RxAndroid- Reaktive Programmierung unter Android

Katharina Schwab, Seminar Neue Technologien, Angewandte Informatik, Hochschule Offenburg

***Abstract*—Reaktive Programmierung hat sich in den letzten Jahren immer weiterverbreitet. Als erstes Werk hierzu wird das von Microsoft für das .NET-Umfeld entwickelte Reaktive Extensions Framework gezählt. Daraus wurden alsbald weitere Bibliotheken und Frameworks abgleitet, die das gegebene Prinzip in andere Sprachen portierten. Daraus entstand das Open Source Projekt reactiveX. Hierbei wurde das Observer Pattern der Gang of four um einige Funktionen erweitert, so dass damit sehr leicht das reaktive Programmierparadigma unterstützen werden kann. Es finden sich im reactiveX Projekt viele Bibliotheken für etliche Sprachen, die alle das Ziel haben die reaktive Programmierung zu unterstützen. Auf GitHub sind diese alle frei verfügbar. Eine dieser Bibliothek ist RxJava, auf welcher die hier behandelte Bibliothek RxAndroid aufsetzt. In RxAndroid wurde RxJava um einige Funktionen erweitert, die speziell für Googels Betriebssystem Android entwickelt wurden. Um Verständnisgrundlagen zu schaffen werden neben RxAndroid und reactiveX auch die reaktive Programmierung sowie das Betriebssystem Android und insbesonder dessen Architektur vorgestellt. Es wird auch auf das reaktive Manifest eingegangen um die Grundprinzipien und Ziele der reaktiven Programmierung zu erläutern.**

***Index Terms*—Android, reactive programming, reactive extensions, RxJava, RxAndroid, Observer Pattern, reactive manifesto**

I. Einleitung

DIESES Paper gliedert sich in fünf Bereiche. Zu Beginn werden die für das Verständnis benötigte Grundlagen erläutert. Diese sind zum einen die reaktive Programmierung als solche, ihr Herkunft, ihr Nutzen und ihre grundlegenden Prinzipien. Zum gleichen Zweck wird das reaktive Manifest kurz vorgestellt und erläutert. Das Observer Pattern sowie Callbacks und Eventhandler werden kurz besprochen, um einen größeren Überblick zu ermöglichen.
Im nächsten Abschnitt wird das für mobile Geräte weit verbreitetet Betriebssystem Android der Firma Google vorgestellt. Besonderer Augenmerk wird hierbei auf die Architektur und den primären Aufbau einer Anwendung unter Android gelegt.
Der dritte Abschnitt befasst sich mit dem Open Source Projekt reactiveX, welches Bibliotheken in vielen Programmiersprachen zur Verfügung stellt. Diese möchten die reaktive Programmierung vereinfachen. Dazu wurde das Observer Pattern erweitert. Ein Vergleich mit dem Iterator Pattern diente als Zielsetzung. Es werden die Hauptbestandteile der Bibliotheken erklärt, wobei es in den diversen Sprachen immer wieder kleiner Abweichungen gibt. Diese Hauptbestandteile sind das Observable mit seinen Standardmethoden, die Operationen, die zur Verfügung gestellt werden und das Prinzip der Scheduler.
Im Anschluss wird die Verwendung von RxAndroid anhand von Beispielen detailliert erläutert. Abschließend wird die Bibliothek RxAndroid bewertet.

II. Reaktive Programmierung

Das Programmierparadigma der reaktiven Programmierung entwickelte sich aus der funktionalen Programmierung, weshalb man auch den Begriff der funktionalen reaktiven Programmierung dafür verwendet. Die „funktionale Programmierung wird oft als Ausdruck dessen beschrieben, *was* berechnet wird und nicht *wie*“ [1] Die reaktive Programmierung „wurde erstmals eingeführt in der Sprache Haskell“ [2]. Es handelt sich um eine datenflußgesteuerte und asynchrone Art des Programmierens.
Zudem soll es „ein Ersatz für das weit verbreitete Observer Pattern, auch bekannt als Listeners oder Callbacks,“ [3] sein. Bekanntheit erlangte dieses Konzept durch Microsofts Reactive Extensions für .NET, welches von Netflix eingesetzt wurden. Daraufhin wurde dieses Konzept auch in andere Sprachen übertragen. Im Rahmen des Projektes reactiveX wurde es bereits in viele Sprachen, darunter Java, Javascript, Clojure und Scala, transformiert. Weitere sollen noch folgen.
Drei Hauptkonzepte wurden definiert: „time-changing values, tracking of dependencies, and automated propagation of changes“ [4]. Gemeint ist damit, dass Änderungen an einem Wert *a* automatisch weitergegeben werden an alle von a abhängigen Werte *b -n.* Es wird somit vermieden ein eigenes Überwachungs- und Updatekonstrukt zu implementieren. Diese deklarativ erstellten Abhängigkeiten werden zur Laufzeit aufgelöst und korrekt verarbeitet. „Aus Sicht des Entwicklers geschieht das Weiterleiten von Änderungen automatisch“ [5].
Ein weiterer, sehr wichtiger Aspekt dieses Paradigmas ist die asynchrone Verarbeitung von Datenströmen. Ein Datenstrom ist ein fortlaufender Strom an einzelnen, zeitlich geordneten

Ereignissen. Diese Streams können Werte, Errors oder „completed signals“ (definitives Ende) beinhalten. Ein gutes Beispiel ist eine grafische Benutzerschnittstelle (GUI). Mittels dieser werden fortlaufend Ereignisse wie Mausbewegungen, Klicks oder Eingaben generiert. Das Schließen des Fensters wäre ein „completed signal“ In der reaktiven Programmierung ist eine Verarbeitung „just-in-time“ vorgesehen. Die Ereignisse werden an unterschiedliche asynchron laufende Ströme weitergereicht, welche unterschiedliche Funktionalitäten aufweisen. Somit werden die Signalquellen mit Transformatoren verbunden. Anders gesagt, es wird eine Pipeline von Operationen zur Datenverarbeitung definiert. Ein Strom kann zum Beispiel in Abhängigkeit der Klicks die GUI-Ansicht verändern, während ein anderer Strom zeitgleich die Eingaben an einen Server oder eine Datenbank weiterleitet. Das Ziel ist es in verteilten Systemen mit sehr großen Datenmenge immer sehr schnelle Antwortzeiten zu erreichen und gleichzeitig zu 100% verfügbar zu sein. Ein Blockieren des Systems soll unter allen Umständen vermieden werden.

A. Das Reaktive Manifest

Dieses Manifest[1] wurde unter anderem von Jonas Bonér, Dave Farley, Roland Kuhn und Martin Thompson im Jahr 2013 verfasst. Über 22.000 Entwickler weltweit haben es bereits unterzeichnet und damit ihre Zustimmung signalisiert. Da immer weniger Anwendungen oder Systeme gebaut werden, die komplett unabhängig von anderen Systemen sind, ergibt es Sinn sich untereinander, ohne konkrete Verträge oder Absprachen, auf gemeinsame Ziele zu konzentrieren.
In diesem Dokument wird ein reaktives System an 4 Merkmalen festgemacht:

1. Antwortbereit („responsive“)
   Dies bedeutet, dass „das System unter allen Umständen zeitgerecht antwortet, solange dies überhaupt möglich ist“ [6] Die Antwortbereitschaft wird als Kernfunktion eines jeden reaktiven Systems angesehen.
2. Widerstandsfähig („resilient“)
   Selbst bei Ausfällen in Hard- oder Software soll die Kernfunktionalität, also die Antwortbereitschaft, noch immer gegeben sein. Der Ausfall eines Teilsystems soll keinerlei Auswirkungen auf die anderen Systemteile haben. Das Beheben des Fehlers wird einer übergeordneten Instanz übertragen, welche einzelne Komponenten replizieren kann um die Funktionalität zu gewährleisten. Keinesfalls soll ein Benutzer etwas von einem Ausfall bemerken.
3. Elastisch(„elastic“)
   Das System bleibt auch bei Lastschwankungen antwortbereit. Auch hier wird mit dem Replizieren von Komponenten verhindert, dass die Funktionalität eingeschränkt wird. Dies bedeutet auch, dass ein reaktives System in seiner Architektur keine Engpässe haben darf.
4. Nachrichtenorientiert („message driven“)
   Die Kommunikation zwischen den einzelnen Komponenten erfolgt mittels asynchroner, nicht-blockierender Nachrichtenübermittlung. Dadurch wird auch eine Ortsunabhängigkeit erreicht. Die Komponenten sind hinsichtlich Fehlern isoliert und entkoppelt.

Man möchte „diesen Ansatz bewusst zu Grunde legen, anstatt ihn in Teilen für jedes Projekt neu zu entdecken“ [6]

B. Observer Pattern, Event Listener und Callbacks

1. Observer Pattern

Das Observer Pattern (Beobachtermuster) gehört zu den Verhaltensmustern. Es ermöglicht es, dass ein Objekt seinen Zustand ändert in Abhängigkeit des Zustandes eines anderen Objekts. Hierbei werden weder die Variante Push noch Polling eingesetzt. Stattdessen werden die Objekte mittels Schnittstellen und Vererbung gekoppelt.

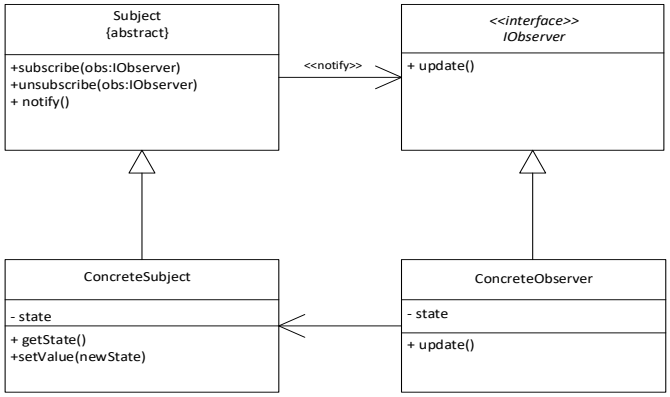

Abbildung 1, UML-Diagramm zum Observer Pattern

2. Event Listener

Im Grunde sind Event listener auch Observer. Allerdings werden sie oftmals spezifischer eingesetzt. Es wird beispielsweise nicht die gesamte GUI überwacht, sondern es wird nur auf spezielle Events, zum Beispiel Mausklicks, gelauscht.
Wie sich dies im konkreten Fall verhält ist rein von der Implementierung der verwendeten Sprache bzw. des Frameworks abhängig. In Java zum Beispiel sind Listener Teil eines Observer Patterns, in .NET nicht.

3. Callbacks

Eine Callbackmethode informiert den Aufrufer über das Beenden einer von ihm angestoßenen Operation. Ein Observer oder ein Listener informiert alle eingeschriebenen Objekte über eine Zustandsänderung, z. B. das Beenden einer Methode. Zudem arbeiten Callbacks im Unterschied zu Observern oder Listenern in der Regel asynchron.
Beispiele sind Methoden in der Form von *CallerObjekt.onStart()* oder *CallerObject.onError()*. Solche Callbacks sind in vielen Bereichen zu finden.

[1] Definition laut Duden: Öffentlich dargelegtes Programm; Bekanntmachung, Statement

III. Android

A. *Herkunft*

Für die Entwicklung auf mobile Geräten steht neben Apples IOS vor allem Android zur Verfügung. Der Großteil der aktuellen Smartphones läuft auf dem von Google weiterentwickelten Android Betriebssystem. Dieses wurde von Andy Rubin in seiner gleichnamigen Firma Android Inc. entwickelt und sollte ursprünglich ein Betriebssystem für Digitalkameras werden. Dieser Plan wurde aber verworfen und man wollte mit dem von Microsoft entwickelten Windows Mobile konkurrieren. Google kaufte die Firma 2005 und stellt den Erfinder ein. Im Jahr 2008 wurde Android offiziell eingeführt und steht seitdem jedem Hersteller von Smartphones frei zur Verfügung. Die Open Handset Alliance, bestehend auf Google und diversen anderen Tech-Firmen, entwickelt dieses Betriebssystem kontinuierlich weiter. Auch das Ökosystem von Android gedeiht. Es gibt unter anderem Projekte in den Bereichen Automotive, TV und der tragbaren Endgeräte.

B. *Architektur*

Wie die Namensgebung der Versionen, die immer nach einer Süßigkeit benannt sind, vielleicht schon vermuten lässt ist der Ausgangspunkt der Entwicklung der Linuxkernel. Bis zu Version 4.3 (“KitKat”), war die auf Java basierende Dalvik Virtuell Machine ein zentraler Bestandteil zur Bytecodeausführung. Mittlerweile gibt es eine eigene Laufzeitumgebung “ART”, welche die Dalvik-VM abgelöst hat.

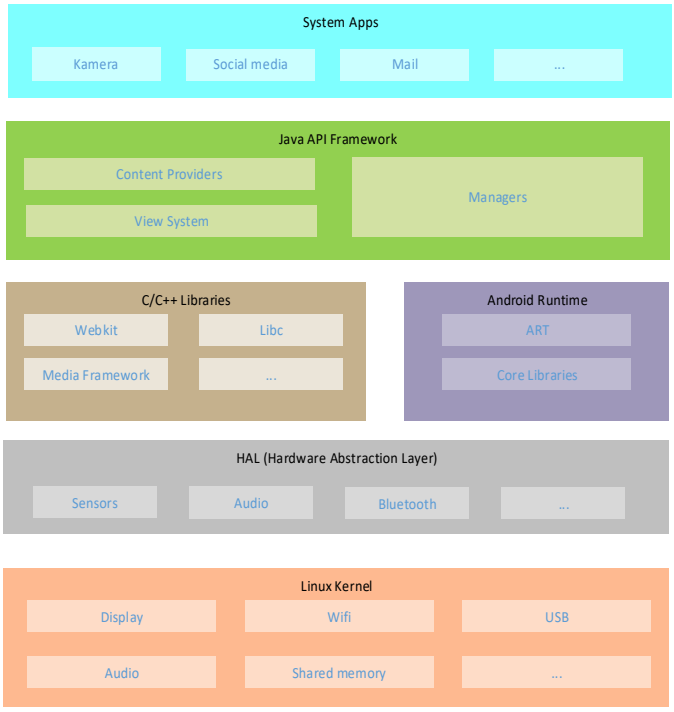

Abbildung 2, Architektur des Betriebssystems Android

C. *Aufbau einer Android App*

Unter Android kann in unterschiedlichen Sprachen entwickeln werden, beispielsweise in Java, Kotlin oder C++. Das Betriebssystem selbst verwaltet alle Apps. Jede wird als eigener User angesehen und bekommt eine eigene Linux User ID.

Alle benötigten Dateien zur Installation müssen in einer Datei mit der Endung *.apk*, der Installationsdatei, vorhanden sein.

Jede Android App besteht aus diversen Komponenten. Diese müssen in der zentralen Datei *Androidmanifest.xml* deklariert werden. Diese Datei wird beim Start der App vom System ausgelesen. Das System startet dann die benötigten Komponenten und instanziiert alle erforderlichen Klassen. Eine App hat keinen „Single Entry Point“, also keine Main()-Funktion, wie aus vielen anderen Programmiersprachen bekannt. Die 4 wichtigsten Komponenten sind

1. Activities
   Hierbei handelt es sich um ein Fenster beziehungsweise eine GUI-Oberfläche, mit der der User interagieren kann.
2. Services
   Ein Service läuft im Hintergrund ab. Er blockiert die Activity nicht. Beispiele sind das Abspielen von Musik im Hintergrund, während der User auch in einer anderen App sein kann. Auch das Senden von Daten an einen Server kann als Background Service ablaufen, hiervon bekommt der User nichts mit.
3. Broadcast receiver
   Dieser kann systemweite Nachrichten versenden und ermöglicht es einer App auf solch eine Nachricht zu antworten oder anderweitig zu reagieren.
4. Content provider
   Er ist zuständig für das Managen von appübergreifenden Dateien.

Apps können sich gegenseitig starten. Hierbei wird der eigentliche Start von System übernommen und die gestartete App erhält ihren eigenen Prozess. Ein beliebtes Beispiel hierfür ist das Fotografieren innerhalb einer App über die auf dem Gerät befindliche Kamera App. Eine App, zum Beispiel Facebook, schickt dem System eine Nachricht, dass die Kamera benötigt wird. Den Start übernimmt das System selbst. In der Facebook App wirkt es nun so, als sei die Kamera Teil der App. Tatsächlich wurde ein neuer Prozess angelegt und das entstandene Foto wird der Facebook App lediglich zur Verfügung gestellt. Auch das Beenden des Kameraprozesses übernimmt das Betriebssystem.

IV. Rx – Reactive Extensions

Das Ziel ist es die Entwicklung von asynchronen und eventbasierten, also reaktiven, Programmen zu unterstützen. „Es befasst sich mit low level-Programmierproblemen wie Threading, Synchronisation, Threadsicherheit und gleichzeitigen Zugriffen auf Datenstrukturen“ [7].

A. *Observable & Observer*

Um mit Rx arbeiten zu können muss man sich zuerst in dessen Grundlagen einarbeiten, welche bei allen programmiersprachenspezifischen Erweiterungen in etwa gleich sind.

Rx versteht sich als eine Erweiterung des Observer Patterns um einige Funktionen, die das reaktive Programmieren ermöglichen sollen.

Als erstes sei hier das Prinzip des Observables erklärt. Es produziert sogenannte Items. Dies sind die Werte oder Objekte, welche asynchron verarbeitet werden sollen. Ein Observer kann sich bei einem oder mehreren Observables registrieren und erhält dann die gesendeten Items, welche er sinnvoll weiterverarbeiten wird. Konkret bedeutet das, dass das Observable-Objekt eine Methode des Observers aufruft sobald neue Daten zur Verfügung stehen. Zu unterscheiden sind hierbei noch „heiße“ und „kalte“ Observables. Ein „heißes“ Observable beginnt mit dem Versenden von Items sobald es erzeugt wurde. Ein „kaltes“ Observable hingegen versendet erst wenn mindestens ein Observer sich registriert hat. Somit wird hierbei sichergestellt, dass alle Items den Observer erreichen, wohingegen beim „heißen“ Observable durchaus auch Items ins Nichts gesendet werden.
Des Weiteren gibt es noch das Subject, welches zugleich Observable und Observer ist. Hier gibt es noch diverse Untervarianten.
Das Prinzip des Singels steht nur in einigen Sprachen, unter anderem RxJava, zur Verfügung. Dieses besondere Observable sendet einmalig ein Item oder eine Fehlermeldung.

B. onNext(), onError(), onComplete()

Die Konventionen sehen vor, dass die Methoden *onNext(), onError()* und *onComplete()* oder Teilmengen davon vom Observer implementiert werden.
Die Methode *onNext()* wird beim Versenden eines jeden Items aufgerufen. Hierin findet die eigentliche Verarbeitung des Items statt.
Die *onError*()-Methode wird einmalig aufgerufen sollte im Observable ein Fehler auftreten. Diese Methode beendet somit das Observable. Nachfolgend werden keine weiteren Methodenaufrufe mehr folgen, also keine weiteren Items gesendet.
OnComplete() beendet ein Observable ordnungsgemäß. Auch hier erfolgen keine weiteren Methodenaufrufe kein weiterer Versand.

C. Scheduler

Per Default ist Rx auf einen Thread ausgelegt. Das bedeutet die Benachrichtigungen vom Observable werden in dem Thread ausgeführt, in dem auch das Registrieren aufgerufen wurde. Mit Hilfe des Schedulers lässt sich bestimmen wieviel Threads erzeugt werden sollen und was in welchem ablaufen soll. Hierzu werden die Methoden *SubscribeOn()* und *ObserveOn*() verwendet. Beide erhalten einen Scheduler als Übergabeparameter. Dieser definiert in seiner Implementation was genau passieren soll. Man hat beispielsweise die Möglichkeit einen Threadpool zu erzeugen und zu verwenden oder auch nur einen neuen Thread für dieses Observable bzw. diesen Observer anzulegen.

D. Operationen

Es werden sehr viele unterschiedliche Funktionen von Rx zur Verfügung gestellt.
In der offiziellen Dokumentation [8] wird eine Unterteilung in 11 Kategorien vorgenommen.
Die Wichtigsten sind:

1. *Erstellen eines Observables*
   Neben der zu erwartenden *Create()*-Methode stehen auch noch Methoden wie beispielsweise *From(), Just()* oder *Defer()* zur Verfügung.
   From() wandelt ein bestehendes Objekt oder eine Datenstruktur in ein Observable um.
   Just() erstellt ein Observable, welches das oder die übergebenen Objekte als Items versendet. Zum Beispiel *Just(„a“, „b“, „c“);* gibt ein Observable zurück das genau die Items mit den Werten a, b und c versendet.
   Defer() sorgt dafür, dass das gewünschte Observable erst angelegt wird, sobald sich ein Observer registriert. Jeder Observer bekommt hier ein eigenes, neues Observable.
2. *Transformieren der Items eines Observables*
   Erwähnenswert sind hier die Funktionen *FlatMap(), Map*() und *Buffer().*
   Erstere wandelt die Items eines Observables in neue Observables um und fügt alle Items die diese senden zu einem einzigen Observable-Objekt zusammen.
   Map() transformiert die Items mittels einer gegebenen Funktion. So ist es zum Beispiel möglich aus Items des Datentyps char oder string Integerwerte zu generieren.
   Buffer() sammelt alle gesendeten Items und gibt sie nach einer periodisch angegebenen Zeit gebündelt weiter.
3. *Filtern von Items eines Observables*
   Beispielhafte Methoden sind *Filter(), Last()* oder *Distinct().*
   Filter() gibt nur Items weiter die eine definierte Eigenschaft aufwiesen,
   Last() reicht nur das letzte vom Observable gesendete Item weiter.
   Distinct() sortiert doppelte Items aus.
4. *Kombinieren von Observables*
   Diese Operationen erlauben es die Items von mehreren Observables zu einem einzigen Observable zu vereinen.

Des Weiteren gibt es Methoden die boolesche Algebra, mathematische Operationen und Fehlerbehandlung ermöglichen. Auch allgemeine Funktionen zum Arbeiten mit Observables stehen zur Verfügung. Hierzu zählt die *Subscribe()*-Methode zum Registrieren eines Observers auf ein Observable oder auch eine *Timeout()*-Methode.

V. RxAndroid

Es ist naheliegend die reaktive Programmierung in Android zu verwenden, weil „die meisten Android-Anwendungen auf permanenter Interaktion mit dem Benutzer basieren“[9]. Das Framework ist eine Erweiterung der Bibliothek RxJava, die aus dem reactiveX Projekt hervorging. RxAndroid selbst besteht aus lediglich 4 Dateien, die 5 Klassen enthalten. Es ist üblich, dass bei der Entwicklung mit RxAndroid RxJava ebenfalls verwendet wird. Dies soll zu einer höheren Performanz beitragen und es soll Fehler verhindern.
Auf GitHub stehen sowohl RxJava als auch RxAndroid frei zur Verfügung. Sie unterliegen der Apache Lizenz 2.0.

A. Projekt einrichten

Für dieses Paper wurde mit der IDE Android Studios Version 3.1.4 gearbeitet. Diese gehört zur IntelliJ Plattform.
Es wurde ein neues Projekt angelegt. Es war zu beachten, dass es noch Schwierigkeiten mit der neuesten Android SDK API, Version 28, gibt. Diese sollte daher in den Einstellungen unter Systemeinstellungen, Android SDK, abgewählt werden. Verwendet wurde Android 8.0 (Oreo). Im Allgemeinen muss genau auf die SDK-Versionen geachtet werden die man wählt. Sie müssen in allen Bereichen wie Systemeinstellungen, aber auch beim Layoutdesigner und den virtuellen Devices gleich sein, weil ansonsten Probleme zum Beispiel beim Rendern der Layoutvorschau auftreten können.
Android Studio bringt direkt das Gradle Build Tool mit, welches Abhängigkeiten auflöst und den Build-Prozess übernimmt.
Um nun RxAndroid verwenden zu können muss als erstes die Abhängigkeit dem Gradle Build Tool bekannt gegeben werden. Hierzu werden in der Datei build.gradle (Module:app) im Ordner Gradle Scripts die URLs der Frameworks bekannt gegeben. Diese Anweisungen sind auf der GitHub-Seite von RxAndroid unter „Binaries“ im ReadMe.md zu finden.

B. App erstellen

Nun kann damit begonnen werden eine Android Application zu entwickeln. Hierzu werden UI-Komponenten angelegt und modifiziert. Für ein erstes triviales Beispiel wurden ein Button und zwei Textfelder gewählt.
In der zentralen Datei *MainActivity.java* werden nun ein Observable und ein Observer angelegt. Dabei ist zu beachten, dass die Klassen aus der reactiveX-Bibliothek gewählt werden. Da Observable und auch Observer generische Typen sind wird hier entschieden mit welchen Datentypen gearbeitet werden wird.
Innerhalb der gegebenen *onCreate()*-Methode wird beides initialisiert. Nun wird entschieden wie sich das Observable verhalten soll. Es stehen diverse Möglichkeiten zum Erstellen zur Verfügung. Um die Komplexität niedrig zu halten wurde die *just()*-Variante gewählt. Sie erwartet nun Werte der Art, die bei der Deklaration angegeben wurden. Diese werden lediglich als Übergabeparameter mitgegeben.
Um ein Observerobjekt zu initialisieren muss das Interface *Observer<T>* implementiert werden. Dieses besteht aus den void-Methoden *onSubscribe(Disposable d)*, *onNext(T t)*, *onError(Throwable e)* und *onComplete()*. Hier wird nun festgelegt was mit den empfangenen Items passieren soll.
Abschließend muss sich der Observer noch beim Observable registrieren. Dies kann entweder direkt beim Erstellen, also in der *onCreate()*-Methode geschehen oder man definiert einen anderen Zeitpunkt, beispielsweise mittels einer *onClick()*-Methode eines Buttons.

C. Ausführung

Android Studios ermöglicht es wahlweise auf einem realen Device oder einem virtuellen Device zu testen. Bei den virtuellen Devices stehen sehr viele Devices unterschiedlicher Hersteller in diversen Größen zur Verfügung. Für dieses Paper wurde als virtuelles Device ein Google Pixel mit 5“ Display verwendet. In Android Studio ist es möglich kleinere Änderungen direkt auf das virtuelle Device zu übertragen ohne dieses nochmals neu starten zu müssen.

D. Einfaches Beispiel

Wie in Abbildung 2 zu sehen ist wurde in der Methode *onSubscribe()* der Default-String eines Textfeldes gelöscht. In der *onNext()*-Methode wurden Strings konkateniert in das Textfeld eingefügt und mittels der *onComplete()*-Methode in einem zweiten Textfeld ein String hinzugefügt um das Beenden des Observables sichtbar zu machen. Die Zustände vor Programmablauf und danach sind in Abbildung 3 ersichtlich.

```
public class MainActivity extends AppCompatActivity {
    private TextView txt1, txt2;
    //Observable emits strings
    private io.reactivex.Observable<String> myObservable;
    //Observer accepts strings
    private Observer<String> myObserver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        txt1 = findViewById(R.id.Txt1);
        txt2 = findViewById(R.id.Txt2);
        myObservable = io.reactivex.Observable.just("android", "is", "easy", "as", "java");
        //to get an Observer we have to implement some methods because it's an interface
        myObserver = new Observer<String>() {
            @Override
            public void onSubscribe(Disposable d) {
                txt1.setText("");
            }
            @Override
            public void onNext(String s) {
                txt1.append(s+" ");
            }
            @Override
            public void onError(Throwable e) {}
            @Override
            public void onComplete() {
                txt2.append(" FINISHED");
            }
        };
    }
    public void LetsSubscribe(View view){
        myObservable.subscribe(myObserver);
    }
```

Abbildung 3, Quellcode des ersten Beispiels

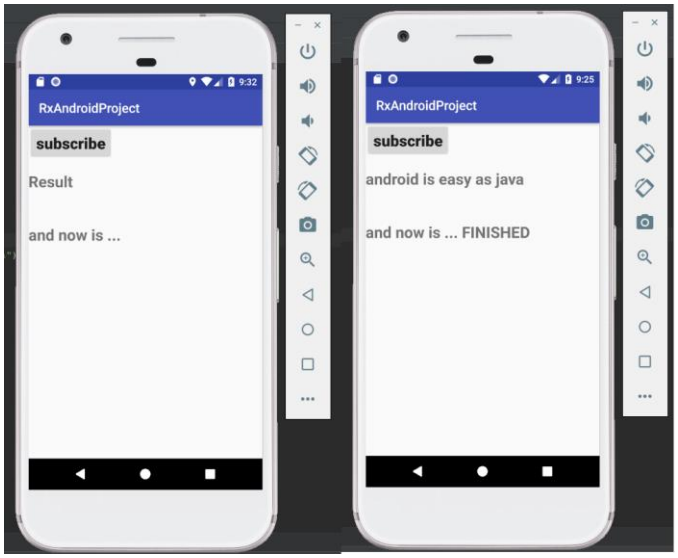

Abbildung 4, Startzustand und Endzustand des ersten Beispiels

E. Beispiel zwei mit map() und zipWith()

Hier wurden nun zwei Observables verwendet. Zu Beginn wurde *myStringObservable* mittels *create()* erstellt. Um die *create()*-Methode verwenden zu können muss die *onNext()*-Methode sowie die *onComplete()*-Methode für dieses Observable implementiert werden. Dies kann durch einen Lambdaausdruck geschehen oder zuvor in einem Handlerobjekt definiert werden, welches dann der *create()*-Methode übergeben wird. Es ist möglich, aber nicht zwingend notwendig, dass auch die beiden anderen Standardmethoden für Observables, *onSubscribe()* und *onError()*, implementiert werden.
Dieses erste Observable sendet eine Menge von Strings, welche zuvor in einer ArrayListe definiert wurden.
Das zweite Observable wurde in gleicher Weise erstellt, bekam aber eine ArrayListe mit Integerwerten.
Diese Integerwerte wurden in einem zweiten Schritt in Strings umgewandelt. Dazu wurde *map()* verwendet. Diesem Operator muss eine Funktion übergeben werden aus welcher hervorgeht was mit den einzelnen Items passieren soll. Im Beispiel wurden die Integerwerte in Strings umgewandelt. Die übergebene Funktion wird auf jedes Item einzeln angewendet. Um die Übergabefunktion auszudrücken wurde ein Lambdaausdruck eingesetzt. Zu beachten ist, dass ein neues Observableobjekt mit den transformierten Items erzeugt wird. Das Quellobservable wird nicht verändert.
Da nun beide Observables zu einem Ergebnis zusammengefasst werden sollten wurde der Operator *zipWith()* eingesetzt. Er bewirkt, dass die jeweils gesendeten Items im Wechsel zusammengefügt werden, unabhängig von einem zeitlichen Delay zwischen den einzelnen Sendungen. Der Aufruf dieses Operators ist nicht ganz trivial. Er wird auf einem Observabelobjekt aufgerufen. In diesem werden die neu erzeugten Items abgelegt. Als ersten Parameter wird ein zweites Observable angegeben. Nun erfolgt, erneut mittels eines Lambdaausdruckes, eine Funktion, die angewandt werden soll. Es werden zunächst 2 Objekte benannt mit denen gearbeitet werden soll. Man wählt je eines aus dem Zielobservable und eines aus dem Quellobservable. In diesem Beispiel wurde an die Namen aus dem Quellobservable *myStringObservable* je eine Reihe von Strings sowie die aus dem Zielobservable *myMappedObservable* stammenden, in Strings umgewandelten Zahlen angehängt.
Nun kann man sich auf das Zielobservable aus der vorherigen Operation registrieren um die neu erzeugten Items zu empfangen. Diese werden gesendet sobald sie den *zipWith()*-Operator verlassen.
Das Ergebnis der Operationen ist in Abbildung 6 zu sehen. Der Quellcode kann Abbildung 5 entnommen werden.

```
myStringObservable = Observable.create(emitter -> {
    try {
        for (String curr : persons) {
            emitter.onNext(curr);
        }
        emitter.onComplete();
    } catch (Exception e) {
        emitter.onError(e);
    }
});

myMappedObservable = Observable.create(emitter -> {
    try {
        for (Integer curr : ages) {
            emitter.onNext(curr);
        }
        emitter.onComplete();
    } catch (Exception e) {
        emitter.onError(e);
    }
})
.map(myInt -> Integer.toString((Integer) myInt))
        .zipWith(myStringObservable,(age, name) -> name.concat(" is "+age+" years old\n") );
```

Abbildung 5, Quellcode des zweiten Beispiels

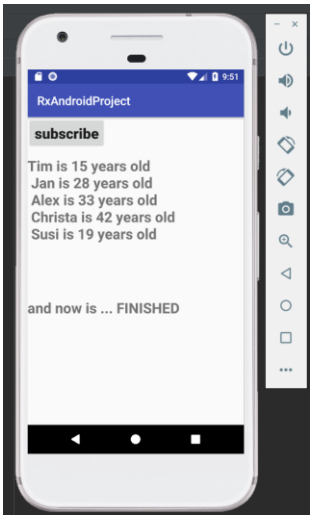

Abbildung 6, Endzustand des zweiten Beispiels

F. Beispiel drei mit Fehlermeldung durch onError()

Um eine Fehlermeldung durch die Standardmethode *onError()* zu provozieren wurde in diesem Beispiel ein Observable erstellt, welches in definierbaren Abständen die Items sendet. Hierzu wurde die Methode *interval()* zum Initialisieren des Observables verwendet. *Interval()* wurde in der Bibliothek RxJava mehrfach überschrieben. Hier wurden der Methode ein initiales Delay von einer Sekunde und der Wert für jedes weitere Delay, hier drei Sekunden, übergeben. Des Weiteren wurde die Zeiteinheit mittels des Enums TimeUnit auf Sekunden festgelegt.
In einem zweiten Schritt wurde der Operator *timeout()* verwendet. Er wurde ebenfalls auf dem Observable aufgerufen. Ihm werden der Zeitwert des Timeouts sowie die Zeiteinheit mittels des TimeUnit-Enums übergeben. Dieser Operator bewirkt, dass ein Fehler geworfen wird mittels der *onError()*-Methode sobald das Observable länger als im Timeout angegeben kein Item versendet. Im Beispiel wurde ein Timeout von 2 Sekunden gewählt. Zu beachten ist, dass in der *onError()*-Methode die Fehlermeldung nicht auf dem Device ausgegeben werden kann. Deshalb wurde sie im Beispiel in der Entwicklungsumgebung ausgegeben mittels

Log.e(). In einem realen System könnte man beispielsweise ein Alert auslösen.
Da hier ein initialer Delay von einer Sekunde und ein Timeout von 2 Sekunden gewählt wurde wird das erste gesendete Item erwartungsgemäß im Display angezeigt. Es folgt kein weiteres Item, da innerhalb des eingestellten 3-Sekunden-Delays der Timeout abläuft und ein Fehler geworfen wird. Nach dem Aufruf von *onError()* erfolgt kein weiterer Versand durch das Observable mehr. Es werden keine weiteren Methoden aufgerufen. Das Observable ist beendet.

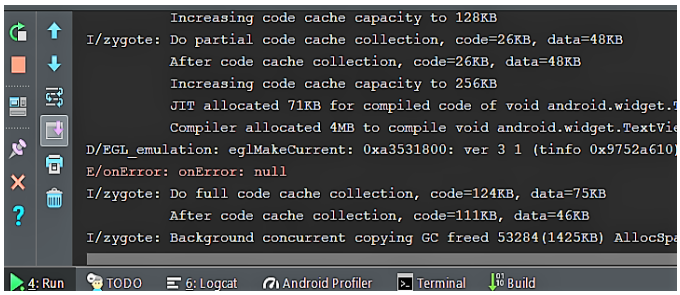

Abbildung 7. Fehlermeldung des dritten Beispiels

```
ublic class MainActivity extends AppCompatActivity {
    private TextView txt1, txt2;

    private Observable<Long> myObservable;

    private Observer<Long> myObserver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        txt1 = findViewById(R.id.Txt1);
        txt2 = findViewById(R.id.Txt2);

        myObservable = Observable.interval( initialDelay: 1,  period: 3, TimeUnit.SECONDS)
                .timeout( timeout: 2, TimeUnit.SECONDS);

        myObserver = new Observer<Long>() {
            @Override
            public void onSubscribe(Disposable d) {}

            @Override
            public void onNext(Long aLong) {
                txt1.setText(aLong + " ");
            }

            @Override
            public void onError(Throwable e) {
                Log.e( tag: "onError",  msg: "onError: "+e.getMessage());

            }

            @Override
            public void onComplete() {
                txt2.setText("FINISHED");
            }
        };
    }
```
Abbildung 8. Quellcode des dritten Beispiels

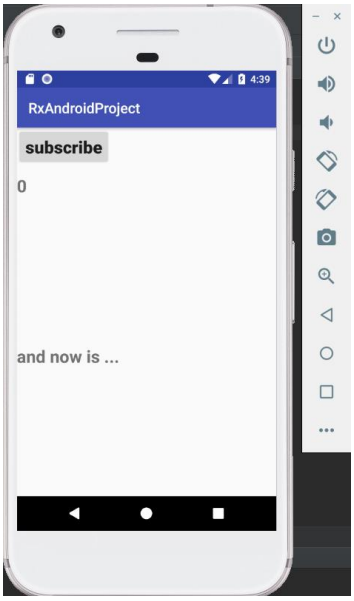

Abbildung 9. Endzustand durch Fehler des dritten Beispiels

G. *Scheduler in reactiveX*

Die reactiveX-Bibliotheken sind alle auf den Betrieb mit einem Thread voreingestellt. Wenn man sich nun die Vorteile der heutigen Hardware, die mittlerweile standardmäßig mehr als einen CPU-Core bietet, zu nutzen machen möchte muss man dies mittels speziellen Funktionen implementieren. Hierzu stehen in RxJava einige Variationen zur Verfügung. Für RxAndroid wurden diese nochmals erweitert. Alle 5 enthaltenen Klassen sind Schedulerklassen, die das Arbeiten auf mehreren Threads speziell im Android unterstützen sollen. Es stehen diverse Scheduler zur Verfügung, die bestimmen welcher Art von Thread erzeugt wird. Das sind unter anderem folgende:

1. *Scheduler.io()*
   Die hier erzeugten Threads sind besonders für I/O-Operationen geeignet. Es können theoretisch unendlich viele Threads erzeugt werden. Daher ist diese Form mit Bedacht zu verwenden.
2. *Scheduler.newThread()*
   Hierbei wird für jede Registrierung ein neuer Thread erzeugt. Auch hier können theoretisch unendlich viele Threads erzeugt werden.
3. *Scheduler.computation()*
   Dieser Scheduler erzeugt einen Threadpool.
   Die Anzahl der erzeugten Threads ist bei dieser Variante auf die Anzahl der auf dem System vorhandenen CPU-Cores beschränkt. Verwendet werden soll diese Variante für besonders rechenintensive Aufgaben.

Es ist weiterhin zu beachten, dass in den reactiveX-Bibliotheken, bzw. mittels deren Schedulern lediglich das Multithreading direkt unterstützt wird. Um echte Parallelität zu erhalten müssen die Observables nochmals speziell konfiguriert werden und es empfiehlt sich hierfür dann die Variante *Scheduler.from()* zu verwenden. Dieser benötigt einen Executor, der selbst erstellt werden muss. Somit ist die Möglichkeit gegeben hoch performante Anwendungen zu schreiben ohne dabei den Vorteil der reaktiven Bibliotheken,

das einfache Verwalten von Multithreading-Anwendungen, zu verlieren.

H. Beispiel vier mit Schedulern

Als erstes wurde in der *onCeate()*-Methode in einem Textfeld ausgegeben von welchem Thread diese Methode gerade ausführt wird. Folgend wurde mit der Funktion *range()* ein Observable erstellt, welches Integerwerte ausgibt. Im ersten Parameter wird der Startwert angegeben. Mittels des zweiten Parameters wird mitgeteilt wie viele Werte gesendet werden sollen. In Beispiel vier wurden die Werte 1 bis 5 gesendet. Der Observer wurde erstellt und so konfiguriert, dass er in der *onSubscribe()*-Methode den aktuell ausführenden Thread ausgibt. In der *onNext()*-Methode wird ebenfalls der ausführende Thread ausgegeben und das aktuelle Ergebnis der Operation angehängt. Mittels *onComplete()* wurde der Thread ausgegeben welcher das Observable beendet.
Aufgrund der in diesem Beispiel notwendigen Verkettung von Operatoren wurde das Observable in der *LetsSubscribe()*-Methode weiter transformiert. Es wurde die Methode *subscribeOn()* aufgerufen. Mittels dieser lässt sich bestimmen in welcher Art von Thread das Senden der Items des Observables und die darauffolgenden Operationen ausgeführt werden sollen. Sie benötigt einen Scheduler als Parameter und kann nur einmalig in einer Kette von Operatoren aufgerufen werden. Ruft man diese Methode auf wird alles Weitere an den übergebenen Thread delegiert und der aufrufende Thread, zum Beispiel der Mainthread mit der UI, wird wieder frei. So lässt sich effizient und einfach ein Einfrieren der Benutzerober-fläche verhindern, ganz im Sinne der im reaktiven Manifest geforderten allzeitigen Antwortbereitschaft. Dieser neue Thread übernimmt nun das Senden sowie das Ausführen aller geforderten Operationen bis hin zum letztlichen Ausliefern des transformierten Items an das registrierte Objekt. Im Beispiel wurde der oben beschriebene *Scheduler.io()* verwendet. Der nächste Operator in der Kette ist die *map()*-Funktion, der hier mittels Lambdaausdruckes die Aufgabe gegeben wurde dem aktuellen Wert 2 hinzuzuaddieren. Die Verkettung geht weiter mit der *doOnNext()*-Methode. Dieser muss mitgeteilt werden, was mit dem gerade erzeugten Zwischenergebnis zu tun ist. Hier wurde es zusammen mit dem Namen des ausführenden Threads einfach ausgegeben. Folgend wurde die Methode *observeOn()* benutzt. Diese benötigt als Übergabeparameter wieder einen Scheduler. Ab diesem Zeitpunkt wird das weitere Transformieren des Zwischenergebnisses und die Verantwortung für die Auslieferung an einen neuen Thread übergeben. Somit ist der ursprünglich verwendete Thread aus der Methode *subscribeOn()* wieder frei und kann das nächste Item bearbeiten. *OnObserve()* kann in einer Operatorenkette mehrmals aufgerufen werden um so die Performanz der Anwendung zu erhöhen. Selbstverständlich sollte dies nicht leichtfertig geschehen, sondern es muss gut abgewogen werden an welchen Stellen ein Threadwechsel tatsächlich sinnvoll ist. Sollte man zu viele Threads erzeugen verschlechtert sich naturgemäß die Performanz und man erreicht das Gegenteil von dem eigentlich angestrebten Ziel.
In dem neu erzeugten Thread wird nun eine weitere Transformation durchgeführt. Mittels *map()* wird das Item mit 10 multipliziert. Im letzten Schritt wird der Observer registriert, welcher das Endergebnis erhalten soll.
Wie in Abbildung 10 im zweiten Textfeld zu sehen ist wurde *onCreate()* noch im Mainthread ausgeführt.
Das Registrieren mittels der *onSubscribe()*-Methode des Observers wurde ebenfalls im Mainthread durchgeführt, wie in Abbildung 10, erste Zeile des ersten Textfeldes, zu sehen ist. Die erste Operation, das Addieren mit 2, wurde in einem Thread mit dem Namen RxCachedThreadScheduler-1 aufgeführt. Durch das im Code nun folgende Aufrufen von *observeOn()* wird nun zum Thread RxComputationThreadPool-1 gewechselt. Dieser beendet die Operationen auch, wie im letzten Textfeld der Abbildung 10 zu sehen ist. Dies entspricht dem erwarteten Verhalten, wonach der Thread, der die letzten Operationen durchführt auch für die Auslieferung verantwortlich ist.

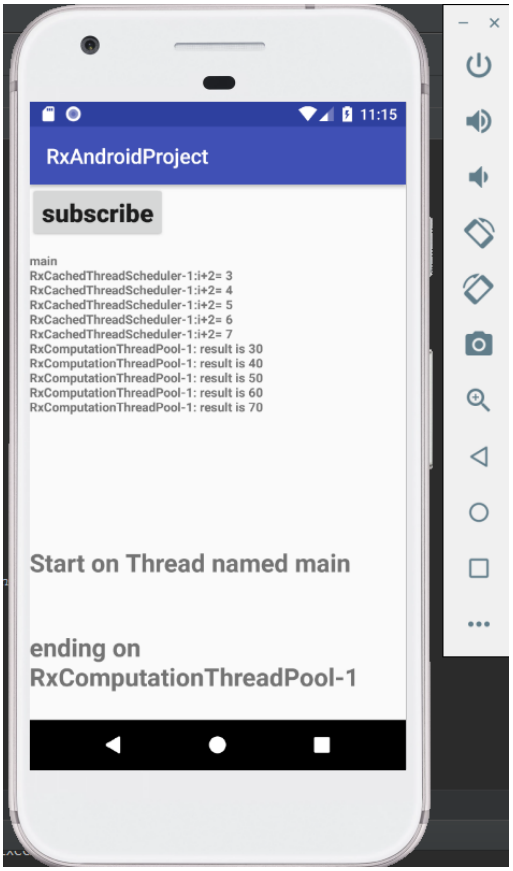

Abbildung 10. Endzustand des vierten Beispiels

```
    String startThread = "Start on Thread named " + Thread.currentThread().getName();
    txtMain.setText(startThread);

    myObservable = Observable.range( start: 1,  count: 5);

    myObserver = new Observer<Integer>() {
        @Override
        public void onSubscribe(Disposable d) {
            txt1.setText(Thread.currentThread().getName()+"\n");
        }

        @Override
        public void onNext(Integer integer) {
            txt1.append(Thread.currentThread().getName()+": result is "+integer+"\n");
        }

        @Override
        public void onError(Throwable e) {
            Log.e( tag: "onError", e.getMessage());
        }

        @Override
        public void onComplete() {
            txt2.setText("ending on "+Thread.currentThread().getName());
        }
    };
}

public void LetsSubscribe(View view) {
    myObservable.subscribeOn(Schedulers.io())
            .map(i-> i+2)
            .doOnNext(i -> txt1.append(Thread.currentThread().getName()+":i+2= "+i+"\n"))
            .observeOn(Schedulers.computation())
            .map(i-> i*10)
            .subscribe(myObserver);
}
```

Abbildung 11. Quellcode des vierten Beispiels

VI. FAZIT

Aufgrund der einfachen Installation der Entwicklungsumgebung Android Studio und deren Komfort in der Nutzung war der Einstieg in Android, als auch in RxAndroid sehr leicht. Android Studio hat das Buildtool Gradle direkt integriert, so dass der Zugang zu den benötigten Bibliotheken RxJava Und RxAndroid mit zwei import-Statements gelungen ist. In manchen Situationen, gerade nach größeren Änderungen wie dem Wechseln zwischen den einzelnen Beispielen, gab es Buildfehler, welche aber durch Aufräumen des Projektes und dessen Rebuild immer schnell behoben werden konnten. Auch die Testmöglichkeiten auf einem realen Endgerät waren intuitiv zu erfassen. Das Erstellen eines virtuellen Devices ging ebenfalls erfreulich einfach und schnell. Lediglich der Auswahl einer geeigneten Androidversion musste etwas mehr Zeit gewidmet werden.
Die vom reactiveX-Projekt angebotenen Funktionen sind gut verständlich dokumentiert und mittels der auf der Website verwendeten und erklärten Marble-Diagramme auch anschaulich dargestellt. Die einzelnen Funktionen haben jeweils eine eigene kurze Seite zur Erklärung. Auf diesen Seiten befinden sich Links zu ausführlicheren Erklärungen und teilweise auch zu Beispielen. Somit ist es gut möglich die zum Verwendungszweck passende Funktion zu finden.
Allerdings sind die Beispiele alle auf RxJava ausgerichtet. Zu RxAndroid fand sich leider kein konkretes Beispiel, was auch dem minimalen Umfang der Bibliothek geschuldet sein kann. Es musste darauf geachtet werden, dass die Beispiele in unterschiedlichen Programmiersprachen geschrieben sind. Allerdings war das Umsetzen in Java eigentlich immer leicht möglich.
RxAndroid kann somit grundsätzlich empfohlen werden. Es ist einsteigerfreundlich und intuitiv anwendbar. Grundlegende Programmierkenntnisse sollten aber dennoch vorhanden sein. Das Umdenken in die reaktive Programmierung stellt keine große Hürde dar, wenn man bereits objektorientiert programmiert hat. Es ist wünschenswert, dass die Bibliothek weiterhin gepflegt wird. Der schnelle Einstieg und das innovative Konzept der reaktiven Programmierung sind so angenehm, dass man Lust bekommt weiterhin in diesem Themenfeld aktiv zu sein.

REFERENZEN

[1] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, vol.21, no. 3, pp.3. September 1989

[2] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, M. Mezini. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on software engeneering*, vol. 43, no.12, pp.1. Dezember 2017.

[3] Stephen Blackheath, Anthony Jones.“Stop listening!“ in *Functional Reactive Programming*, Shelter Island, New York, USA: Manning Publications Co., 2016, Kapitel 1

[4] Alessandro Margara, Guido Salvaneschi. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on software engeneering*, vol. 44, no.7, pp.1. Juli 2018.

[5] Bainomugisha, A.L. Carreton, T. van Cutsem, S. Mostinckx, W. de Meuter. A Survey on Reactive Programming. *ACM Computing Surveys*, vol.45, no.4, Article 42, pp.52:4. August 2013

[6] Jonas Bonér, Dave Farley, Roland Kuhn und Martin Thompson. *„Das Reaktive Manifest“.* [Online]. Abrufbar: https://www.reactivemanifesto.org/de. Stand: 30.07.2018 12.15 Uhr

[7] Z.Jovanovic, R.Bacevic, R. Markovic, S. Randjic. „Android Application for Observing Data Streamsfrom Built-In Sensors using RxJava“ Präsentiert bei 23rd Telecommunicatins forum TELFOR 2015, Belgrad, Serbien, 24-26. November 2015

[8] *„ReactiveX“* [Online]. Abrufbar: http://reactivex.io/documentation/operators.html. Stand: 07.08.2018 11.10 Uhr

[9] A.Sutula. Functional reactive paradigm advantages for android development. *International Scientific Journal* no.9 2015