaryDark">#303F9F</color>
    <color name="Theme2.colorOnPrimaryDark">#fff</color>

    <color name="Theme2.colorPrimaryLight">#C5CAE9</color>
    <color name="Theme2.colorOnPrimaryLight">#000</color>

    <color name="Theme2.colorAccent">#607D8B</color>
    <color name="Theme2.colorOnAccent">#fff</color>

    <color name="Theme2.colorPrimaryText">#212121</color>
    <color name="Theme2.colorSecondaryText">#757575</color>

    <color name="Theme2.colorDivider">#BDBDBD</color>

    <color name="Theme2.colorBackground">#fff</color>
    <color name="Theme2.colorOnBackground">#000</color>

</resources>
```

A.2.2 Darkmode

Codebeispiel A.3: Test

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Theme 1 -->
```

```
4       <color name="Theme1.colorPrimary">#009688</color>
5       <color name="Theme1.colorOnPrimary">#ddd</color>
6
7       <color name="Theme1.colorPrimaryDark">#303030</color>
8       <color name="Theme1.colorOnPrimaryDark">#fff</color>
9
10      <color name="Theme1.colorPrimaryLight">#404040</color>
11      <color name="Theme1.colorOnPrimaryLight">#ddd</color>
12
13      <color name="Theme1.colorAccent">#00BCD4</color>
14      <color name="Theme1.colorOnAccent">#ddd</color>
15
16      <color name="Theme1.colorPrimaryText">#fff</color>
17      <color name="Theme1.colorSecondaryText">#ddd</color>
18
19      <color name="Theme1.colorDivider">#BDBDBD</color>
20
21      <color name="Theme1.colorBackground">#202020</color>
22      <color name="Theme1.colorOnBackground">#ddd</color>
23
24      <!-- Theme 2 -->
25      <color name="Theme2.colorPrimary">#3F51B5</color>
26      <color name="Theme2.colorOnPrimary">#ddd</color>
27
28      <color name="Theme2.colorPrimaryDark">#303030</color>
29      <color name="Theme2.colorOnPrimaryDark">#fff</color>
30
31      <color name="Theme2.colorPrimaryLight">#404040</color>
32      <color name="Theme2.colorOnPrimaryLight">#ddd</color>
33
34      <color name="Theme2.colorAccent">#607D8B</color>
35      <color name="Theme2.colorOnAccent">#ddd</color>
36
37      <color name="Theme2.colorPrimaryText">#fff</color>
38      <color name="Theme2.colorSecondaryText">#ddd</color>
39
40      <color name="Theme2.colorDivider">#BDBDBD</color>
41
42      <color name="Theme2.colorBackground">#202020</color>
43      <color name="Theme2.colorOnBackground">#ddd</color>
44
45  </resources>
```

A.3 Themes

Codebeispiel A.4: Test

```
1  <resources>
2      <!-- Base application theme. -->
3      <style name="Theme.AppTheme_1" parent="BaseTheme">
4          <item name="themeName">@string/Theme1_ThemeName</item>
5          <!-- Customize your theme here. -->
6          <item name="colorPrimary">@color/Theme1.colorPrimary</
               item>
7          <item name="colorOnPrimary">@color/Theme1.
               colorOnPrimary</item>
8
9          <item name="colorPrimaryDark">@color/Theme1.
               colorPrimaryDark</item>
```

```
10          <item name="colorOnPrimaryDark">@color/Theme1.
                colorOnPrimaryDark</item>
11
12          <item name="colorPrimaryLight">@color/Theme1.
                colorPrimaryLight</item>
13          <item name="colorOnPrimaryLight">@color/Theme1.
                colorOnPrimaryLight</item>
14
15          <item name="colorAccent">@color/Theme1.colorAccent</
                item>
16          <item name="colorOnAccent">@color/Theme1.colorOnAccent<
                /item>
17
18          <item name="colorPrimaryText">@color/Theme1.
                colorPrimaryText</item>
19          <item name="colorSecondaryText">@color/Theme1.
                colorSecondaryText</item>
20
21          <item name="colorDivider">@color/Theme1.colorDivider</
                item>
22
23          <item name="colorBackground">@color/Theme1.
                colorBackground</item>
24          <item name="colorOnBackground">@color/Theme1.
                colorOnBackground</item>
25
26          <item name="colorChecked">@color/Theme1.colorPrimary</
                item>
27          <item name="colorUnchecked">@color/Theme1.colorAccent</
                item>
28
29          <item name="actionMenuTextColor">@color/Theme1.
                colorOnPrimaryDark</item>
30
31      </style>
32
33      <style name="Theme.AppTheme_2" parent="BaseTheme">
34          <item name="themeName">@string/Theme2_ThemeName</item>
35          <!-- Customize your theme here. -->
36          <item name="colorPrimary">@color/Theme2.colorPrimary</
                item>
37          <item name="colorOnPrimary">@color/Theme2.
                colorOnPrimary</item>
38
39          <item name="colorPrimaryDark">@color/Theme2.
                colorPrimaryDark</item>
40          <item name="colorOnPrimaryDark">@color/Theme2.
                colorOnPrimaryDark</item>
41
42          <item name="colorPrimaryLight">@color/Theme2.
                colorPrimaryLight</item>
43          <item name="colorOnPrimaryLight">@color/Theme2.
                colorOnPrimaryLight</item>
44
45          <item name="colorAccent">@color/Theme2.colorAccent</
                item>
46          <item name="colorOnAccent">@color/Theme2.colorOnAccent<
                /item>
47
48          <item name="colorPrimaryText">@color/Theme2.
                colorPrimaryText</item>
```

```
49          <item name="colorSecondaryText">@color/Theme2.
                colorSecondaryText</item>
50
51          <item name="colorDivider">@color/Theme2.colorDivider</
                item>
52
53          <item name="colorBackground">@color/Theme2.
                colorBackground</item>
54          <item name="colorOnBackground">@color/Theme2.
                colorOnBackground</item>
55
56          <item name="actionMenuTextColor">@color/Theme2.
                colorOnPrimaryDark</item>
57      </style>
58
59      <style name="BaseTheme" parent="Theme.AppCompat.DayNight.
            NoActionBar">
60          <item name="frueherVorschlag">#f50</item>
61          <item name="inDiskussion">#ff0</item>
62          <item name="veroeffentlicht">#0a0</item>
63          <item name="veraltet">#333</item>
64          <item name="zukuenftigePlanung">#FFFFFF</item>
65          <item name="defaultStatusColor">#000</item>
66          <item name="spinnerDropDownItemStyle">@style/
                spinnerDropDownItemStyle</item>
67          <item name="isFavorit">#f00</item>
68          <item name="isNotFavorit">#0f0</item>
69      </style>
70
71      <!-- Alert Dialog -->
72      <style name="AlertDialog.Filter" parent="@style/
            ThemeOverlay.MaterialComponents.MaterialAlertDialog">
73          <!-- Background Color-->
74          <item name="android:background">?attr/colorPrimaryDark<
                /item>
75          <!-- Text Color for title and message -->
76          <item name="colorOnSurface">?attr/colorOnPrimaryDark</
                item>
77          <item name="android:textColor">?attr/colorOnPrimaryDark
                </item>
78          <item name="android:layout_width">wrap_content</item>
79          <!-- Style for positive button -->
80          <item name="buttonBarPositiveButtonStyle">@style/
                PositiveButtonStyle</item>
81          <!-- Style for negative button -->
82      </style>
83
84
85      <style name="OptionenFragment.Entry" parent="Base.CardView"
            >
86          <item name="android:background">@drawable/
                rounded_color_border</item>
87          <item name="android:backgroundTint">?attr/
                colorPrimaryDark</item>
88          <item name="android:textColor">?attr/colorOnPrimaryDark
                </item>
89          <item name="android:padding">10dp</item>
90          <item name="android:layout_marginLeft">5dp</item>
91          <item name="android:layout_marginRight">5dp</item>
92          <item name="android:layout_marginBottom">2dp</item>
93          <item name="android:layout_weight">1</item>
94          <item name="android:layout_width">match_parent</item>
```

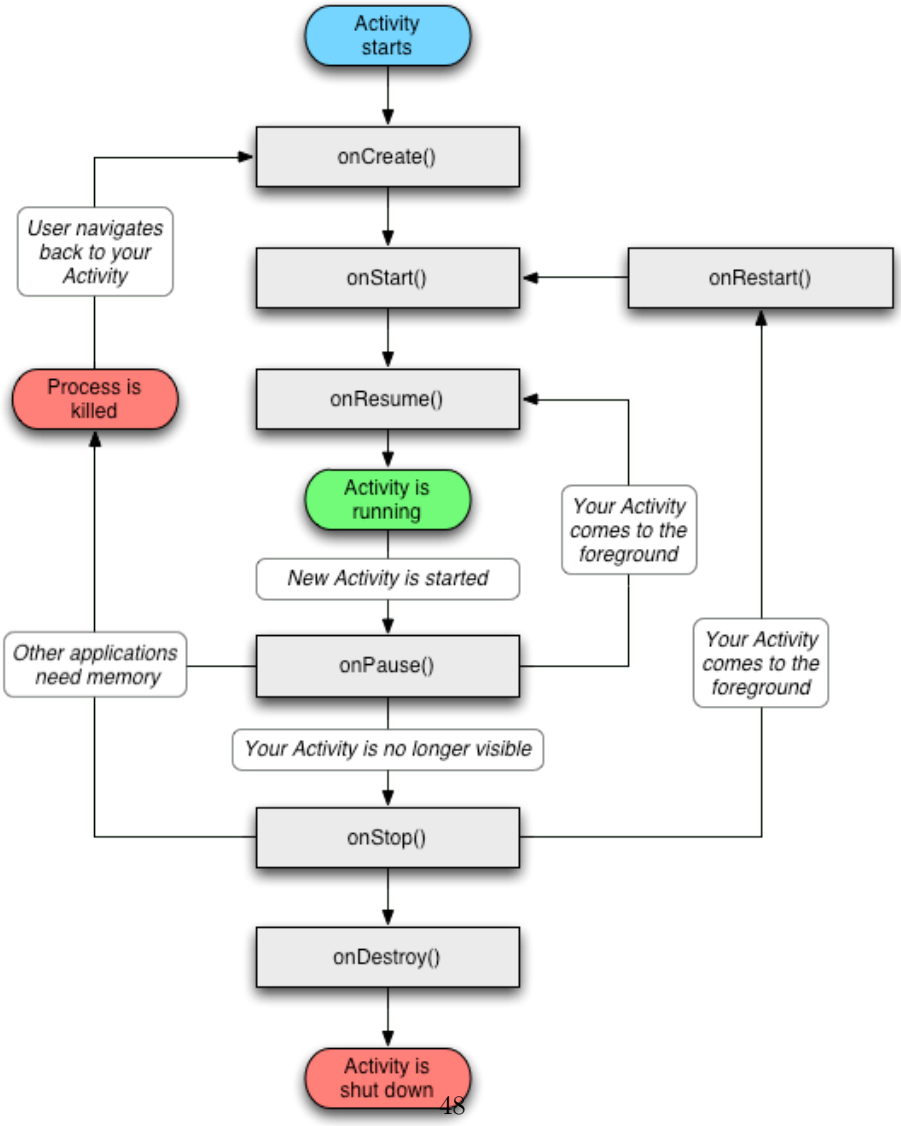
```
95          <item name="android:layout_height">70dp</item>
96          <item name="android:textSize">18sp</item>
97          <item name="textAllCaps">false</item>
98          <item name="android:textAlignment">textStart</item>
99      </style>
100
101     <style name="AlertDialog.Filter.Spinner" parent="Widget.
            AppCompat.Spinner.DropDown">
102         <item name="android:layout_height">50dp</item>
103         <item name="android:layout_width">200dp</item>
104         <item name="android:textColor">?attr/colorOnPrimary</
                item>
105         <item name="android:spinnerMode">dialog</item>
106     </style>
107
108     <style name="AlertDialog.Filter.CheckBox.TextView" parent="
            Widget.AppCompat.TextView">
109         <item name="android:layout_width">match_parent</item>
110         <item name="android:layout_height">wrap_content</item>
111         <item name="android:gravity">center</item>
112         <item name="android:textSize">20sp</item>
113         <item name="android:textColor">?attr/colorOnPrimaryDark
                </item>
114     </style>
115
116     <style name="AlertDialog.Filter.CheckBox.Box" parent="
            Widget.AppCompat.CompoundButton.CheckBox">
117         <item name="android:layout_width">wrap_content</item>
118         <item name="android:layout_height">wrap_content</item>
119         <item name="android:buttonTint">?attr/
                colorOnPrimaryDark</item>
120     </style>
121
122     <style name="AlertDialog.Filter.Divider">
123         <item name="android:layout_width">match_parent</item>
124         <item name="android:layout_height">2dp</item>
125         <item name="android:backgroundTint">?attr/
                colorOnPrimaryDark</item>
126         <item name="android:layout_marginTop">10dp</item>
127         <item name="android:layout_marginBottom">10dp</item>
128
129     </style>
130
131     <style name="PositiveButtonStyle" parent="@style/Widget.
            MaterialComponents.Button">
132         <!-- text color for the button -->
133         <item name="android:textColor">?attr/colorOnPrimary</
                item>
134         <!-- Background tint for the button -->
135         <item name="backgroundTint">?attr/colorPrimary</item>
136     </style>
137
138     <style name="spinnerItemStyle" parent="android:Widget.
            TextView.SpinnerItem">
139         <item name="android:background">?attr/colorPrimaryDark<
                /item>
140         <item name="android:editTextColor">?attr/
                colorOnPrimaryDark</item>
141     </style>
142     <style name="ThemeOverlay.SearchView" parent="">
143         <!-- Text color -->
```

```
144         <item name="android:editTextColor">?attr/
                colorOnPrimaryDark</item>
145     </style>
146     <style name="spinnerDropDownItemStyle" parent="
            android:TextAppearance.Widget.TextView.SpinnerItem">
147         <item name="android:background">?attr/colorPrimaryDark<
                /item>
148         <item name="android:editTextColor">?attr/
                colorOnPrimaryDark</item>
149     </style>
150
151     <style name="AppTheme.AppBarOverlay" parent="ThemeOverlay.
            AppCompat.Dark.ActionBar" />
152
153     <style name="AppTheme.PopupOverlay" parent="ThemeOverlay.
            AppCompat.Light" />
154
155     <style name="ProgressStyle" parent="Theme.AppCompat.Light.
            Dialog.Alert">
156         <item name="colorAccent">?attr/colorPrimary</item>
157     </style>
158
159     <style name="AppTheme.RatingBar" parent="Theme.AppCompat">
160         <item name="android:progressTint">?attr/colorAccent</
                item>
161         <item name="android:progressBackgroundTint">?attr/
                colorPrimaryDark</item>
162         <item name="android:secondaryProgressTint">?attr/
                colorPrimary</item>
163     </style>
164
165     <style name="AutoCompleteStyle">
166         <item name="colorAccent">?attr/colorAccent</item>
167     </style>
168
169     <style name="customAlertDialog" parent="@android:style/
            Theme.Dialog">
170         <item name="android:background">?attr/colorPrimaryDark<
                /item>
171         <item name="textColorAlertDialogListItem">?attr/
                colorOnPrimaryDark</item>
172         <item name="android:textSize">20dp</item>
173         <item name="android:windowNoTitle">true</item>
174     </style>
175
176     <style name="OptionenFragment"></style>
177
178     <string name="Theme1_ThemeName">Green</string>
179     <string name="Theme2_ThemeName">Blue</string>
180
181 </resources>
```

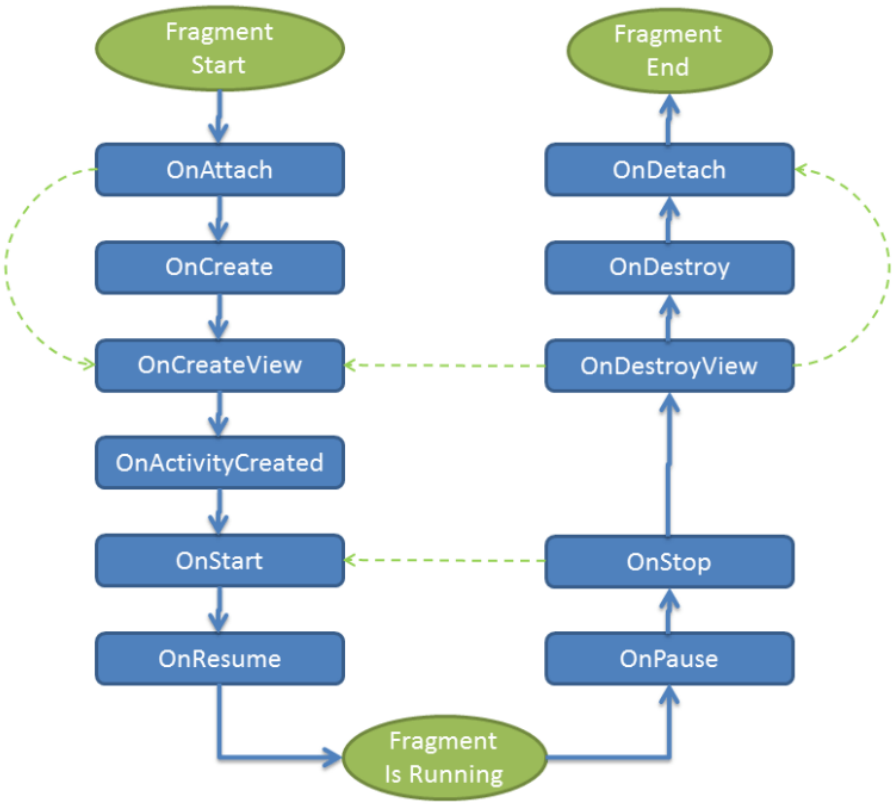
Anhang B

Lifecycles

B.1 Activity-Lifecycle



B.2 Fragment-Lifecycle



Anhang C

Backend

C.1 Filter

Codebeispiel C.1: Test

```
1  package com.Fachhochschulebib.fhb.pruefungsplaner.utils
2
3  import android.util.Log
4  import androidx.lifecycle.LiveData
5  import androidx.lifecycle.MutableLiveData
6  import com.Fachhochschulebib.fhb.pruefungsplaner.model.room.
       TestPlanEntry
7  import java.text.SimpleDateFormat
8  import java.util.*
9
10 /**
11  * Inner Class to filter the table of moduls. Used by
        TermineFragment-fragment and FavoritenFragment-fragment.
12  *
13  * @author Alexander Lange
14  * @since 1.6
15  * @see Terminefragment
16  * @see Favoritenfragment
17  */
18 object Filter {
19     /**
20      * Parameter to Filter with the Modulename.
21      * Calls the [onModuleNameChangedListener] and the [
            onFilterChangedListener].
22      *
23      * @author Alexander Lange
24      * @since 1.6
25      * @see onModulNameChangedListener
26      * @see onFilterChangedListener
27      */
28     var modulName: String? = null
29         set(value) {
30             field = value
31             filterChanged()
32         }
33
34     /**
35      * Parameter to Filter with the Coursename.
```

```
36          * Calls the [onCourseNameChangedListener] and the [
                onFilterChangedListener].
37          *
38          * @author Alexander Lange
39          * @since 1.6
40          * @see onCourseNameChangedListener
41          * @see onFilterChangedListener
42          */
43         var courseName: String? = null
44             set(value) {
45                 field = value
46                 filterChanged()
47             }
48
49         /**
50          * Parameter to Filter with a specific date.
51          * Calls the [onDateChangedListener] and the [
                onFilterChangedListener].
52          *
53          * @author Alexander Lange
54          * @since 1.6
55          * @see onDateChangedListener
56          * @see onFilterChangedListener
57          */
58         var datum: Date? = null
59             set(value) {
60                 field = value
61                 filterChanged()
62             }
63
64         /**
65          * Parameter to filter with a specific examiner.
66          * Calls the [onExaminerChangedListener] and [
                onFilterChangedListener].
67          * @author Alexander Lange
68          * @since 1.6
69          * @see onExaminerChangedListener
70          * @see onFilterChangedListener
71          */
72         var examiner: String? = null
73             set(value) {
74                 field = value
75                 filterChanged()
76             }
77
78         /**
79          * Array of 6 semester, where each field contains a boolean
                , if the semester is selected (true), or not (false)
80          *
81          * @author Alexander Lange
82          * @since 1.6
83          */
84         var semester: Array<Boolean> = arrayOf(true, true, true,
            true, true, true)
85             set(value) {
86                 return
87             }
88
89         /**
90          * Public method to set the value for a specific semester.
91          * Calls the [onSemesterChangedListener] and the [
                onFilterChangedListener]
```

```
92          * @param[pSemester] The semester to set the value.
93          * @param[active] If the semester is checked or not.
94          * @author Alexander Lange
95          * @since 1.6
96          * @see onSemesterChangedListener
97          * @see onFilterChangedListener
98          *
99          */
100         fun SetSemester(pSemester: Int, active: Boolean) {
101             semester[pSemester] = active
102             filterChanged()
103         }
104
105         /**
106          * Invokes every Method, appended to the
                onFilterChangedListener.
107          *
108          * @author Alexander Lange
109          * @since 1.6
110          * @see onFilterChangedListener
111          */
112         private fun filterChanged() {
113             for (i in onFilterChangedListener) {
114                 i.invoke()
115             }
116         }
117
118         /**
119          * Validates a testplanentry-Object. Checks if all Filter-
                values agree with the given entry.
120          *
121          * @param[context] Current context
122          * @param[entry] The Entry that needs to be validated
123          * @return true->The entry agrees with the filter,false->
                the entry does not agree with the filter
124          * @author Alexander Lange
125          * @since 1.6
126          */
127         fun validateFilter(entry: TestPlanEntry?): Boolean {
128             if (entry == null) {
129                 return false
130             }
131             if (modulName != null && entry.module?.lowercase()?.
                    startsWith(
132                         modulName?.lowercase() ?: entry.module?.
                                lowercase() ?: "-1"
133                     ) == false
134             ) {
135                 return false
136             }
137             if (entry.course?.lowercase()?.startsWith(
138                         courseName?.lowercase() ?: entry.course?.
                                lowercase() ?: "-1"
139                     ) == false
140             ) {
141                 return false
142             }
143             if (datum != null) {
144                 val sdf = SimpleDateFormat("yyyy-MM-dd")
145                 val date = sdf.parse(entry.date)
146                 if (!datum!!.atDay(date)) {
147                     return false
```

```
148                 }
149             }
150             if (entry.firstExaminer?.lowercase()?.startsWith(
151                     examiner?.lowercase() ?: entry.firstExaminer?.
                            lowercase() ?: "-1"
152                 ) == false
153             ) {
154                 return false
155             }
156             if (entry.semester != null && !semester[entry.semester
                    !!.toInt().minus(1)]) {
157                 return false
158             }
159             return true
160         }
161 
162         fun validateList(list: List<TestPlanEntry>): List<
                TestPlanEntry> {
163             val ret = mutableListOf<TestPlanEntry>()
164             list.forEach {
165                 if (validateFilter(it)) {
166                     ret.add(it)
167                 }
168             }
169             return ret
170         }
171 
172         fun validateList(liveData: LiveData<List<TestPlanEntry>?>):
                 LiveData<List<TestPlanEntry>?> {
173             val list = liveData.value
174             val filtered = list?.let { validateList(it) }
175             val ret = MutableLiveData<List<TestPlanEntry>>()
176             ret.postValue(filtered)
177             return ret
178         }
179 
180         /**
181          * Resets the Filter, sets every value to null.
182          * Calls the onResetListener.
183          *
184          * @author Alexander Lange
185          * @since 1.6
186          */
187         fun reset() {
188             courseName = null
189             modulName = null
190             datum = null
191             semester.fill(true)
192             filterChanged()
193         }
194         var onFilterChangedListener: MutableList<() -> Unit> =
                mutableListOf()
195 }
```