Hochschule
Kaiserslautern
University of
Applied Sciences

PENTASYS
Unser Maßstab ist der Mensch

Hochschule Kaiserslautern
- Fachbereich Informatik und Microsystemtechnik -

Der Titel der Arbeit

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt von

Vorname Nachname
12345

Betreuer Hochschule:	Prof. Dr.
Betreuer PENTASYS:	B. Sc.

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

> Dies ist ein Zitat.

verstand, scheinen nun doch vorueber zu Dies ist der Text sein.
siehe: http://janeden.net/die-praeambel

Inhaltsverzeichnis

1 Einleitung **1**

2 Was bedeutet *"reactive"* im Kontext der Softwareentwicklung **2**

3 Abgrenzung zwischen Java 8 Streams und Reactive Streams **5**

3.1 Java 8 Streams 5

3.1.1 Bulk Operations 6

3.2 Unterscheidung zu Reactive Streams 6

4 Bausteine von Reactive Programming **8**

4.1 Observer Pattern 8

4.2 Back Pressure 9

4.3 Reactive Streams 10

5 Unterscheidung von Reactive Systems, Reactive Programming und Functional Reactive Programming **11**

5.1 Differenzierung zwischen Reactive Proramming und Reactive Systems 11

5.2 Reactive Programming vs. Functional Reactive Programming 13

6 Einführung in Reactive Programming mit RxJava2 **15**

6.1 Synchronität und Asynchronität 16

6.2 Parallelisierung und Nebenläufigkeit 16

6.3 Push und Pull 16

6.4 Observable 17

6.5 Flowable 19

6.6 Single 21

6.7 Subject 22

6.8 Processor 23

6.9 Schedulers 24

6.10 Operationen 25

6.10.1 Operation filter() 25

6.10.2 Operation map() 26

6.10.3 Operation merge() 27
6.10.4 Operation zip() 28

7 Beispiel: Implementierung eines Systemmonitors 30
7.1 Klassenbeschreibungen SServerSSeite 30
7.1.1 API für Systemwerte 30
7.2 Klassenbeschreibungen "ClientSSeite 30
7.2.1 RxJavaFx 30
7.3 Hier eventuell Testing noch beschreiben 30

8 Evaluierung 31
8.1 Ausblick 31

Literaturverzeichnis I

Abbildungsverzeichnis III

Tabellenverzeichnis IV

Kapitel 1

Einleitung

Durch den starken Wachstum der IT entstehen immer wieder neue Möglichkeiten Anwendungen zu realisieren. Durch den technischen Fortschritt werden schon bekannte Muster und Architekturen weiter entwickelt oder Ideen für die neu entstandenen Anforderungen umgesetzt. Eine dieser Ideen ist *REACTIVE PROGRAMMING*. Vor noch nicht allzu langer Zeit waren Monolithen die auf eigens gehosteten Servern ausgerollt wurden der Stand der Dinge. Durch die mittlerweile entstandene Vielfalt an Endgeräten wie Smartphones, das Deployment in der Cloud oder die Menge an spezialisierten Programmiersprachen, Frameworks und Entwicklungswerkzeugen haben sich Anforderungen herausgestellt die sich mit bekannten Lösungen wie zum Beispiel einer objektorientierten Herangehensweise nicht zur vollen Zufriedenheit erfüllen lassen. Für einen Benutzer ist es üblich, dass Änderungen sofort sichtbar sind und angefragte Daten in Bruchteilen einer Sekunde bereit stehen, und das zu jedem Zeitpunkt. Kann eine Applikation dies nicht leisten, kann die User Expierience [1] in Mitleidenschaft gezogen werden, was bei der Menge an Diensten gleicher Art dazu führen kann, dass Benutzer auf die Dienste von Mitbewerber zurück greifen. Um diesem vorzubeugen wurde aus vorhandenen Konzepten ein grundlegendes Manifest für *Reaktive Systeme*[2] erstellt. Die im Manifest angeführten Eigenschaften werden im Verlauf dieser Arbeit noch genauer aufgegriffen. Es sei nur jetzt schon gesagt, dass zur Umsetzung der Kriterien für ein Reaktives System, die Verwendung von Reactive Programming nicht notwendig ist, jedoch meist sinnvoll scheint.
Ziel dieser Arbeit ist es, eine Einführung in die Welt von Reactive Programming zu geben. Dazu wird im nächsten Kapitel eine Einordnung in das Gesamtbild der Softwareentwicklung durchgeführt. Ebenso werden die Eigenschaften und Eigenheiten von Reactive Programming beschrieben. Weiterhin gibt es einen Überblick über einige Frameworks mit deren Hilfe ein reaktives Programmieren realisiert werden kann. Darauf folgend wird eine Implementierung einer Beispielanwendung besprochen, um dem Vorangegangen eine praktische Anwendung hinzuzufügen.

[1] TODO: User Experience Def
[2] [Bon14]: Reactive Manifesto 2.0. www.reactivemanifesto.org

Kapitel 2

Was bedeutet *"reactive"* im Kontext der Softwareentwicklung

Grundlegend wird etwas als reaktiv bezeichnet, wenn eine Reaktion durch eine vorangegangene Aktion ausgelöst wird. Bei diesen Aktionen handelt es sich entweder um Veränderungen an verwendeten Daten oder stattfindende Ereignisse(*Events*). Eine gutes Beispiel zur Veranschaulichung ist die Benutzeroberfläche(*GUI - Graphical User Interface*). Ein Benutzer bestätigt eine vorgenommene Eingabe durch das klicken eines Button innerhalb der GUI. Dieses Event sorgt dafür, dass die Applikation einen vorgegebenen Vorgang ausführt. Dieses Vorgehen sorgt in der klassischen objektorientierten Programmierweise mit sequentiellem Ablauf sowie dem imperativen Ansatz grundsätzlich für ein stetig wachsendes Maß an Komplexität. Durch die unterschiedlichen Events die innerhalb der GUI ausgelöst werden können(Mausklick, Tastendruck, usw.) ist ein klassisch imperativer und sequentieller Ablauf des Programmcodes nicht realistisch, da kein Entwickler weiß, in welcher Reihenfolge und zu welchem Zeitpunkt ein, beziehungsweise welches, Event ausgelöst wird. Somit spielt hier die *Inversion of Control*, also das Umkehren der Kontrolle, eine große Rolle [Mar05]. Diese besagt, dass nicht der geschrieben Code den Ablauf beschreibt, sondern die Kontrolle bei dem Framework, welches für die Interaktion zuständig ist, liegt, und dieses entscheidet wiederum wie und wann auf ein Event reagiert wird. Im Code spiegelt es sich dadurch wider, dass mögliche Reaktionen, meist als Methoden oder Funktionen realisiert, an die mögliche eintreffenden Ereignisse gebunden werden. Dies geschieht über so genannte Event-Handler.

```
public class Handler implements EventHandler< Event >
{
 @Override
 public void handle( Event arg0 )
 {
  System.out.println( "Event of type: " + arg0.getEventType() );
  arg0.consume();
 }
}
```

Listing 2.1: Implementierung des eigentlichen Event Handlers

Im Listing 2.1 sieht man eine Klasse welche das Event Hander Interface implementiert. Dieses Interface liefert eine Funktion *handle(Event)* die implementiert werden muss. Innerhalb dieser Methode wird die Reaktion auf das aufgetretene Event definiert, in diesem Beispiel soll die Art des Events auf der Konsole ausgegeben werden. In Listing 2.2 wird nun dem einzigen Element der GUI, dem Button, ein Event Handler zugewiesen. Die Parameter besagen, dass jedes auftretenden Mouse-Event den Handler auslöst. Somit wird der Handler an ein auftretendes Ereignis gebunden, und der Programmablauf findet nicht mehr sequentiell statt, sondern das Framework der Oberfläche erkennt das Event und der passende Handler wird aufgerufen. An ein Objekt können mehrere Handler gebunden werden, die auf das gleiche oder unterschiedliche Events reagieren.

```
public class ApplicationRunner extends Application
{
 public static void main( String[] args )
 {
  Application.launch( args );
 }

 @Override
 public void start( Stage arg0 ) throws Exception
 {
  HBox root = new HBox();
  Button button = new Button( "hi, I'm button! Hover over me!" );

  button.addEventHandler( MouseEvent.ANY, new Handler() );

  root.getChildren().setAll( button );
  arg0.setScene( new Scene( root, 180, 40 ) );
  arg0.show();
 }
}
```

Listing 2.2: Klasse zum Anwendungsstart in welcher auch der EventHander gesetzt wird.

Die Abbildung 2.1 zeigt nun den Button an welchem der Event-Handler registriert wurde. Jedes Mouse-Event welches vom Button erkannt wird, löst diesen Handler aus. Eine Beispielausgabe sieht man in Abbildung 2.2.
Dieses Beispiel veranschaulicht wie grundsätzlich ein reaktives Verhalten einer Anwendung erzeugt wird. Es wird klar, dass Reaktivität somit schon lange in der Softwareentwicklung eine Rolle spielt.

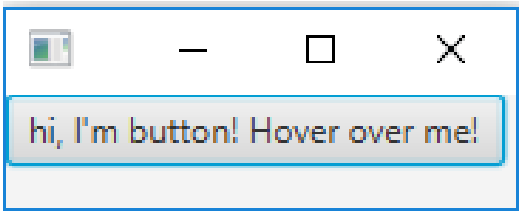

Abbildung 2.1: Anwendung mit Button.

```
ApplicationRunner [Java Application] C:\Dev\j
Event of type: MOUSE_MOVED
Event of type: MOUSE_MOVED
Event of type: MOUSE_MOVED
Event of type: MOUSE_MOVED
Event of type: MOUSE_PRESSED
Event of type: MOUSE_RELEASED
Event of type: MOUSE_CLICKED
Event of type: MOUSE_MOVED
Event of type: MOUSE_EXITED
Event of type: MOUSE_EXITED_TARGET
```

Abbildung 2.2: Die Konsolenausgabe für die Button Events.

Kapitel 3

Abgrenzung zwischen Java 8 Streams und Reactive Streams

Im Verlauf dieser Arbeit wird beschrieben welche Bestandteile benötigt werden, damit von Reactive Streams gesprochen werden kann. Bevor jedoch diese Bestandteile erläutert werden, muss eine Abgrenzung zu den seit Java 8 vorhanden Streams stattfinden. Durch die ähnliche Art der Verwendung beider ist eine Verwechslung nicht ausgeschlossen, obwohl unterschiedliche Konzepte realisiert werden. Daher folgt eine kurze Beschreibung der Java 8 Streams und eine allgemeine Unterscheidung zu den im weiteren Lauf verwendeten Reactive Streams.

3.1 Java 8 Streams

Das Konzept der Streams stellt eine Abstraktion für Folgen von Bearbeitungsschritten auf Daten dar [Ind15]. Streams erinnern an Collections, sind jedoch nur einmal traversierbar und nehmen keine direkten Speicherung der Daten vor. Collections können auch als Streams repräsentiert werden. Nahe liegt die verbreitete Analogie der Fließbandverarbeitung. Man hat eine Menge von Objekten die nacheinander gewissen Arbeitsschritten unterzogen werden. Diese werden einmalig durchgeführt und die Zwischenergebnisse bleiben nicht vorhanden. In Listing 3.1 sieht man, dass mit dem Methodenaufruf *stream()* auf der Collection ein Stream parametrisiert auf den selben Datentyp wie die Collection erstellt werden kann. Es findet keine Änderung an den einzelnen Elemente statt, denn es wird nur das Ergebnis der Operation ausgegeben.

```
public class StreamsAndBulk
{
 public static void main( String[] args )
 {
  List< String > asCollection = Arrays.asList( "Alpha", "Beta", "Gamma", "
    Delta" );
  Stream< String > asStream = asCollection.stream();
  asStream.map( s -> s.toUpperCase() ).filter( s -> s.contains( "L" ) ).
    forEach( s -> System.out.println( s ) );
 }
```

Listing 3.1: Beispiel Erstellung, Verarbeitung und Ergebnisermittlung von Streams.

3.1.1 Bulk Operations

Bulk Operations gelten als funktional und sind seit Java 8 auf Collections sowie Streams anwendbar. Diese Operationen müssen nicht separat implementiert werden und können direkt auf Collections oder Streams ausgeführt werden. Jedoch sind nicht zu beiden Arten die exakt gleichen Operation verfügbar. Somit kann durch eine Operation zum Beispiel eine Veränderung an jedem Objekt einer Liste ausgeführt werden, oder eben auf jedem Objekt welches einen Stream durchquert. Die Operation können verkettet werden. Als Ergebnis werden die Elemente, zum Beispiel nach der Filterung ausgegeben, oder in einer neuen Liste gespeichert. Auch bei einer Verkettung wird immer nur das Ergebnis weiter gereicht, die Eingangswerte bleiben unverändert. Erwähnung sollten noch die unterschiedlichen Arten von Operationen finden. Man unterscheidet zwischen drei Arten: *Erzeugung*, *Berechnung* und *Ergebnisermittlung* die sich wie folgt abbilden lassen [Ind15] :

$$\underbrace{Quelle \Rightarrow STREAM}_{Erstellung} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow ... \Rightarrow OP_n}_{Berechnung} \Rightarrow \underbrace{Ergebnis}_{Ergebnisermittlung}$$

Bei der Erzeugung wird von der Änderung der Datenrepräsentation von einem Datentyp einer Collection oder eines Arrays in einen Stream gesprochen. Java bietet hierfür für jeweils Arrays oder Collections Methoden an. Resultierend erhält man einen Stream. Eine Reihe von Berechnungen kann nun verkettet stattfinden, zum Beispiel eine Transformation oder Filterung nach Kriterien. Sind die Operationen abgeschlossen wird das Ergebnis, zum Beispiel auf der Konsole, ausgegeben oder in einem Datentyp gespeichert. Konkret ist der Ablauf in Listing 3.1 zu sehen. Wie vorhin schon erwähnt wird via Methodenaufruf eine Stream-Repräsentation der Collection geschaffen. Die Berechnung beziehungsweise Bearbeitung findet in der Konkatenation der Operation *map()* und *filter()* statt. Zuerst wird jeder String in Großbuchstaben transformiert, und das Ergebnis an die Filter-Operation weitergegeben. Anschließend wird gefiltert ob der String den angegeben Kriterien genügt. Die Ergebnisermittlung findet in der *forEach()*-Methode statt, da hier jedes Element, das nach dem filtern noch übrig ist, auf die Konsole geschrieben wird. Es können beliebig viele Stream-Objekte von einer Collection erstellt werden, jedoch ist die Traversierung, und somit die Verwendung, eines Streams nur einmalig möglich.

3.2 Unterscheidung zu Reactive Streams

Die vorab beschriebenen Streams und die dazugehörigen Operationen funktionieren durch die Verwendung des Pull-Verfahrens. Mittels Iterator wird immer ein weiteres Element von der Datenquelle geholt und darauf die Operationen angewandt. Ebenso sind die Streams endlich,

da nach der oder den Operationen der Stream immer terminiert. Auch sind diese Streams nur einmalig verwendbar. Wird von Reactive Streams gesprochen gelten diese Eigenschaften nicht. Es wird nach dem Push-Verfahren gehandelt, was heißt, das nicht mit dem Iterator gearbeitet wird. Auch müssen diese Streams nicht endlich sein, was wiederum heißt, dass die Anzahl der Elemente nicht vorab bekannt sein muss. Auch ist eine mehrfache Verwendung dieser Reactive Streams möglich. Die Bulk Operations sind auch auf Reactive Streams möglich, es liegen jedoch viele weitere Operationen vor die speziell für die Verwendung des Push-Verfahrens implementiert wurden. Diese signifikanten Unterschiede zeigen, dass wenn auch in beiden Fällen von Datenströmen gesprochen wird, der grundlegende Gedanke wie mit Streams umgegangen wird eine vollkommen andere ist.

Kapitel 4

Bausteine von Reactive Programming

Durch große Datenraten, Datenmengen und viele vernetzte Geräte und Dienste die zur heutigen Zeit in Umlauf sind, entstehen viele Informationen die verarbeitet werden können beziehungsweise müssen. Durch die vielen Berechnungen und Verarbeitungsschritte die dafür nötig sind, spielt hier das Gesetz von Amdahl eine große Rolle [Amd]. Dieses besagt, dass nie alle Teile eines Programms parallel ausgeführt werden können. Deshalb zerlegt man das Programm in eine sequentiellen und einen parallelen Teil. Reactive Programming versucht nun eine Lösung abzubilden, welche die Auslastung der Rechenressourcen optimiert und die Anzahl der Stellen welche serialisiert ablaufen zu verringern [Bon]. Ein Ansatz diese Lösung zu realisieren ist die Verwendung von besagten Reactive Streams. Neu auftretende Daten werden als Ströme betrachtet, die unter Beobachtung stehen und Änderungen sowie Bearbeitung der Daten asynchron und parallel zu den anderen Funktionen einer Anwendung ausgeführt werden können. Damit dieses Verfahren der Datenverarbeitung funktioniert, sind einige Konzepte zu beachten.

4.1 Observer Pattern

Ein bewährtes und klassisches Vorgehen bei Anwendungsanforderungen dieser Art ist das schon 1994 von Erich Gamma beschriebene Entwurfsmuster, dem *Observer-Pattern* [GHJV11]. Auch hier lässt sich wieder die GUI als guten Beispiel heran ziehen. Als Beispiel betrachte man eine Oberfläche welche die Temperatur im Raum anzeigt. Um nun das Beobachtermuster zu implementieren, muss der sich ändernde Wert, der zum Beispiel von einem Temperatursensor gemessen wird, beobachtet werden. Dieser Sensor repräsentiert in Abbildung 4.1 eine Implementierung des Subject. Man nehme an auf der GUI soll der Temperaturwert als Balkendiagramm(BarChartForTemp), also als Art Thermometer dargestellt werden. Ebenso soll die aktuelle Temperatur in Schrift(LabelForTemp) auf der GUI erscheinen. Die beiden Objekte implementieren das Observer-Interface wie in Abbildung 4.1 gezeigt. Dies heißt wiederum, dass diese beiden Observer die *register()*-Methode des Subjects aufrufen und somit in die Observer-Liste dieses Subjects, also dem Sensor, aufgenommen werden. Ändert sich nun zum Beispiel die Temperatur wird die *notifyObservers()*-Methode aufgerufen, was heißt, dass über die Liste der

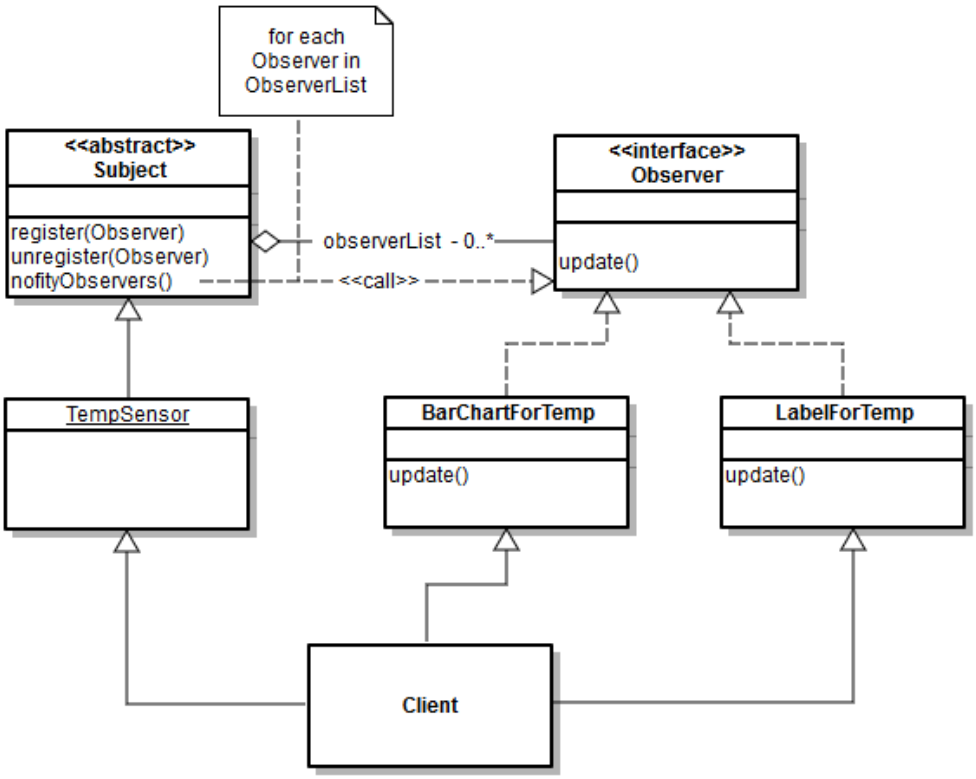

Abbildung 4.1: Schematischer Aufbau eines Observer Patterns.

Observer iteriert wird und die *update()*-Methode jedes Observers aufgerufen wird. Diese Methoden können nach dem *Push* oder *Pull* Verfahren implementiert werden. Bei Verwendung des Push-Modells wird dem Observer der Wert des geänderten Parameters direkt mitgeteilt. Das Pull-Modell verfolgt den Ansatz, dass der Observer eine Information erhält, dass sich Werte geändert haben, er muss jedoch über die Getter-Methoden die für ihn relevanten Werte aktiv nachfragen. Je nach Vorgehen ergeben sich Vor- und Nachteile. Der Push-Ansatz steht eher für lose Kopplung, da der Observer keine Details des Subject kennen muss. Jedoch sinkt dadurch die Flexibilität, da Observer Interfaces exakter beschrieben werden müssen, damit das Subject weiß, welche Informationen weiter gereicht werden sollen. Die Kopplung im Vergleich zum Pull-Ansatz ist nicht wirklich lose, da jeder Beobachter wissen muss, welche Daten das observierte Objekt repräsentiert und wie auf diese Daten zugegriffen werden kann. Jedoch findet sich hier die Flexibilität wieder, da jeder Beobachter wenn er Information braucht, exakt diese Daten abrufen kann und sich nicht auf die korrekte Datenverteilung des Subjects verlassen muss.

4.2 Back Pressure

Der weitere wichtige Aspekt der Reactive Streams ist ein nicht blockierender Back Pressure. Man betrachtet hier den Push-Ansatz. Ein Producer generiert Daten, die ein Consumer verarbeitet. Es kann nun passieren das mehr Daten produziert werden als konsumiert werden

können. Wird darauf nicht reagiert, kann es zu starken Ressourcenproblemen oder zu einem Programmabsturz durch einen *OutOfMemoryError* kommen. Um dies zu verhindern, benötigt man eine Möglichkeit, die Anzahl der zu verarbeitenden Objekte zu regulieren. Ein sogenannter *Feedback Channel* steht dafür zur Verfügung. Über diesen kann ein Consumer seinem Producer mitteilen, mit wie vielen Objekten er maximal umgehen kann. Dadurch ergibt sich die Möglichkeit auf Seiten des Producers seine Datenbeschaffung und Datenweitergabe zu regulieren, sodass der Producer zu jeder Zeit verwendbar bleibt und nicht blockiert. Dies geschieht zum Beispiel über einen zusätzlichen Puffer an Ausgang des Producers, wodurch die direkte Datenweitergabe verzögert werden kann. Dies besteht jedoch nur, wenn es dem Producer auch möglich ist, seine Geschwindigkeit zu steuern. Zu beachten ist, dass grundsätzlich jeder Consumer über einen Eingangspuffer verfügt, in welchem die geforderten Objekte zwischen lagern. Wird der Back Pressure nicht richtig behandelt oder die Puffer fehlerhaft konfiguriert, kann es zu eine *MissingBackPressureException* kommen.

4.3 Reactive Streams

Nach dem anfangs erwähnten Observer Pattern werden die Streams observiert und die Daten werden den Observern publiziert. Verfügen diese Observable Streams nun über die Funktionalität des Back Pressures ist eine nicht blockierende asynchrone Bearbeitung möglich. Nach diesen Voraussetzungen wurde in der Welt der JVM die Initiative der *Reactive Streams* geschaffen, um einen Standard für diese Art von Verarbeitung zu etablieren [rsm]. Das zusätzlich zu bewältigende Problem ist die unterschiedliche Implementierung der bereits existierenden reaktiven Frameworks. Durch die Standardisierung soll eine Kompatibilität von reaktiven Komponenten untereinander gesichert werden, auch wenn besagte Komponenten auf unterschiedliche Frameworks basieren. Viele Entwicklerteams von reaktiven Frameworks haben sich mittlerweile dieser Initiative angeschlossen und die Schnittstellen soweit angepasst, dass diese Kompatibilität gewährleistet werden kann [rsl]. Mit Java 9 wird in der sogenannten *Flow API* eine Implementierung nach den *Reactive Streams* Kriterien geliefert [floa]. Ein Anwendungsbeispiel wird von der offiziellen Java-Community bereits zur Verfügung gestellt [flob].

Kapitel 5

Unterscheidung von Reactive Systems, Reactive Programming und Functional Reactive Programming

Im Zusammenhang mit Reactive Programming treten weitere Begriffe auf, die auch das Schlüsselwort *reactive* in ihrer Bezeichnung tragen. Die *Reactive Systems* sowie das *Fuctional Reactive Programming*. In diesem Kapitel wird eine Differenzierung dieser Begriffe von Reactive Programming durchgeführt. Ebenso soll dadurch eine Einordnung von Reactive Programming in den Kontext der Softwareentwicklung vereinfacht werden.

5.1 Differenzierung zwischen Reactive Proramming und Reactive Systems

Bislang wurde von einem reinen Vorgehen für eine passende Implementierung gesprochen. Wie jedoch wirkt es sich aus, wenn eine komplette Anwendung reaktiv reagieren soll? Wie verhält es sich weiter wenn mehrere Teile reaktiv funktionieren und kommunizieren sollen? Der grundlegende Gedanke reaktiver Systeme wurde schon im Jahre 1985 in einem Paper von D. Harel und A. Pnueli beschrieben [HP85]. Zur heutigen Zeit jedoch bezieht man sich bei Richtlinien zur Gestaltung reaktiver Systeme eher auf das Reactive Manifesto [Bon14]. Laut Jonas Bonér und den vielen Unterstützern sind vier Bestandteile essentiell, damit eine Anwendung die Anforderung erfüllt um als reaktiv zu gelten. Die Ansicht des Manifests stützt sich auf einen Architekturbeziehungsweise Designstil und soll als Grundlage zur Entwicklung reaktiver Systeme dienen. Die folgenden Erklärungen sind dem Manifest entnommen und sollen einen Verständnis zu den vier Eigenschaften bieten wie sie auch in Abbildung 5.1 aufgezeigt werden. Wie in diesem beschrieben sind Reaktive Systeme:

- **Antwortbereit (engl. responsive):**
  Ein System muss immer zeitgerecht antworten. Die Antwortbereitschaft ist die Grund-

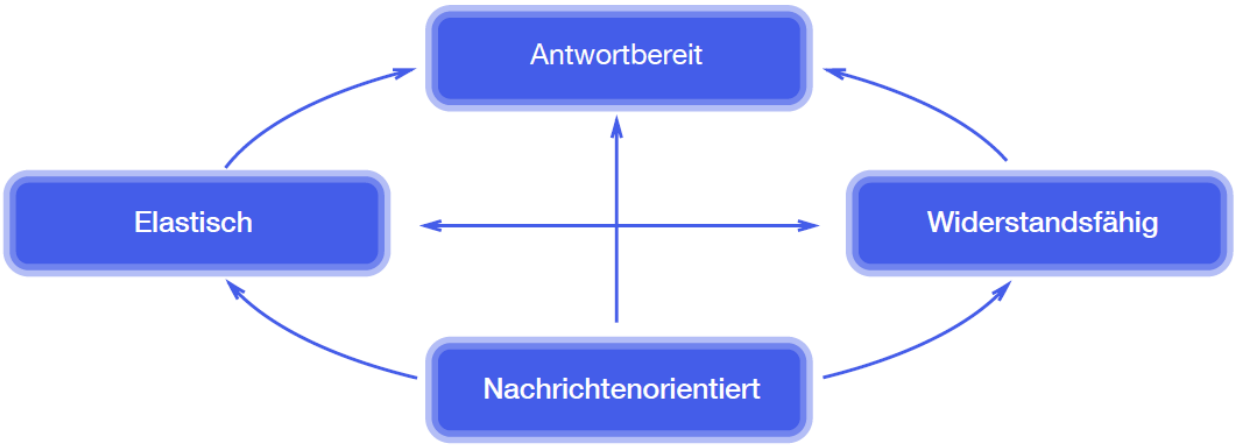

Abbildung 5.1: Die vier relevanten Bestandteile für ein reaktives System.Quelle [Bon14]

lage für die Benutzbarkeit besagten Systems. Ebenso wird um eine Fehlerbehandlung durchführen zu können eine geregelte Antwortbereitschaft vorausgesetzt.

- **Widerstandsfähig (engl. resilient):**
  Ein System muss auch bei Ausfällen die Antwortbereitschaft aufrecht erhalten. Dies wird durch Replikation der Funktionalität, der Isolation von Komponenten sowie dem Delegieren von Verantwortlichkeiten erzielt.

- **Elastisch (engl. elastic):**
  Das System muss bei sich ändernden Lasten die Funktionalität und Antwortbereitschaft aufrecht erhalten. Ressourcen müssen den auftretenden Lasten, ob steigend oder sinkend, angepasst werden können. Ebenso müssen Engpässe innerhalb des Systems unterbunden werden um die Elastizität zu bewahren.

- **Nachrichtenorientiert (engl. message driven):**
  Ein loses System soll zur Kommunikation zwischen den Komponenten auf asynchrone, ortsunabhängige Nachrichtenübermittlung zurück greifen. Somit ist nicht relevant auf welchen Rechner die einzelnen Komponenten ausgeführt werden, wodurch wiederum eine gute Skalierbarkeit erreicht werden kann.

Das Manifest beschreibt jedoch nicht den Zusammenhang von Reactive Systems zum Reactive Programming. Wohl aus diesem Grund hat Jonas Bonér einen weiteren Artikel verfasst, der sich dieser Thematik annimmt [Bon]. Die grundlegenden Unterschiede wurde in dem Artikel wie folgt zusammengefasst:

- Reactive Programming ist eine Teilmenge von reaktiven Systemen auf Implementierungsebene

- Reactive Programming liefert Leistung und effektive Ressourcennutzung auf Komponentenebene, speziell für Softwareentwickler.
- Reaktive Systeme hingegen bieten Robustheit und Elastizität auf Systemebene zur Gestaltung von Cloud-kompatiblen oder verteilten Anwendungen, speziell für Softwarearchitekten oder DevOps, also der Möglichkeit eine schnelle und qualitativ hochwertige Software zu entwickeln und auszurollen.
- Es ist von großem Vorteil Reactive Programming innerhalb der Bestandteile von reaktiven Systemen zu verwenden.
- Es is ebenso von Vorteil reaktive Systeme zur Interaktion zwischen reaktiv programmierten Komponenten zu verwenden.

Wie aus diesen Punkten klar wird, liegt also der genaue Unterschied zwischen Reactive Programming und Reactive Systems darin, aus welcher Perspektive die Betrachtung stattfindet. Architektonisch wird von Systemen gesprochen, die innerhalb und zur Interaktion mit anderen reaktiv reagiert. Betrachtet man das Ganze aus Entwicklersicht in Bezug auf eine Komponente, kann diese unter Zuhilfenahme von Reactive Programming für ein asynchrones, paralleles Verhalten entwickelt werden. Wie Jonas Bonér schon schreibt, ist das Zusammenspiel beider sehr oft hilfreich um das gewünschte Ziel zu erreichen. Ein weiterer wichtiger Punkt ist die Unterscheidung von *ereignisorientiert* zu *nachrichtenorientiert*. Nachrichten werden auf Systemebene genutzt. im Gegensatz zu Ereignissen sind Nachrichten klar an einen Empfänger adressiert. Ereignisse treten auf und müssen beobachtet werden um das gewünschte Resultat zu erzielen. Nachrichten sind somit gut geeignet bekannte Empfänger in einem verteilten System, zum Beispiel über das Netzwerk, zu kontaktieren. Innerhalb der Komponente besteht die Funktion (man denke hier wieder an die N Benutzeroberfläche) oft aus der Reaktion auf Ereignisse unterschiedlicher Art die direkt in der Komponente verarbeitet werden sollen. Somit herrscht hier ein ereignisgetriebenes Verhalten.

5.2 Reactive Programming vs. Functional Reactive Programming

Auf der Suche nach einer Definition zu Reactive Programming stößt man oft auf den Begriff Functional Reactive Programming. Teilweise werden die Begriffe sogar synonym verwendet [NC17]. Grundlegend betrachtet man bei den bekannten Programmierparadigmen zwischen imperativen und deklarativen Paradigmen, wobei zum Beispiel die objektorientierte Programmierung im imperativen und funktionale Programmierung im deklarativen Bereich angesiedelt wird. Funktionale Programmierung zeichnet vor allem die Nutzung des Lambda-Kalküls aus [lam]. Ein Lambda-Ausdruck wird in der Mathematik wie folgt dargestellt:

$$\lambda x.x + 2$$

Dieser Ausdruck besagt, dass jeder Wert x auf $x+2$ abgebildet wird. Statt Rechenanweisungen werden Programme als Menge von Definitionen beschrieben, die mathematisch eine Abbildung von Eingabedaten auf Ausgabedaten darstellt und gleichzeitig selbst aus Funktionsaufrufen zusammengesetzt sind [fpw]. Wird eine funktionale Programmiersprache nun um den Faktor Zeit erweitert kann ein reaktives Verhalten erreicht werden. Diese Art der Realisierung nennt man Functional Reactive Programming [hsk]. Dieser Zeitfaktor zeigt sich im Funktionalen in der kontinuierlichen Wertänderung im Laufe der Zeit. Die Verwechslung der Begriffe entsteht wenn man sich zum Beispiel Java 8 anschaut. Mit Java 8 nahm die Klasse der Funktionen sowie die Lambda-Ausdrücke Einzug in die objektorientierte Programmiersprache Java. In Java-Code werden Lambda-Ausdrücke wie folgt dargestellt:

```
List< Integer > list = Arrays.asList( 1, 2, 3, 4, 5 );
list.forEach( element -> System.out.println( "Number" + element ) );
```

Listing 5.1: Lambda Beispiel in Java

In Listing 5.1 wird jedes der Listenelemente auf die Konsolenasugabe mit konkateniertem String abgebildet. Dadurch wird ein ähnliches Verhalten der funktionalen Programmierung reproduziert. Somit sind funktionale Eigenschaften zwar in Java vorhanden, trotzdem kann Java nicht als funktionale Programmiersprache verstanden werden. Wird auch hier nun der Faktor Zeit mit betrachtet, kann darin der essentielle Unterschied ausgemacht werden. Wird beim Functional Reactive Programming von einer Wertänderung im Laufe der Zeit gesprochen, ist beim Reactive Programming von konkreten Werten zu sprechen die im Laufe der Zeit ausgegeben werden. Also findet keine Änderung des Wertes statt, es werden neue konkrete Werte über einen Zeitraum weiter gereicht.

Kapitel 6

Einführung in Reactive Programming mit RxJava2

ReactiveX ist ein Open Source Projekt und vereint reaktive Bibliotheken für viele Sprachen und Frameworks unter einem Dach. Die Bibliotheken halten sich an den gleichen Aufbau und Benennung um eine gewisse Konsistenz zu schaffen. Die erste dieser Bibliotheken wurde unter Leitung von Erik Meijer bei Microsoft entwickelt und nennt sich Rx.NET, als Erweiterung zum .NET Framework von Microsoft. Das Kürzel *Rx* steht für Reactive Extentions, also reaktive Erweiterungen. Folgend wurde von Ben Christensen und Jafar Husain als Entwickler bei Netflix die Erweiterung für Java geschrieben und auf GitHub veröffentlicht. Es folgten viele weitere Implementierungen für zum Beispiel Ruby, Python oder JavaScript. Die Bibliothek der Reactive Extensions ist sehr klein, bringt jedoch die nötigen Eigenschaften mit, um asynchrone und nicht blockierende Anwendungen zu entwickeln. ReaktiveX sagt über sich, dass die besten Ideen des Beobachtermuster, des Iteratormusters sowie der funktionalen Programmierung hinter ihrer Art der Implementierung steckt. Die Schon in Java vorhandenen Klassen wurden überlagert und für ein reaktives Verhalten gerüstet. Aktuell wird RxJava als Version 1 und Version 2 entwickelt. In Version 2 wurden Änderungen vorgenommen um den Vorgaben der Asynchronität und der Verwendung des Back Pressures der Reactive Streams Initiative zu genügen. Da es sich nicht nur um eine Erweiterung sondern um eine Neuentwicklung parallel zu Version 1 handelt, sind im Moment beide Versionen im Entwicklungsstatus. RxJava2 wird in dieser Arbeit als Referenzbibliothek verwendet.[rxg].

Der zentrale Baustein dieser Bibliothek ist die Observable-Klasse im Zusammenspiel mit dem Observer Interface. Ein Observable repräsentiert einen Data- beziehungsweise Eventstream. Es ist für das Push-Verfahren konzipiert (reaktiv) kann aber auch mit dem Pull-Verfahren verwendet werden (interaktiv) [NC17]. Es folgt nun eine kurze Schilderung wie die wichtigsten Eigenschaften erreicht werden bevor die eigentlichen Struktur, im Genauen eine Beschreibung der Basisklassen der Bibliothek, veranschaulicht wird. Diese Eigenschaften wurden vom bereits erwähnten Ben Christensen in Kapitel 1 innerhalb des Buches von Tomasz Nurkiewicz beschrieben [NC17].

6.1 Synchronität und Asynchronität

Bei dem bisher gelesenen wird schnell klar, dass die Asynchronität ein essentieller Bestandteil sein muss. Jedoch ist das Observable standardmäßig synchron implementiert und ein asynchrones Verhalten muss explizit gefordert werden. Erfolg eine Registrierung(Subscription) eines Observers an einem Observable wird die Weitergabe der Elemente des Streams auf dem Thread des Observers ausgeführt. Ebenso finden die Bulk Operations, also Transformation, Modifikation und Komposition der Elemente oder Streams grundsätzlich synchron statt. Werden also Daten zum Beispiel aus einem Cache geladen und über den Stream zur Verfügung gestellt ist der Standardweg des synchronen Vorgehens vollkommen richtig um den Overhead der expliziten Asynchronität zu umgehen. Finden aber Abfragen zum Beispiel über eine Netzwerkressource statt, die unterschiedliche lange Latenzen aufweist, kann es notwendig sein, die Anfragen auf weiteren Threads auszuführen. Dies kann mittels eigens erstellte Threads, Threadpools oder Schedulers umgesetzt werden. Somit werden die Callback-Methoden des Observers von dem zusätzlichen erstellten Thread aufgerufen und der eigentliche Observer Thread wird nicht weiter blockiert.

6.2 Parallelisierung und Nebenläufigkeit

Parallelisierung und Nebenläufigkeit sind vor allem auf Systemebene zu betrachten. Nebenläufigkeit besagt, dass mehrere Tasks gleichzeitig ausgeführt werden können. Findet diese Bearbeitung auf physisch unterschiedlichen Kernen zur gleichen Zeit statt, spricht man von paralleler Ausführung. Ist jedoch nur ein Kern vorhanden, oder Tasks werden trotz mehrerer Kerne auf nur einem ausgeführt, findet eine sequentielle Abarbeitung statt. Der Kernel des Systems bringt hierfür die Funktion des *time-slicing* mit. Diese regelt wann welcher Task Rechenzeit zugewiesen bekommt. Um dieses Verfahren in Verbindung mit den Observables zu bringen gilt, dass ein Observable Stream immer serialisierbar und thread-safe sein muss. Die Callback-Methoden des Subscribers können also nie zeitgleich aufgerufen werden. Dieser Vertrag muss eingehalten werden, um die Funktionalität des Observables zu gewährleisten.

6.3 Push und Pull

Wird Pulling von Objekten einer Liste über das Iterable Interface durchgeführt, so wird im Gegenzug Pushing via Observable realisiert. Beide Schnittstellen bieten die gleiche Funktionalität, nur der Datenfluss findet in die entgegengesetzte Richtung statt. Durch diese Dualität können beide Vorgehen äquivalent verwendet werden. Will man ein weiteres Objekt einer Liste über den Iterator abfragen, wird die next()-Methode aktiv aufgerufen und wenn vorhanden wird ein weiteres Objekt dem Verbraucher zurück gegeben. Hingegen wird bei der Verwendung von Observables die Daten des Streams mit der onNext()-Methode des Verbrauchers gepusht. Wie die Tabelle 6.1 zeigt gilt dies ebenso beim Auftreten eines Fehlers oder beim Erreichen

Pull (Iterable)	Push (Observable)
T next()	onNext(T)
throws Exception	onError(Throwable)
returns	onComplete()

Tabelle 6.1: Vergleich zwischen Funktionalität der Iterable- und Observable-Schnittstelle.

des Endes der Datenquelle. Die Verbindung zwischen Observable und Observer findet über ein Subscription statt. Damit werden die beiden zu einem Paar gebunden und die entsprechenden Methoden des Observers können nun von dem Stream angesprochen werden. Dies beschreibt auch noch eine weitere Eigenschaft. Ein Observable publiziert nur die Ereignisse wenn es jemanden gibt der diese Ereignisse auch anfordert. Dies wird auch als *Lazy Initialization* bezeichnet. Somit wird das Arbeiten durch das Subscriben und nicht durch das Erstellen eines Observables angestoßen. Im Vergleich dazu kann ein Objekt vom Typ Future betrachtet werden. Wird ein Future erstellt, wird auf ein Ergebnis gewartet, welches direkt und einmalig asynchron ausgeführt wird und innerhalb des Futures zur Verfügung steht sobald das Ereignis abgeschlossen ist. Ein mehrfaches Ausführen eines Futures ist nicht möglich, anders als beim Observable wo zu jeder Zeit ein weiterer Subscriber hinzu kommen kann. Durch dieses Verhalten wird die Wiederverwendbarkeit des Observables erreicht.

6.4 Observable

Es wurde in den vorherigen Abschnitten schon einiges über das Observable-Objekt gesagt, doch der Vollständigkeit halber wird hier seine Eigenschaften und Fähigkeiten zusammengefasst beschrieben. Es handelt sich hierbei um das Kernelement der RxJava Bibliothek.

```
public static void main(String[] args)
{
 Observable<String> obs0 = Observable.just("Java", "Kotlin", "Scala");
 obs0.subscribe(s -> {System.out.println(s);},
   t -> {t.printStackTrace();},
   () -> {System.out.println("finish");} );
}
```

Listing 6.1: Beispiel Observable Initialisierung und Subscription

Ein Ereignisstrom über einen Zeitraum wird von einem Observable repräsentiert. Dieser Ereignisstrom ist jedoch variabel. Weder muss die Anzahl der auftretenden Ereignisse bekannt sein, noch müssen diese in einem geregelten Intervall auftreten. Ebenso wenig muss der Zeitraum begrenzt sein, was bedeutet, dass es sich auch um unendliche Eventstreams handelt kann. Die Tabelle 6.1 zeigt die Methodenaufrufe die eine Observable auslösen kann. Somit wird auch klar, dass nur drei Arten von Events auftreten können. Eine beliebige Anzahl von *onNext(T)*-Methodenaufrufen mit dem Eventtyp T, das einmalige Event der Fertigstellung mit dem *on-*

Complete()-Aufruf oder das Aufkommen eines Fehlers mit der *onError(Throwable)*-Methode. Somit gilt:

$$OnNext * (OnComplete|OnError)?$$

Daraus resultiert, dass entweder ein gewolltes Ende, oder ein unvorhergesehener Fehler auftritt, aber niemals beides[1]. Schaut man sich nun Listing 6.1 an, sieht man den Ablauf. Ein Observable wird initialisiert, hier mit drei Elementen, also ein endlicher Stream. Durch das *subscribe* wird anonym ein Subscriber erzeugt. Bei dem ersten Lambda-Ausdruck handelt es sich um die *onNext()*-Methode. Weiter geht es mit *onError()* und *onComplete()*. Man sieht bei einem fehlerfreien Durchlauf die drei Elemente auf der Konsole und ebenso die Ausgabe für das beenden des Streams. Würde ein Fehler auftreten würde die Ausgabe der Elemente beendet werden und der Stacktrace würde ausgegeben werden, statt des *finish*-Strings. Auch wurde schon ein wichtiger Punkt angeschnitten, nämlich dass zu jeder Zeit ein neuer Subscriber hinzu kommen kann. Jeder Subscriber erhält einen separaten Observable-Stream. Observables mit dieser Eigenschaft nennt man auch *Cold Observable*.

```
System.out.println( "--------Cold Observable--------" );
Observable< Long > obs = Observable.interval( 1, TimeUnit.SECONDS );
obs.subscribe( s -> System.out.println( s ) );
Thread.sleep( 1000 );
obs.subscribe( s -> System.out.println( s ) );
Thread.sleep( 5000 );
```

Listing 6.2: Beispiel Cold Observable

Wie hier in Listing 6.2 zu sehen werden zwei Objekte des Observable-Streams erstellt. Es ist zu beachten, dass somit jeder Subscriber einen eigenen Timer bekommt. Will man nun aber die beiden Rechenoperationen immer auf den selben Wert durchführen wird es auf diesem Wege nicht funktionieren. Somit muss es die Möglichkeit geben diese Informationen zu teilen.

```
System.out.println( "--------Hot Observable--------" );
Observable< Long > observable = Observable.interval( 1, TimeUnit.SECONDS )
 ;
ConnectableObservable< Long > connectable = observable.publish();
connectable.connect();
Thread.sleep( 5000 );
connectable.subscribe( s -> System.out.println( s ) );
Thread.sleep( 5000 );
connectable.subscribe( s -> System.out.println( s ) );
Thread.sleep( 10000 );
```

Listing 6.3: Beispiel Hot Observable

Der Methodenaufruf *publish()* ist hier der Schlüssel. In Listing 6.3 sieht man diesen Aufruf, nachdem das Intervall gestartet wird. Durch diesen Aufruf erhält man eine besondere Form

[1] pdf in bib aufnehmen und zum regex binden

des Observables, das *ConnectableObservable*. Grundlegend hat sich am Aufbau des Observables nichts geändert, jedoch beginnt das ausgeben der Daten nicht durch eine Subscription, sondern durch das Aufrufen der *connect()*-Methode. Durch diese Eigenschaft wird das Observable zu einem sogenannten *Hot Observable*. Schaltet sich nun ein Subscriber auf den Stream beginnt er mit den Werten der zu diesem Zeitpunkt ausgegeben wird. Die schon vergangenen Events sind für den Subscriber nicht mehr zu erreichen. Es macht zum Beispiel Sinn, wenn eine Mausbewegung getrackt wird, da neue Subscriber nicht wissen müssen wo sich der Mauszeiger vorher befand, denn es ist nur wichtig wo sich der Zeiger aktuell befindet. Je nach Problemstellung ist das eine oder das andere von Vorteil. Ist es entscheidend eine konsistente und vollständige Abarbeitung von Events durchzuführen greift man auf Cold Observables zurück. Handelt es sich um einen Stream über den man keine konkrete Kontrolle hat, wie ein Stream für Ein- und Ausgabe, wird man meist Hot Observables verwenden.
Ein Punkt welcher das Observable nicht vollständig abdeckt ist der Back Pressure. Es sind keine Möglichkeiten enthalten aktiv auf Back Pressure reagieren können. Wie schon erwähnt ist seit Version 2 der RxJava Bibliothek eine Implementierung diese Möglichkeit nach Richtlinien der Reactive Stream Initiative vorhanden. Dies wurde in der Klasse *Flowable* implementiert.

6.5 Flowable

Diese Klasse ist nach den Kriterien der Reactive Streams realisiert, was wiederum heißt, eine asynchrone Bearbeitung ohne blockender Back Pressure muss möglich sein. Durch die Implementierung *"implements org.reactivestreams.Publisher<T>"* des Publisher Interfaces wird die Handhabung des Back Pressures ermöglicht, und da auch die Funktionalität des Observables vorhanden ist, kann eine Bearbeitung asynchron stattfinden. In der offiziellen Dokumentation von RxJava2 wird beschrieben wann es sich besser eignet auf Observables oder aber auf Flowables zu setzt [rxd]. Wann sollte man Observables nutzen:

- Wenn im Datenfluss nicht mehr als 1000 Elemente auftreten, es somit als keine Chance für einen *OutOfMemoryError* gibt.
- Wenn GUI Events behandelt werden. Diese können nicht durch ihr unregelmäßiges Aufkommen schlecht gepuffert werden. Außerdem ist das Auftreten meist geringfügig. Elemente in der Auftrittshäufigkeit von maximal 1000 Hz sollte zu bewältigen sein.
- Wenn die Plattform keine Streams unterstützt, also die Java Version diese noch nicht bereit stellt. Grundsätzlich bringen Observables weniger Overhead mit sich.

Wann sollte man Flowables nutzen:

- Wenn mehrere Tausende Elemente generiert werden und die Möglichkeit vorhanden ist mit der Quelle zu kommunizieren um bei fehlendem Back Pressure die Frequenz zu regeln.

- Beim Lesen von der Festplatte. Dies geschieht blockierend und mit dem Pull-Ansatz, was sich hervorragend eignet um die Elemente als Back Pressure zu puffern.
- Beim Lesen aus der Datenbank. Auch das geschieht via Pull und blockiert. Also wieder von Vorteil mit einem Puffer zu arbeiten.
- Kommunikation über ein Netzwerk. Hierbei kann, sofern unterstützt, eine logische Menge an Elementen angefragt werden welche im Puffer gesichert werden und nach und nach abgearbeitet werden können.
- Weiter blockierende auf Pull-basierenden Datenquellen deren API eventuell in Zukunft auch auf ein reaktives Verhalten umgestellt werden. Hier liegt der Vorteil in der Reactive Streams Konvention die auch in Zukunft verwendet werden wird.

Jedes Observable und Flowable Objekt besitzt eine Puffer um nicht verarbeitete Daten zwischen zu speichern. Diese Größe beträgt bei Flowables standardmäßig 128 Objekte, kann jedoch geändert werden. Bei Observables ist dieser Puffer dynamisch und kann wenn nötig den kompletten Speicher beanspruchen. Betrachtet man das Listing 6.5 sieht man den Aufruf der *onBackpressureBuffer()*-Methode. Diese Methode bietet ein Observable-Objekt nicht. Es handelt sich um eine Implementierung des Publisher-Interfaces. Hiermit kann direkt eine Puffergröße für jedes Flowable festgelegt werden. Kann nun ein Subscriber also nur eine gewisse Anzahl von Elementen verarbeiten, kann über den bereits erwähnten Feedback-Kanal des Flowable informiert werden, die Objekte wenn möglich langsamer auszugeben. Die Verzögerung wird durch den zusätzlichen Puffer der *onBackpressureBuffer()*-Methode ermöglicht. In diesem Beispiel wäre dieser Aufruf nicht notwendig doch zur Veranschaulichung wurde er hinzugefügt. Da auch die Anzahl der Elemente bekannt ist, kann der Puffer natürlich genau auf die Anzahl der Elemente festgelegt werden.

```
Flowable< String > flowing = Flowable.just( "Java", "Kotlin", "Scala" );
flowing.onBackpressureBuffer( 3 ).subscribe( s -> {
 System.out.println( s );
}, t -> {
 t.printStackTrace();
}, () -> {
 System.out.println( "finish" );
} );
```

Listing 6.4: Beispiel Flowable mit BackPressureBuffer

Um nun die Verwendung des Feedback-Kanals noch zu betrachten, wurde das Beispiel um einen *DefaultSubscriber* erweitert.

```
DefaultSubscriber< String > sub = new DefaultSubscriber< String >()
{
 @Override
```

```
  public void onStart()
  {
   request( 1 );
  }

  @Override
  public void onNext( String t )
  {
   System.out.println( t );
   request( 1 );
  }
 flowing.subscribe( sub );
```

Listing 6.5: DefaultSubscriber mit Back Pressure

Durch den Aufruf der *onStart()*-Methode wird direkt zu Beginn der Subscription durch den *request(1)* Aufruf die verträgliche Menge des Subscribers weiter gegeben. Ebenso findet diese Anfrage auch beim Aufruf jedes *onNext()* Callbacks statt. Durch diesen Aufruf erfährt das Flowable, dass immer nur ein Element an den Subscriber ausgegeben werden soll. Die beiden weiteren Elemente würde nun im BackPressureBuffer unterkommen und bei weiterer Anfrage zur Verfügung gestellt werden. Wie schon erwähnt ist dies in diesem kurzen Beispiel jedoch nicht nötig und dient nur zur Erklärung der Funktionsweise.

6.6 Single

Wo Observables oder Flowables eine Menge von Null bis endliche viele Elemente ausgeben, steht Single, wie der Name schon sagt, dafür, dass immer nur einmal auf eine Event reagiert werden kann. Somit sind auch die drei bekannten Methoden zur Datenausgabe hier überflüssig. Stattdessen sind nur zwei Methoden vorhanden, *onSuccess()* und *onError()*. Auch hier gilt, dass immer nur eine der Methoden aufgerufen werden kann. Also eine erfolgreiche Datenausgabe oder eine fehlerbehaftete. Anschließend werden die Subscriptions gelöst und das Singe wird beendet. Auch hier sind wieder eine Reihe von Bulk Operations möglich.

```
public static void main(String[] args)
{
 Single<String> single = Single.just("Covfefe");
 single.map(s -> s.length()).subscribe(
   element -> {System.out.println("Length: " + element);},
   error -> {error.printStackTrace();}
   );
}
```

Listing 6.6: Beispiel eines Singles

In Listing 6.6 sieht man die Verwendung eines Single-Objekts. Es kann nur mit einem Objekt erstellt werden, die Bearbeitung dieses Elements findet wieder mit einer Auswahl an bekannten Operation statt. In der *subscribe()*-Methode sieht man die zwei Lambda-Ausdrücke die wieder die möglichen Methodenaufrufe, also bei Erfolg oder Fehlerfall, die ausgeführt werden können. Ein Beispiel für die Verwendung von Single findet sich zum Beispiel bei einer HTTP-GET Anfrage. Man möchte ein Objekt abrufen, und sobald es angekommen ist, soll es verwendet werden. Da dieser Aufruf für immer nur ein Objekt stattfindet, lohnt sich hier die Verwendung des Singles. Geht die Anfrage raus muss nicht gewartet werden, und sobald das Ergebnis eingetroffen ist, wird die *onSuccess()*-Methode aufgerufen oder eben im Fehlerfall die *onError()*-Methode.

6.7 Subject

Als Subject versteht man ein Objekt, dass sowohl als Observable und auch als Observer fungieren kann. Es kann an ein Observable subscriben, diese Daten verarbeiten und seinen eigenen Subscribern weiterleiten.

```
public class SubjectExample
{
 public static void main(String[] args)
 {
  Observable<String> obs0 = Observable.just("Java", "Kotlin", "Scala");
  Subject<String> subj = PublishSubject.create();
  subj.map(s -> s.toUpperCase()).subscribe(s -> {System.out.println(s);});
  obs0.subscribe(subj);
```

Listing 6.7: Beispielverwendung eines Subjects

Um das Vorgehen bei Verwendung eines Subject zu veranschaulichen betrachetet man Listing 6.7. Es wird wieder das schon bekannte Observable-Objekter stellen, hinzu kommt ein Subject, hier speziell PublicSubject. Dieses Subject bekommt eine Subscription in welcher alle String-Werte in Großbuchstaben umgewandelt und anschließend auf der Konsole geschrieben werden. Wenn nun das Subject selbst an dem vorhanden Observable subscribed, werden die von diesem ausgegebenen Werte gemapped und an die Subscriber des Subject weitergegeben. Zusätzlich zum PublicSubject stehen noch drei weitere Arten zur Verwendung bereit:

- AsycSubject: Ein AsyncSubject gibt seinen Subscribern nur den letzte Wert der Quelle weiter, und das auch nur falls das Quell-Observable beendet wurde. Endet es durch einen Fehler wird die Fehlermeldung direkt an die Subscriber des Subjects weiter gereicht.
- BehaviorSubject: Ein BehaviorSubject beginnt sobald es einen Subscriber verzeichnet kann mit der Übermittlung des letzten Wertes der Quelle, oder mit einem Standardwert falls von der Quelle noch nichts ausgegeben wurde. Anschließend werden alle weite-

ren Werte weitergegeben. Auch hier wird bei Auftreten einen Fehler die Meldung direkt durchgereicht.

- PublishSubject: Ein PublishSubject überträgt genau die Werte, die von seiner Quelle in dem Zeitraum, in welchem auch eine Subscription vorliegt, ausgegeben wurden. Alles vorherige wird nicht übermittelt. Somit liegt ein Verhalten eines Hot-Observables vor.

- ReplaySubject: Ein ReplaySubject überträgt an jeden Subscriber alle Werte die jemals von seiner Quelle ausgegeben wurden. Dies ist mit Vorsicht zu genießen da hier natürlich ein Back Pressure Problem nahe liegt.

Auch hier wurde auf Grund der Konformität zur Reactive Streams Initiative eine weitere Klasse implementiert, welche die Möglichkeit bietet aktiv auf Back Pressure zu reagieren. Die Processor-Klasse.

6.8 Processor

Was das Flowable für das Observable ist, stellt der Processor für das Subject dar. Eine Implementierung um den Anforderungen der Reactive Streams gerecht zu werden. Die Kompatibilität und das Handling des Back Pressures stehen auch hier wieder im Vordergrund. Die vier Ausprägungen des Subject sind auch bei der Processor-Klasse wiederzufinden. Zusätzlich wurde noch der *UnicastProcessor* implementiert. Hier wird nur genau ein Subscriber zugelassen über die komplette Existenz des Processor-Objekts. Wenn versucht wird dagegen zu verstoßen wird eine *IllegalStateException* geworfen.

```
public class ProcessorExample
{
 public static void main(String[] args)
 {
  Flowable<String> flow = Flowable.just("Java", "Kotlin", "Scala");
  PublishProcessor<String> proc = PublishProcessor.create();
  proc.onBackpressureBuffer(1).map(s -> s.toUpperCase()).subscribe(s -> {
   System.out.println(s);});
  flow.subscribe(proc);
 }
```

Listing 6.8: Beispielverwendung eines Processors

In Listing 6.8 ist ein einfaches Beispiel zu finden. Die Ähnlichkeit zur Subject-Klasse ist unverkennbar. Jedoch werden hier eben die Flowables verwendet, und das Back Pressure kann aktiv kontrolliert werden. Ist also bekannt das Schnittstellen nach den Richtlinien der Reactive Streams Initiative entwickelt wurden, ist es lohnenswert auf die Verwendung von Flowables und Processors zu setzten um auf einfachem Wege eine Kompatibilität zu ermöglichen.

6.9 Schedulers

Im Abschnitt zu Parallelisierung und Nebenläufigkeit wurde schon beschrieben, dass jedes Observable-Objekt serialisierbar und thread-safe sein muss. Dadurch ergibt sich die Möglichkeit Observables oder einzelne Operationen auf unterschiedlichen Threads auszuführen. RxJava bringt hierfür einige Scheduler mit, die unterschiedliche Arten von Threads oder Threadpools mitbringen um Multithreading zu ermöglichen. Um zum Beispiel eine Berechnung auf einem anderen Thread auszuführen bringt die Observable-Klasse zwei Methoden mit: *subscribeOn()* und *observeOn()*. Beide Funktionen nehmen einen Scheduler als Parameter entgegen, jedoch unterscheiden sie sich in ihrer Funktionsweise.

- subscribeOn: Wird diese Funktion aufgerufen wird jedes Element des Observable vollständig auf den vom Scheduler bereit gestellten Thread bearbeitet. Finden mehrere Aufrufe statt wird immer nur der erste Aufruf durchgeführt, da die Ausführung des Observable durch die Thread-Sicherheit immer nur auf einem Thread ausgeführt werden.

- observeOn: Hier wird nicht der Observable Stream an sich betrachtet sondern die einzelnen Elemente. Somit ist es möglich, dass Operationen und Subscriptions jedes Mal auf einem neuen Thread stattfinden können. Dies heißt wiederum, dass jeder Aufruf von *observerOn* dazu führt, dass die folgenden Operationen auf dem neuen Thread ausgeführt werden.

```
observable.subscribeOn( Schedulers.newThread() ).map( mapping ).
 subscribeOn( Schedulers.newThread() )
  .subscribe( s -> {
   System.out.println( s + ": " + Thread.currentThread().getName() );
```

Listing 6.9: Beispiele für Scheduling mit zwei subscribeOn-Aufrufen

In Listing 6.9 sieht mein ein Observable welches eine Operation, eine Subscription und zwei *subscribeOn*-Methodenaufrufe beinhaltet. In diesem Beispiel wird alles auf einem Thread ausgeführt. Dieser Thread wird von dem ersten auftretenden *Schedulers.newThread()* erstellt und jeder weitere Aufruf wird ignoriert, da das komplette Observable nur auf einen Thread ausgeführt werden kann. Betrachtet man nun das Listing 6.10 erkennt man die gleiche Struktur, jedoch wird in diesem Fall die *observeOn*-Methode zweimal verwendet. Wie beschrieben wird hier jeder einzelne Bestandteil des Observables als einzelner Task behandelt und kann somit immer auf einem neuen Thread ausgeführt werden. Hier wird in beiden Methodenaufrufen ein einer Thread erzeugt. Zur Veranschaulichung betrachtet man Tabelle 6.2. Die erste Spalte zeigt den Programmfluss des Listings 6.9. Das Observable wird auf einem Thread immer pro Element durchlaufen. Im Vergleich dazu kann man in der zweiten Spalte den Programmfluss von Listing 6.10 sehen. Auf dem ersten Thread findet zuerst die Operation für das mapping von jedem Element statt, und auf dem zweiten Thread die danach folgende Operation, in diesem Fall die Subscription.

subscribeOn	observeOn
mapping: RxNewThreadScheduler-1	mapping: RxNewThreadScheduler-1
java: RxNewThreadScheduler-1	mapping: RxNewThreadScheduler-1
mapping: RxNewThreadScheduler-1	mapping: RxNewThreadScheduler-1
kotlin: RxNewThreadScheduler-1	java: RxNewThreadScheduler-2
mapping: RxNewThreadScheduler-1	kotlin: RxNewThreadScheduler-2
scala: RxNewThreadScheduler-1	scala: RxNewThreadScheduler-2

Tabelle 6.2: Ablauf der Threads bei Verwendung von subscribeOn und observeOn

```
observable.observeOn( Schedulers.newThread() ).map( mapping ).observeOn(
 Schedulers.newThread() ).subscribe( s -> {
 System.out.println( s + ": " + Thread.currentThread().getName() );
} );
```

Listing 6.10: Beispiele für Scheduling mit zwei observeOn-Aufrufen

Somit muss je nach Vorhaben und Anforderung die passende Art von Scheduling betrieben werden.

6.10 Operationen

Es wurde schon einiges über die auf Observable anwendbaren Bulk-Operations geschrieben und in manchen Listings auch angewandt, jedoch noch nicht im einzelnen geschildert wie der genaue Ablauf solche einer Operation stattfindet. Um die ganze Mechanik besser zu verstehen, werden folgend vier der gängigen Operationen veranschaulicht. Hierzu werden sogenannte Marble-Diagramme verwendet. Die oberer Linie stellt immer ein Observable-Stream in zeitlichem Ablauf dar, also ähnlich einem Zeitstrahl. Jedes Element, dargestellt durch die farblichen Marbles, wird durch die mittlere Operation geführt und auf das ausgehenden Observable, am unteren Ende des Bildes ausgegeben.

6.10.1 Operation filter()

Wie der Name schon sagt, handelt es sich bei Filter um eine Operation die einen Ausdruck prüft, und sofern das Element gültig ist, wird es weitergeben. Verwendet wir hier der von RxJava mitgebrachte Datentyp *Predicate*. Er beinhaltet eine Methode *test()* welche ein boolean zurück liefert. Somit ist es möglich einfache aber auch komplexere Überprüfungen innerhalb der Filter-Operation auszuführen. Im Bild 6.1 sieht man eine Filterung nach dem Kriterium, dass nur jedes Element, das eine Wert größer 10 repräsentiert, ausgegeben wird. Da jedes Element sequentiell überprüft wird, erhält das untere Observable immer genau dann ein Element wenn es auch zu diesem Zeitpunkt die Prüfung passiert hat. In Java-Code sieht das ganze dann aus

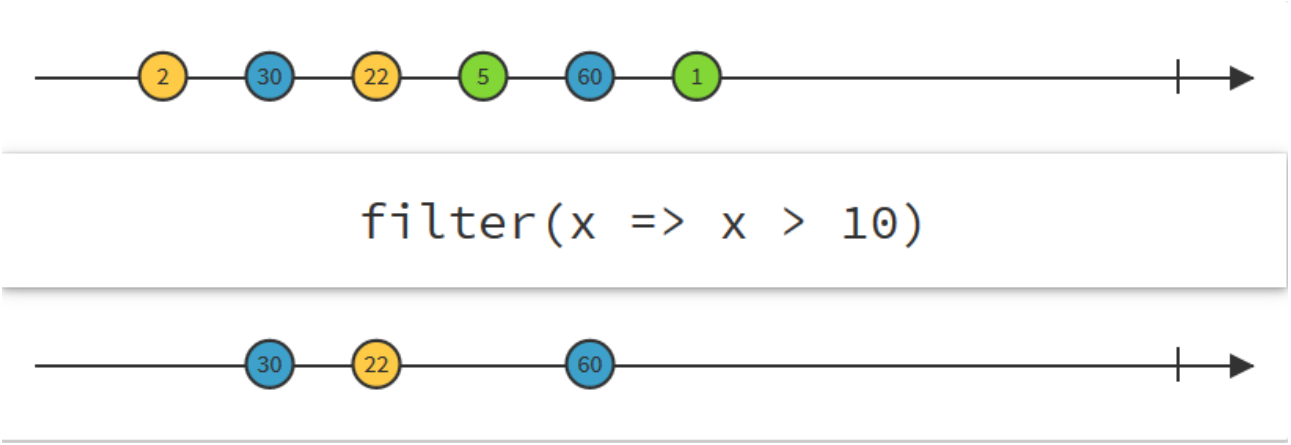

Abbildung 6.1: Datenfluss Filter-Operation

wie in Listing 6.11. Innerhalb des Predicate wird die *test()*-Methode implementiert und die Prüfung ob der Wert größer 10 ist durchgeführt.

```
Observable< Integer > obs0 = Observable.just( 2, 30, 22, 5, 60, 1 );
Predicate< Integer > p = new Predicate< Integer >()
{
 @Override
 public boolean test( Integer arg0 ) throws Exception
 {
  return ( arg0 > 10 );
 }
};
obs0.filter( p ).subscribe( x -> System.out.println( x ) );
```

Listing 6.11: Beispiel Filter Operation

Parametriert wird das Predicate auf den Wert Integer, da nur ein Datentyp übergeben wird, Resultat is immer Datentyp boolean. Das Predicate kann natürlich auch inline als Lambda oder anonym deklariert werden.

6.10.2 Operation map()

Die Map-Operation gilt als Transformationsoperation. Ihre Aufgabe ist es, jeden auftretenden Wert auf einen Funktionswert abzubilden und das Resultat auszugeben. In Abbildung 6.2 sieht man auf dem ursprünglichen Observable drei auftretenden Elemente. Jedes dieser Elemente wird mit 10 multipliziert und danach an das resultierenden Observable ausgegeben.

```
Observable< Integer > obs1 = Observable.just( 1, 2, 3 );
obs1.map( x -> x * 10 ).subscribe( x -> System.out.println( x ) );
```

Listing 6.12: Beispiel Map Operation

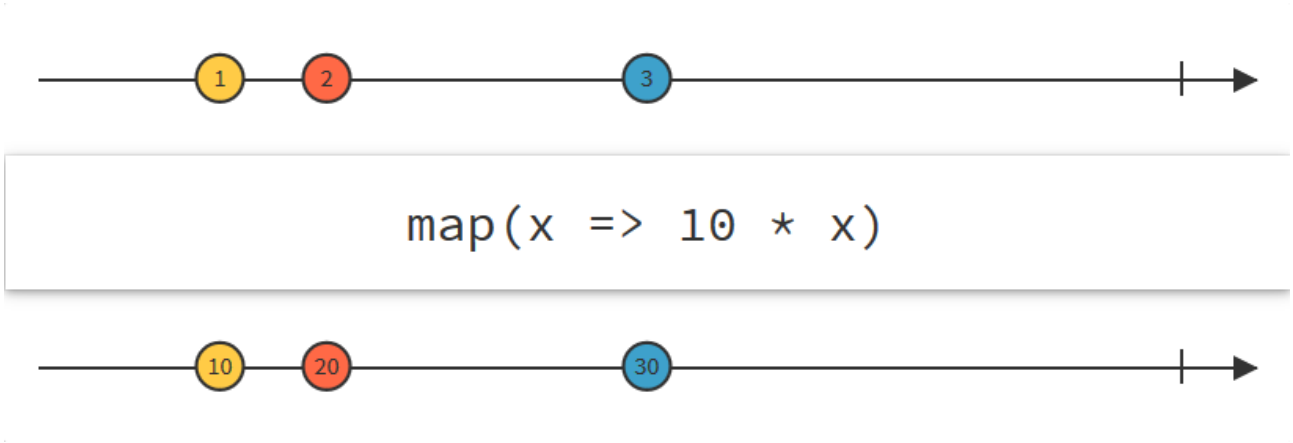

Abbildung 6.2: Datenfluss Map-Operation

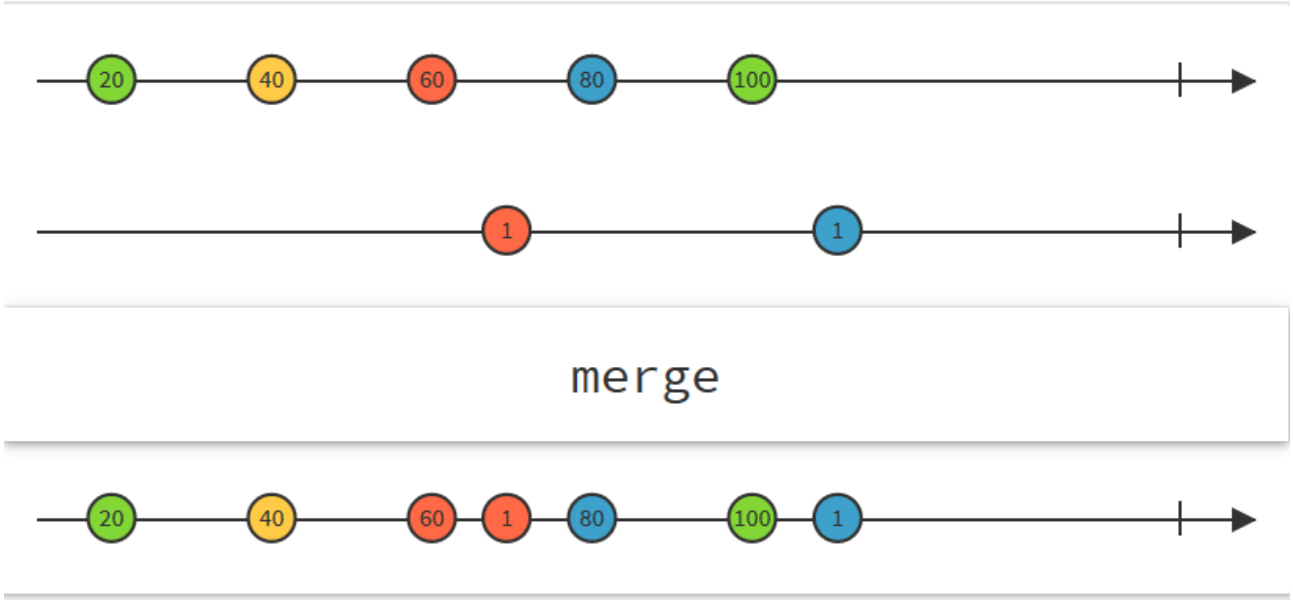

Abbildung 6.3: Datenfluss Merge-Operation

In Programmcode wird es wie erwartet als Lambda-Ausdruck dargestellt. Innerhalb des Mapping wird $x- > x \cdot 10$ abgebildet.

6.10.3 Operation merge()

Merge repräsentiert eine Observable-Komposition. Es handelt sich hierbei um eine recht einfache Variante, denn es werden lediglich alle Elemente aus mindestens zwei Observables in einen Observable-Stream zusammengefasst. Betrachtet man auch hier die passende Abbildung 6.3 wird deutlich wie die Komposition stattfindet. Zu jedem Zeitpunkt zu welchem eines der Observables ein Element ausgibt wird es an das zusammengeführte Observable weitergegeben. Somit wird die Reihenfolge der Elemente rein von dem Faktor Zeit bestimmt.

```
Observable< Integer > obs2 = Observable.just( 20, 40, 60, 80, 100 );
Observable< Integer > obs3 = Observable.interval( 10, TimeUnit.
 MICROSECONDS ).map( i -> 1 );
```

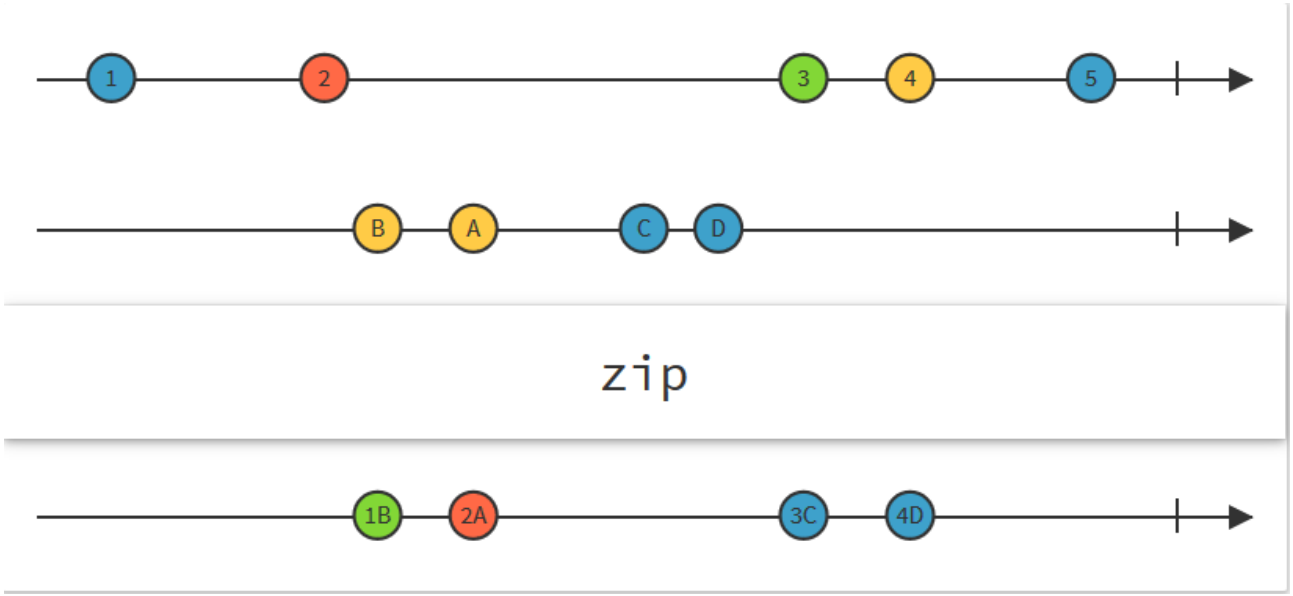

Abbildung 6.4: Datenfluss Zip-Operation

```
Observable.merge( obs2, obs3 ).subscribe( x -> System.out.println( x ) );
```

Listing 6.13: Beispiel Merge Operation

Im Listing 6.13 ist die entsprechende Implementierung zu sehen. Eines der Observables beinhaltet die Werte in zwanziger Schritten, das weitere produziert jede Nanosekunde ein Element. Das Mapping muss stattfinden, da immer nur Elemente eines Datentyps in ein Observable zusammen geführt werden können. Wie vorab schon erwähnt ist die Unendlichkeit bei der Komposition von Streams mit Vorsicht zu genießen hinsichtlich der Verwertbarkeit des Observerables und der Performanz einer Anwendung.

6.10.4 Operation zip()

Bei dem *zip*-Operator handelt es sich ebenfalls um eine Komposition von Observables. Wie die Funktionsweise eines Reißverschlusses werden die Elemente miteinander verbunden. Abbildung 6.4 erläutert diese Weise genauer. Die Elemente der beiden Observables können unterschiedlichen Typs sein und können zu unterschiedlichen Zeitpunkten ausgegeben werden. Zip geht nun so vor, dass ab dem Moment, ab welchem mehrere Observables zusammen gelegt werden, immer die nächsten Elemente miteinander verknüpft werden und als Resultat ein neuer beziehungsweise anderen Datentyp entstehen kann.

```
Observable< Integer > obs4 = Observable.just( 1, 2, 3, 4, 5 );
Observable< String > obs5 = Observable.just( "B", "A", "C", "D" );
Observable.zip( obs4, obs5, ( x, y ) -> y + x.intValue() ).subscribe( z ->
  System.out.println( z ) );
```

Listing 6.14: Beispiel Zip Operation

Um es sich wieder von Entwicklungsseite zu veranschaulichen ist in Listing 6.14 ein einfaches Beispiel dargestellt. Wie in der Abbildung werden einmal Zahlen einmal Buchstaben als Observables repräsentiert und mit dem Zip-Operator verknüpft. Zip muss immer wissen wie die Objekte verbunden werden sollen. Hierzu wird eine entsprechende Funktion implementiert. Hier ist wieder die Verwendung von Lambdas möglich, wie es auch im Listing 6.14 stattfindet, oder es muss eine explizite Funktion definiert werden, als Methode oder anonym.

Kapitel 7

Beispiel: Implementierung eines Systemmonitors

Beschreibung der Funktionen der Anwendung. Überblick über Projekt/Klassenstruktur. Verwendete Tools und Versionen.

7.1 Klassenbeschreibungen SServerSSeite

7.1.1 API für Systemwerte

Framework für die Systemwerte kurz erläutern.

7.2 Klassenbeschreibungen "ClientSSeite

7.2.1 RxJavaFx

Kurzbeschreibung mit Verweis auf gitbook

7.3 Hier eventuell Testing noch beschreiben

Kapitel 8

Evaluierung

Hier Eval schreiben

8.1 Ausblick

Hier Ausblick auf interessante Entwicklungen geben

Literaturverzeichnis

[Amd] Amdahl, Gene M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities: Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California. In: *IEEE SSCS NEWS* 2007. https://people.cs.umass.edu/~emery/classes/cmpsci691st/readings/Conc/Amdahl-04785615.pdf

[Bon] Bonér, Jonas Klang V.: Reactive Programming versus Reactive Systems. https://info.lightbend.com/reactive-programming-versus-reactive-systems.html

[Bon14] Bonér, Jonas Farley Dave Kuhn Roland Thompson M.: the-reactive-manifesto-2.0. (2014). http://www.reactivemanifesto.org/

[floa] *Java 9 Doc - Class Flow.* http://download.java.net/java/jdk9/docs/api/index.html?java/util/concurrent/Flow.html

[flob] *Reactive Programming with JDK 9 Flow API.* https://community.oracle.com/docs/DOC-1006738

[fpw] *Funktionale Programmierung.* https://de.wikipedia.org/wiki/Funktionale_Programmierung

[GHJV11] Gamma, Erich ; Helm Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software.* 39. printing. Boston : Addison-Wesley, 2011 (Addison-Wesley professional computing series). – ISBN 0201633612

[HP85] Harel, D. ; Pnueli, A.: On the Development of Reactive Systems. Version: 1985. http://dx.doi.org/10.1007/978-3-642-82453-1{_}17. In: Apt, Krzysztof R. (Hrsg.): *Logics and Models of Concurrent Systems.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1985. – DOI 10.1007/978–3–642–82453–1_17. – ISBN 978–3–642–82453–1, 477–498

[hsk] *Functional Reactive Programming.* https://wiki.haskell.org/FRP

[Ind15] Inden, Michael: *Java 8 - die Neuerungen: Lambdas, Streams, Date And Time API und JavaFX 8 im Überblick*. 2., aktualisierte und erw. Aufl. Heidelberg : dpunkt-Verl., 2015. – ISBN 9783864902901

[lam] *Lambda-Kalkül*. https://de.wikipedia.org/wiki/Lambda-Kalk%C3%BCl

[Mar05] Martin Fowler: *InversionOfControl*. https://martinfowler.com/bliki/InversionOfControl.html. Version: 2005

[NC17] Nurkiewicz, Tomasz ; Christensen, Ben: *Reactive programming with RxJava: Creating asynchronous, event-based applications*. Sebastopol, CA : O'Reilly, 2017. – ISBN 9781491931653

[rsl] *Reactive Streams 1.0.0 is here!* http://www.reactive-streams.org/announce-1.0.0

[rsm] *Reactive Streams*. http://www.reactive-streams.org/

[rxd] *ReactiveX - What's different in 2.0*. https://github.com/ReactiveX/RxJava/wiki/What\T1\textquoterights-different-in-2.0

[rxg] *RxJava: Reactive Extensions for the JVM - offizielles GitHub Repository*. https://github.com/ReactiveX/RxJava

Abbildungsverzeichnis

2.1 Anwendung mit Button. 4
2.2 Die Konsolenausgabe für die Button Events. 4

4.1 Schematischer Aufbau eines Observer Patterns. 9

5.1 Die vier relevanten Bestandteile für ein reaktives System.Quelle [Bon14] 12

6.1 Datenfluss Filter-Operation 26
6.2 Datenfluss Map-Operation 27
6.3 Datenfluss Merge-Operation 27
6.4 Datenfluss Zip-Operation 28

Tabellenverzeichnis

6.1 Vergleich zwischen Funktionalität der Iterable- und Observable-Schnittstelle. . . 17
6.2 Ablauf der Threads bei Verwendung von subscribeOn und observeOn 25

Listingverzeichnis

2.1 Implementierung des eigentlichen Event Handlers 2
2.2 Klasse zum Anwendungsstart in welcher auch der EventHander gesetzt wird. . . 3
3.1 Beispiel Erstellung, Verarbeitung und Ergebnisermittlung von Streams. 5
5.1 Lambda Beispiel in Java 14
6.1 Beispiel Observable Initialisierung und Subscription 17
6.2 Beispiel Cold Observable 18
6.3 Beispiel Hot Observable 18
6.4 Beispiel Flowable mit BackPressureBuffer 20
6.5 DefaultSubscriber mit Back Pressure 20
6.6 Beispiel eines Singles 21
6.7 Beispielverwendung eines Subjects 22
6.8 Beispielverwendung eines Processors 23
6.9 Beispiele für Scheduling mit zwei subscribeOn-Aufrufen 24
6.10 Beispiele für Scheduling mit zwei observeOn-Aufrufen 25
6.11 Beispiel Filter Operation 26
6.12 Beispiel Map Operation 26
6.13 Beispiel Merge Operation 27
6.14 Beispiel Zip Operation 28