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

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 ist Reactive Programming

Ist man neu in der Domäne von reaktiver Entwicklung stellt man schnell fest, dass allerhand mögliche Definitionen und Beschreibungen findet was Reactive Programming denn zu sein scheint. Bevor jedoch hier eine Definition erläutert wird, muss zwischen unterschiedlichen Begrifflichkeiten differenziert werden: *Reactive Systems, Functional Reactive Programming* und natürlich *Reactive Programming*. Vorab wird jedoch ergründet worum es sich grundlegend handelt und wieso es zur heutigen Zeit von Relevanz ist, über die möglichen Verwendung von Reactive Programming zu sprechen.

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

Wie die Wortherkunft verlauten lässt, wird etwas als reaktiv bezeichnet, wenn eine Reaktion durch eine vorangegangene Aktion ausgelöst wird. Bei diesen Aktionen handelt es sich meist um Veränderungen an verwendeten Daten und 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 klassischer imperativer sowie sequentieller Ablauf den 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 sich das dadurch wider, das mögliche Reaktionen, meist als Methoden oder Funktionen realisiert, an die mögliche eintreffenden

Ereignisse gebunden werden. Dies geschieht über so genannte Event-Handler beziehungsweise Event-Listener oder über die Verwendung von Callbacks.

```
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. Das Interface liefert eine Funktion welche überschrieben werden muss. Innerhalb dieser Methode wird die Reaktion auf das aufgetretene Event definiert, in diesem Beispiel soll in Konsole die Event-Art ausgegeben werden. In Listing 2.2 wird nun dem einzigen Element der GUI, dem Button, ein Event Handler hinzugefügt. Die Parameter besagen, dass jedes auftretenden Mouse-Event den Handler auslöst. Somit wird der Handler ein ein auftretendes Ereignis gebunden, und dir Programmablauf findet nicht mehr sequentiell statt, sondern das Framework der Oberfläche erkennt das Event und der passende Handler wird aufgerufen. Ein einem Objekt können natürliche mehrere Handler gebunden werden, die auf das gleiche oder unterschiedliche Event reagieren müssen.

```
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();
 }
}
```

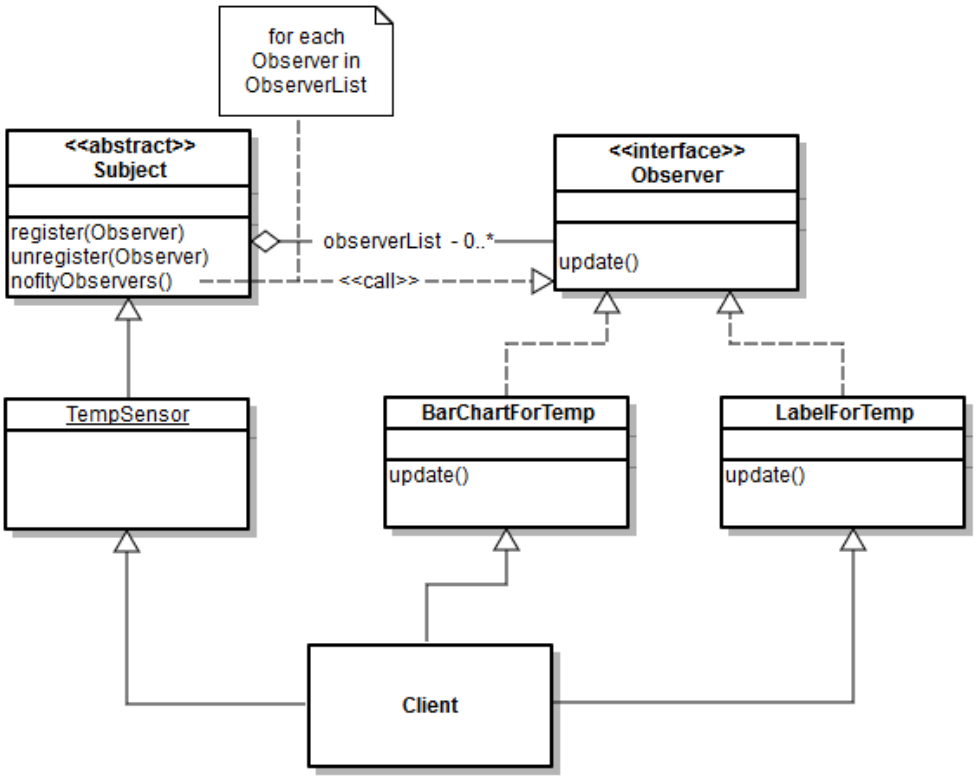

Abbildung 2.1: Schematischer Aufbau eines Observer Patterns.

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

Ein bewährtes und klassisches Vorgehen bei Anwendungsanforderungen dieser Art ist das schon 1994 von Erich Gamma und seinen Mitstreitern beschriebenes Entwurfsmuster, dem *Observer-Pattern* [GHJV11]. Auch hier lässt sich wieder die GUI als guten Beispiel heran ziehen. Man betrachtet eine Oberfläche welche die Temperatur im Raum anzeigt, diese Oberfäche repräsentiert in Abbildung 2.1 das Subject. Die Temperatur wird als Zahl in Grad Celsius sowie graphisch in einem Balken dargestellt. Obwohl beide Elemente auf die selben Daten zugreifen, stehen programmatisch die Objekte in keine Zusammenhang. Um nun das Beobachtermuster zu implementieren, muss der sich ändernde Wert, der zum Beispiel von einem Temperatursensor gemessen wird, beobachtet werden. Deshalb werden die beiden Objekte, die Ausgabe per Zahlenwert und der graphische Balken, als Observer, in der Abbildung als ConcreteObserverA und ConcreteObserverB registriert. Dies heißt wiederum, dass die beiden Observer die *register()*-Methode des Subjects aufrufen und somit in die Observer-Liste dieses Subjects aufgenommen werden. Ändert sich nun zum Beispiel die Temperatur wird die *notifyObservers()*-Methode aufgerufen, was heißt, dass über die Liste der 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 ge-

änderten Parameters direkt mitgeteilt. Das Pull-Modell verfolgt den Ansatz, dass der Observer eine Information erhält, dass sich Werte geändert haben, muss jedoch über die Getter-Methoden die für ihn relevanten Wert 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 wissen muss. Jedoch sinkt dadurch die Flexibilität, da Observer Interfaces exakter beschrieben werden müssen, damit das Subject weiß, welche Information 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.

2.1.1 Differenzierung zwischen Reactive Proramming und Reactive Systems

Das zuvor angesprochene Verhalten bezieht sich nur auf ein Beispiel. Wie jedoch wirkt es sich aus, wenn eine komplette Anwendung unter Verwendung dieses Stils entwickelt wird? 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 Anwendungen eher auf das Reactive Manifesto [Bon14]. Laut Jonas Bonér und den vielen Unterstützern sind vier Bestandteile essentiell, damit eine Anwendung die Anforderung erfüllt um sich reaktiv zu verhalten. Die Ansicht des Manifest stützt sich auf einen Architektur- beziehungsweise 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 2.2 aufgezeigt werden. Wie in diesem beschrieben sind Reaktive Systeme:

- **Antwortbereit (engl. responsive):**
  Ein System muss immer zeitgerecht antworten. Die Antwortbereitschaft ist die Grundlage 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. Die wird durch Replikation der Funktionalität, der Isolation von Komponenten sowie dem Delegieren von Verantwortung 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 sinken,

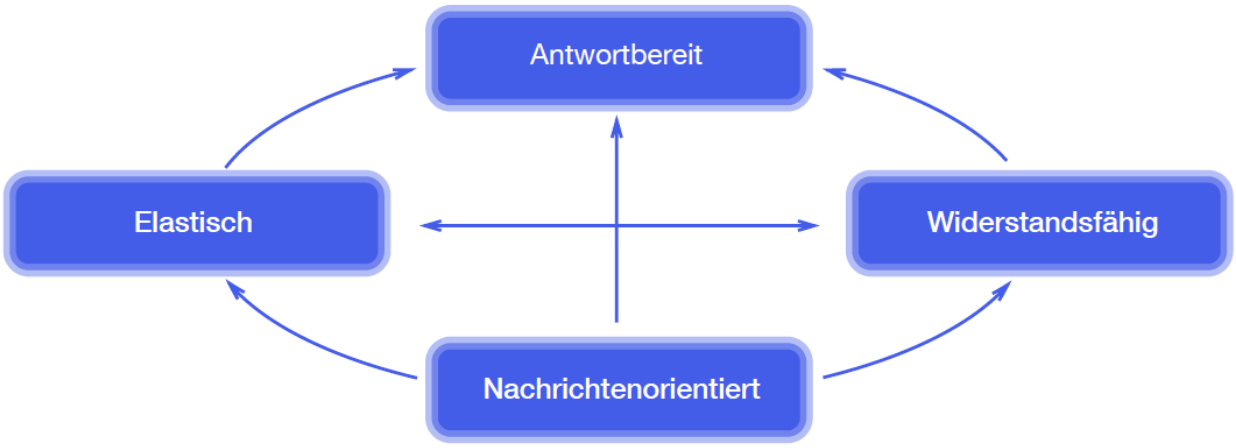

Abbildung 2.2: Die vier relevanten Bestandteile für ein reaktives System.Quelle [Bon14]

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 entsteht.

Das Manifest erwähnt jedoch in keinster Weise den Zusammenhang von Systemen zum Reactive Programming. Auch 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
- 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 Nutzeroberfläche) oft aus der Reaktion auf Ereignisse unterschiedlicher Art die direkt in der Komponente verarbeitet werden sollen. Somit herrscht hier ein ereignisgetriebenes Verhalten.

2.2 Reactive Programming vs. Functional Reactive Programming

Auf der Suche nach einer Definition zu Reactive Programming stößt man oft auf es 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 relevanten Faktor Zeit erweitert kann ein reaktives Verhalten beschrieben 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. Somit kann das Functional Reactive Programming als Teilmenge des Paradigma der reaktiven Programmierung angesehen werden. 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 2.3: Lambda Beispiel in Java

In 2.3 wird jedes der Listenelemente wird auf die Konsolenausgabe mit konkateniertem String abgebildet. Dadurch wird ein ähnliches Verhalten der funktionalen Programmierung reproduziert. Somit kann innerhalb von Java reaktiv unter Zuhilfenahme von funktionalen Methodiken entwickelt werden. Der Unterschied findet sich also genau an der Stelle, dass funktionale Eigenschaften zwar in Java vorhanden sind, dadurch Java allerdings nicht als funktionale Programmiersprache verstanden werden kann, denn auch innerhalb der Objektorientierung kann auf dem imperativen Weg ein reaktives Verhalten entwickelt werden. Faktor Zeit wird im reaktiven Programmieren eher darin reflektiert, dass konkrete Werte im Laufe der Zeit gesendet werden. Reactive Programming kann somit als eine Abstraktion gesehen werden, die sich über die Vorhandenen Paradigmen erstreckt und je nach Situation und Anwendungsbereich auf andere Programmierstile zurück greift.

2.3 Wie wird Reactive Programming realisiert?

Es wurde berichtet was allgemein unter einem reaktiven Verhalten verstanden wird. Ebenso wurde die Begrifflichkeiten und Zusammenhängen zwischen Reactive Programming, Reactive Systems und Functional Reactive Programming erläutert. Es stellt sich jedoch noch die Frage wieso aktuell die Reaktivität von Applikation stark gefordert wird und wie eine Umsetzung davon aussieht. Durch große Datenraten, Datenmengen und vielen vernetzten 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. Somit lohnt es sich, denn Ablauf in einen sequentiellen und einen parallelen Teil zu zerlegen. Die Blockade die somit noch bleibt, sind die Programmteile die sequentiell abgearbeitet werden müssen und nicht weiter optmiert werden können. Reactive Programming versucht nun eine Lösung abzubilden die den Ablauf innerhalb des parallelen Teils vereinfacht und beschleunigt um eine möglichst geringe Latenz und große Flexibilität und Reaktionsfreudigkeit eines Programms widerzuspiegeln. Ein Ansatz diese Lösung zu realisieren ist die Verwendung von Streams. Die Stream-Api wurde wie die Lambda-Funktionalität mit Java 8 eingeführt. Sich veränderte oder neue 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.

2.3.1 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 respräsentiert werden. Nahe liegt die verbreitete Analogie der Fließbandverarbeitung. Man hat eine Menge von Objekten die nacheinander gewissen Operationen unterzogen werden. Diese wird einmalig durchgeführt und die Zwischenergebnisse bleiben nicht vorhanden. In Listing 2.4 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 er werden nur Kopien weiter gereicht.

```
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 2.4: Beispiel Erstellung, Verarbeitung und Ergebnisermittlung von Streams.

2.3.2 Bulk Operations

Diese Operationen gelten als funktional in sind seit Java 8 auf Collections sowie Streams anwendbar. Diese Operation 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. Zu beachtet ist, dass jede Operation auf einer Kopie des eigentlichen Objekts ausgeführt wird, also bleibt zum Beispiel die Liste unverändert wenn man solch eine Massenoperation auf die Elemente der Liste ausführt. Als Ergebnis wird eine modifizierte Kopie der ursprünglichen Liste geliefert. Auch bei einer Verkettung wird immer nur eine Kopie des Eingangsobjekts modifiziert und weiter gereicht. Erwähnung sollte noch die unterschiedliche Art von Operation finden. Es wird zwischen drei Arten von Operationen unterschieden: *Erzeugung*, *Berechnung* und *Ergebnisermittlung* die sich wie folgt abbilden

lassen [Ind15] :

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

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. Resultieren erhält man einen Stream. Eine Reihe von Berechnungen kann nun verketten stattfinden, zum Beispiel eine direkte Manipulation oder Filterung nach Kriterien. Sind die Berechnungen abgeschlossen wird das Ergebnis zum Beispiel auf der Konsole ausgegeben oder in einem Datentyp gespeichert. Konkret ist der Ablauf in Listing 2.4 zu sehen. Wie vorhin schon erwähnt wird via Methodenaufruf eine Stream-Repräsentation der Collection geschaffen. Die Berechnung beziehungsweise Bearbeitung finden in der Konkatenation der Operation *map()* und *filter()* statt. Zuerst wird eine Kopie jeden Strings in Großbuchstaben transformiert, anschließend wird gefiltert ob der String ein $\ddot{L}^{\cdot\cdot}$ enthält. 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 noch beliebig viele Stream-Objekte von einer Collection erstellt werden, ebenso kann beliebig oft durch einen Stream durch gegangen werden. Dies ist möglich das nie eine direkte Änderung vorgenommen wird, sondern wie erwähnt immer nur Kopien weiter gegeben werden.

2.3.3 Back Pressure

Man nehme an, zwei Streams stehen in Verbindung zueinander. Die Bearbeitung der einzelnen Ströme finden asynchron und parallel statt, so wie es bei der reaktiven Programmierung beabsichtigt wird. Der eine Stream führt nun eine kurze Überprüfung durch der zweite führt eine Berechnung durch. Nach dieser Berechnung soll eine Verknüpfung mit dem nächsten Objekt des ersten Streams stattfinden, jedoch ist die Überprüfung doppelt so schnell. Dadurch entsteht ein Rückstau innerhalb des ersten Streams. Je nach Implementierung werden alle Elemente auf Seiten des Verarbeitenden gepuffert[1]. Dieses theoretische Beispiel zeigt, dass es bei der Arbeit mit vielen asynchronen und parallelen Operationen als sehr wichtig gilt, jede Interaktion zwischen zweier solcher Programmelemente genau zu kalkulieren. Findet dies nicht statt, kann zum Beispiel ein hoher Ressourcenverbrauch oder ein Programmabsturz durch eine *OutOfMemoryError* die Folge sein. Um dies zu verhindern muss diese Art von Fehler vorbeugend behandelt werden um Schlimmeres zu verhindern. Eine Möglichkeit wäre zum Beispiel das Setzt einer festen Puffergröße und beim Erreichen dieser das Werfen einer *MissingBackpressureException*.

[1]Zum Beispiel bei der Observable-Klasse des RxJava2 Frameworks

2.4 Reactive Streams

Festhalten lässt sich somit, dass die Verarbeitung von asynchronen, parallele Streams die Reaktivität generiert. Nach dem anfangs erwähnten Observer Pattern werden die Streams observiert und die Daten werden den Observern publiziert. In der Welt der JVM wurde die Initiative der *Reactive Streams* geschaffen, um einen Standard für die Verarbeitung von asynchroner Streamverarbeitung mit nicht blockierendem Back Pressure zu etablieren [rsm]. Das zu bewältigenden Problem ist die unterschiedliche Implementierung der bereits existierenden reaktiven Frameworks. Somit 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]. Einige der hier aufgeführten Frameworks werden in folgendem Abschnitt noch genauer beschrieben. 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].

2.5 Überblick über bekannte Frameworks und ihre Eigenschaften

Die Welt der Softwareentwicklung bietet viele unterschiedliche Programmiersprachen für unterschiedliche Anwendungsfelder. Da es den Umfang dieser Arbeit überschreiten würde sich mit allen vorhandenen Frameworks zu den jeweiligen Sprachen zu befassen, wird auch in diesem Abschnitt hauptsächlich die Vielfalt innerhalb des Java Universums behandelt. Insbesondere die Reactor Bibliothek, das Akka Toolkit und die ReactiveX API, speziell RxJava.

2.5.1 ReactiveX

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 erschaffen. 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 Erweiterung. 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. Vorteil von diesen Erweiterung ist die Unabhängigkeit. Somit ist die Bibliothek 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 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, daher folgt eine genau Beschreibung in einem weiteren Kapitel [rxg]. Somit wird auf ein Beispiel in diesem Abschnitt verzichtet.

2.5.2 Akka

Akka bezeichnet sich selbst als Toolkit und nicht als Framework. Es kann wie jede beliebige Bibliothek in ein Java Projekt eingebunden werden. Jedoch ist diese Werkzeugkiste etwas umfangreicher als die Erweiterungen von ReactiveX. Entscheidend ist hier das relevante Modul der *Akka Streams*. Spezielle das von Akka umgesetzt Actor-Modell spielt hier eine Rolle. [2]. Akka versucht mit seinem Vorgehen die Prinzipien des Reaktiven Manifest und auch der Reactive Stream Initiative umzusetzen.

2.5.3 Reactor

Reactor ist der ReactiveX Erweiterung sehr ähnlich. Die API ist bis auf ein paar Gemeinsamkeiten anders benannt, jedoch ist die Funktionalität fast deckungsgleich. Dies kommt daher, dass die Codebasen von beiden Projekten praktisch identisch sind [Kar16]. Beide Projekte arbeiten stark zusammen und beide haben schon von den jeweils anderen Implementierungen übernommen. Reactor existiert jedoch nur in der Java Welt und wird in ersten Sinne von Pivotal entwickelt. Dadurch ergibt sich auch die Nutzung in der neuesten Version des Webentwicklungs-Frameworks Spring 5. Die Reactive Bibliothek wird ab Version 5 direkt mit Spring ausgeliefert und soll eine reaktive Reaktion für REST-API's ermöglichen und vereinfachen.

Framework für JavaFX - RxJavaFX

Einführung und Eigenschaften erläutern

2.6 Testen von reaktivem Code mit dem JUnit Framework

Noch nichts genaues. Muss noch geschaut werden wie die Funktionalität von JUnit RP abdeckt.

[2] Beschreibung actor Modell, am besten mit Code beispiel

Kapitel 3

Einführung in Reactive Programming mit RxJava2

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-Vorgehen verwendet werden (interaktiv) [NC17]. Weitere Eigenschaften sind die Nutzung für asynchrone und synchrone Implementierungen und die Repräsentation von Null bis unendliche[1] viele Werte oder Ereignisse im Laufe der Zeit. Es folgt nun eine kurze Schilderung wie diese 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].

3.1 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 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.

[1] Mathematisch gesehen: $D = [0, \infty)$

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

Tabelle 3.1: Vergleich zwischen Funktionalität der Iterable- und Observable-Schnittstelle. Quelle [mai]

3.2 Parallelisierung und Nebenläufigkeit

Wie bekannt sein dürfte ist die Parallelisierung und Nebenläufigkeit eher auf Systemebene zu betrachten. Als parallel bezeichnet man die Ausführung unterschiedlicher Tasks auf verschiedenen Kernen oder Maschinen. Voraussetzung ist die wirklich gleichzeitige Bearbeitung der Tasks. Von Nebenläufigkeit wird gesprochen wenn eine Recheneinheit mehre Tasks oder Threads verarbeitet und immer nur einer dieser Aufgaben zur einer Zeit bearbeitet wird. Nach einem gewissen Zeitraum bekommt ein anderen Task die Rechenleistung und die vorherige Task wurde beendet wenn die Aufgabe erfüllt wurde oder wartet auf erneute Rechenzeit. Dieses Verfahren wird *time slicing* genannt und wird von Kernel des Betriebssystems verwaltet. Somit ist Parallelisierung immer auch nebenläufig aber Nebenläufigkeit nicht unbedingt auch parallelisiert. Um dieses Verfahren in Verbindung mit den Observables zu bringen ist zu sagen, dass ein Observable Objekt immer serialisiert und thread-safe sein muss. Die Callback-Methoden des Subscribers dürfen also nie zeitgleich aufgerufen werden. Parallelisierung und Nebenläufigkeit werden also dadurch erreicht, dass man Observables miteinander verbindet und jeder der Streams parallel oder nebenläufig mit den jeweils anderen interagieren kann zum Beispiel mit den Bulk-Operations merge und zip, aber dazu später mehr.

3.3 Push und Pull

Wird synchrones Pulling von Objekten einer Liste über das Iterable Interface durchgeführt, so wird im Gegenzug ein asynchrones 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[2] des Verbrauchers gepusht. Wie die Tabelle 3.1 zeigt gilt dies ebenso beim Auftreten eines Fehlers oder beim Erreichen des Endes der Datenquelle. Die Verbindung zwischen Observable und Observer findet über ein Subscription statt. Damit werden die beiden zu einem Paar gebunden

[2]Diese Methode wird durch ein Autftreten eines Ereignisses oder von Daten im Stream aufgerufen. Somit handelt es sich bei dieser Art Methode um Callback Methoden.

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 fordert. Dies wird auch als *faules* Verhalten bezeichnet. Somit wird das Arbeiten durch das Subscriben und nicht durch das Erstellen eines Observables verursacht. 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. Somit ist ein Observable-Objekt beliebig oft verwendbar.

3.4 Observable

Es wurde in den vorherigen Abschnitten schon etwas über das Observable gesagt, doch der Vollständigkeit halber wird hier seine Eigenschaften und Fähigkeiten zusammengefasst beschrieben. Es handelt sich hierbei um die Quintessenz 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 3.1: Beispiel Observable Initialisierung und Subscription

Der Ereignisstrom über einen Zeitraum wird von dem 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 im unendliche Eventstreams handelt kann. Die Tabelle 3.1 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 *onComplete()*-Aufruf oder das Aufkommen eines Fehlers wird kommuniziert mit der *onError(Throwable)*-Methode. Somit gilt:

$$OnNext \cdot (OnComplete|OnError)?$$

Daraus resultiert das entweder ein gewolltes Ende oder ein unvorhergesehener Fehler auftritt, aber niemals beides[3]. Schaut man sich nun Listing 3.1 an, sieht man den Ablauf. Ein Observable

[3]pdf in bib aufnehmen und zum regex binden

wird initialisiert[4], hier mit drei Elementen, also ein endlicher Stream. Durch das *subscribe* wird anonym ein Subscriber beschrieben. 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 seine eigene Kopie die Streams. Dies nennt man auch *Cold Observable*.

```
System.out.println("--------Cold Observable----------------------");
Observable<Long> obs = Observable.interval(1, TimeUnit.SECONDS);
Disposable d1 = obs.map(tick -> tick*2).subscribe(s -> System.out.println(
 s));
Thread.sleep(1000);
Disposable d2 = obs.map(tick -> tick*20).subscribe(s -> System.out.println
 (s));
Thread.sleep(5000);
```

Listing 3.2: Beispiel Cold Observable

Wie hier in Listing 3.2 zu sehen werden zwei Kopien des Observable-Streams erstellt. Es ist zu beachten, dass somit jeder Subscriber einen eigenen Timer bekommt. Will man nun aber die beiden Rechenoperation 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> obs2 = Observable.interval(1, TimeUnit.SECONDS).publish()
 .autoConnect();
obs2.map(tick -> tick*2).subscribe(s -> System.out.println(s));
Thread.sleep(1000);
obs2.map(tick -> tick*20).subscribe(s -> System.out.println(s));
Thread.sleep(5000);
```

Listing 3.3: Beispiel Hot Observable

Der Methodenaufruf *publish()* ist hier der Schlüssel. In Listing 3.3 sieht man, nachdem das Intervall gestartet wird besagten Aufruf. Sollen Daten geteilt werden spricht man von einem *Hot Observable*. Via *publish()* erfährt das Observable-Objekt, dass die Daten geteilt werden sollen. Mit *autoConnect()* wird bestätigt, dass sich jeder neue Subscriber automatisch verbinden kann. Als Parameter kann auch eine Begrenzung für die Subscriber-Anzahl übergeben werden. Grundlegend hat sich am Aufbau des Observerables nichts geändert. Sobald ein Hot Observable eine Subscription erhält beginnt es Daten auszugeben. Schaltet sich nun ein weiterer Subscriber

[4]Es gibt unterschiedliche Methoden je nach Objektart welche als Observable dargestellt werden soll. Eine Liste finden man hier: https://github.com/ReactiveX/RxJava/wiki/Creating-Observables

auf den Stream beginnt er mit den Werten die auch der oder die anderen Subscriber erhalten. Somit wird der neue Subscriber die schon vergangenen Events nicht mehr erfahren können. Es macht zum Beispiel Sinn, wenn eine Mausbewegung getrackt wird, da neue Subscriber nicht wissen müssen wo sich der Mauszeiger vorher befand, sondern 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 geht kein Weg an Cold Observables vorbei. Handelt es sich um einen Stream über den man keine konkrete Kontrolle hat, Paradebeispiel I/O, wird man meist auf Hot Observables zurück greifen.
Es gibt eine Unzahl an Methoden und vor allen Dingen Bulk Operations die auf einem Observable Stream ausgeführt werden können. Die komplette Übersicht findet man in der Javadoc [rxa]. Ein Punkt welcher das Observable nicht abdeckt ist das Back Pressure. Es sind keine Möglichkeiten erhalten die auf Back Pressure reagieren können. Jedoch ist nicht blockierender Back Pressure ja notwendig um mit der Reactive Streams Initative konform zu gehen. Daher wurde die Klasse *Flowable* implementiert.

3.5 Flowable

Diese Klasse ist nach den Kriterien der Reactive Streams realisiert. Flowable bietet weitestgehend alles, was auch Observable zu bieten hat, bringt aber durch die Implementierung *"implements org.reactivestreams.Publisher<T>"* des Publisher Interfaces eine hervorragende Verträglichkeit zu den Reactive Streams mit sich. Ebenso besteht die Möglichkeit Back Pressure zu handhaben, indem Puffergrößen[5] eingestellt werden können[6]. In der offiziellen Dokumentation von RxJava2 wird beschrieben wann es sich besser eignet auf Observable oder aber auf Flowable zu setzt [rxd]. Wann sollte man Observable 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 Flowable 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.

[5] Flowable.observeOn hat standardmäßig Platz für 128 Elemente im Puffer
[6] Beispiel für Buffer settings von flowable

- 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.

Betrachtet man das Listing 3.4 sieht man im Vergleich zum Observable den Aufruf der *onBackpressureBuffer()*-Methode. Hiermit kann direkt eine Puffergröße für jede Subscription festgelegt werden.

```
public static void main(String[] args)
{
 Flowable<String> flowing = Flowable.just("Java", "Kotlin", "Scala");
 flowing.onBackpressureBuffer(1).subscribe(s -> {System.out.println(s);},
   t -> {t.printStackTrace();},
   () -> {System.out.println("finish");} );
}
```

Listing 3.4: Beispiel Flowable mit Back Pressure

3.6 Single

Wo Observable oder Flow eine Menge von Null bis endliche viele Elemente ausgeben, steht Singe, wie der Name schon sagt, dafür, dass immer nur ein Wert ausgegeben wird. 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 3.5: Beispiel eines Singles

In Listing 3.5 sieht man die Verwendung eines Single-Objekts. Es kann nur mit einem Objekt erstellt werden, di 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.

3.7 Subject

Also Subject versteht man ein Objekt, dass sowohl als Observable und auch als Observer fungieren kann. Es kann an mehreren Observables 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 3.6: Beispielverwendung eines Subjects

Um es das Vorgehen bei Verwendung eines Subject zu veranschaulichen betracheten wir Listing 3.6. Wir erstellen wieder das schon bekannte Observable-Objekt und ein Subject, hier ein speziell ein PublicSubject. Dieses Subject bekommt eine Subscription in welcher alle String-Werte in Großbuchstaben umgewandelt und anschließen auf der Konsole geschrieben werden. Wenn nun das Subject selbst an dem vorhanden Observable subscribed, werden die von diesem ausgegebenen Werte an gemapped und an die Subscription 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 weiteren 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.

Dieses Back Pressure Problem kann wie schon beim Observable nicht direkt gelöst werden. Somit ist auch hier noch keine Konformität zur Reactive Streams Initiative zu sehen. Dafür gibt es eine weiter Klasse: die Processor-Klasse.

3.8 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 3.7: Beispielverwendung eines Processors

In Listing 3.7 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 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.

3.9 Schedulers

Im Abschnitt zu Parallelisierung und Nebenläufigkeit wurde schon beschrieben, dass jedes Observable-Objekt serialisierbar und thread-safe gestaltet wird. Auch Threads kamen zur Sprache und wie eine Parallelisierung oder Nebenläufigkeit erozeugt werden kann. Da jeder der Bulk Operations auf einem Observable immer wieder ein neues, eigenständiges Observable weiterreicht, besteht die Möglichkeit Verarbeitungsschritte auf anderen Threads als auf dem des Subscribers auszuführen. Dafür bringt das Observable zwei Operationen mit, *subscribeOn* und *observeOn*. Wenn also ein Observable mit *subscribeOn* auf eine Thread Zugang erhält, werden alle weiteren Berechnungen auch auf diesem Thread ausgeführt. Soll nun die Ausgabe der berechneten Werte an den Subscriber übergeben werden, muss dies natürlich auf dem Thread stattfinden, auf welchem der Subscriber läuft. Um auf diesen Thread zurückzukehren verwendet man *observeOn*. Das Paradebeispiel stellt hier wieder die GUI. In JavaFx gibt es einen Scheduler[7] der für alle Änderungen auf der Oberfläche verantwortlich ist. Findet nun eine Berechnung auf dem *Schedulers.Computation()*-Threadpool statt, wird diese Veränderung in der GUI nur sichbar, wenn mit *observerOn(JavaFxScheduler.platform())* auf den Thread der Benutzeroberfläche zurück gekehrt wird. Das selbe gilt auch bei Verwendung des RxJava Frameworks unter Android, da hier auch immer der Main-Thread der Applikation, auch UI-Thread genannt[8], für die Darstellung zuständig ist.

3.10 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. Zur Veranschaulichung werden sogenannte Marble-Diagramme verwendet. Die oberer Linie stellt immer ein Observable-Stream dar in zeitlichem Ablauf, also ähnlich einem Zeitstrahl. Jedes Element, dargestellt durch die farblichen Marbles, wird durch die mittlere Operation geführt und auf das ausgehenden Observable, immer am unteren Ende des Bildes ausgegeben.

3.10.1 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

[7] JavaFx Scheduler linken

[8] Wenn unter Android eine App gestartet wird, wird für jede App ein Main-Thread erzeugt, der für alle I/O Aktion zuständig ist, sofern die App aktiv ist.

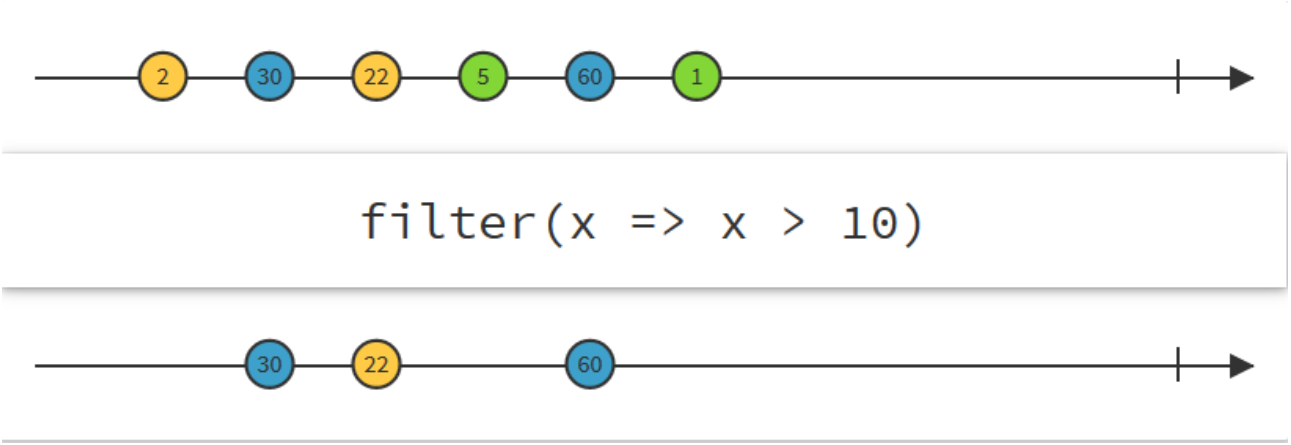

Abbildung 3.1: Datenfluss Filter-Operation

Filter-Operation auszuführen. Im Bild 3.1 sieht man eine Filterung nach dem Kriterium, dass nur jeden Element, das eine Wert größer 10 inne hält, 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 wie in Listing 3.8. Innerhalb des Predicate wird die *test()*-Methode überschrieben 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 3.8: Beispiel Filter Operation

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

3.10.2 Operation map()

Die Map-Operation gilt als Transformationsoperation. Ihre Aufgabe ist es, jeden auftretenden Wert auf eine Funktion abzubilden und das Resultat auszugeben. In Abbildung 3.2 sieht man auf dem ursprünglichen Observable drei auftretenden Elemente. Jedes dieser Elemente wird mit 10 multipliziert und danach auf das resultierenden Observable ausgegeben.

```
Observable< Integer > obs1 = Observable.just( 1, 2, 3 );
```

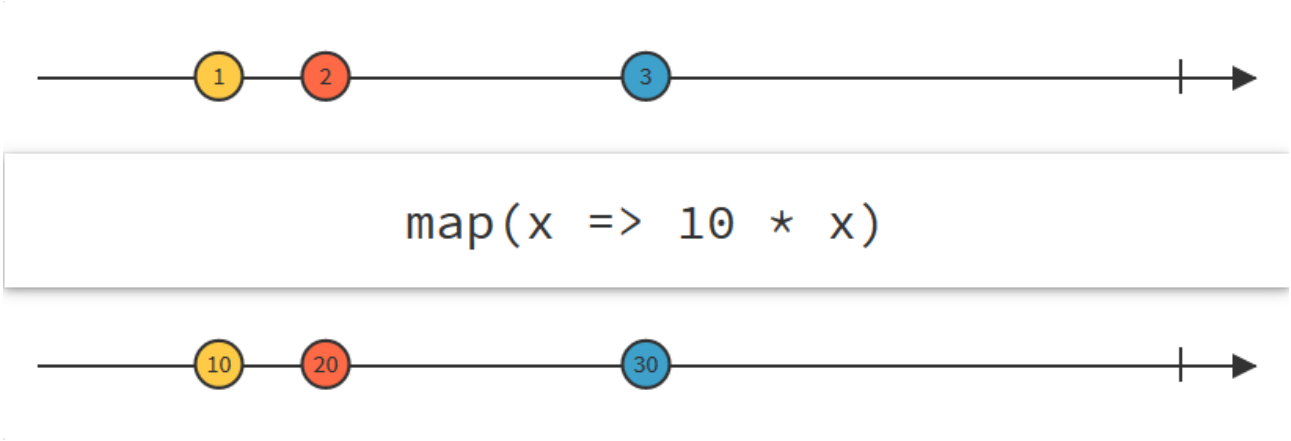

Abbildung 3.2: Datenfluss Map-Operation

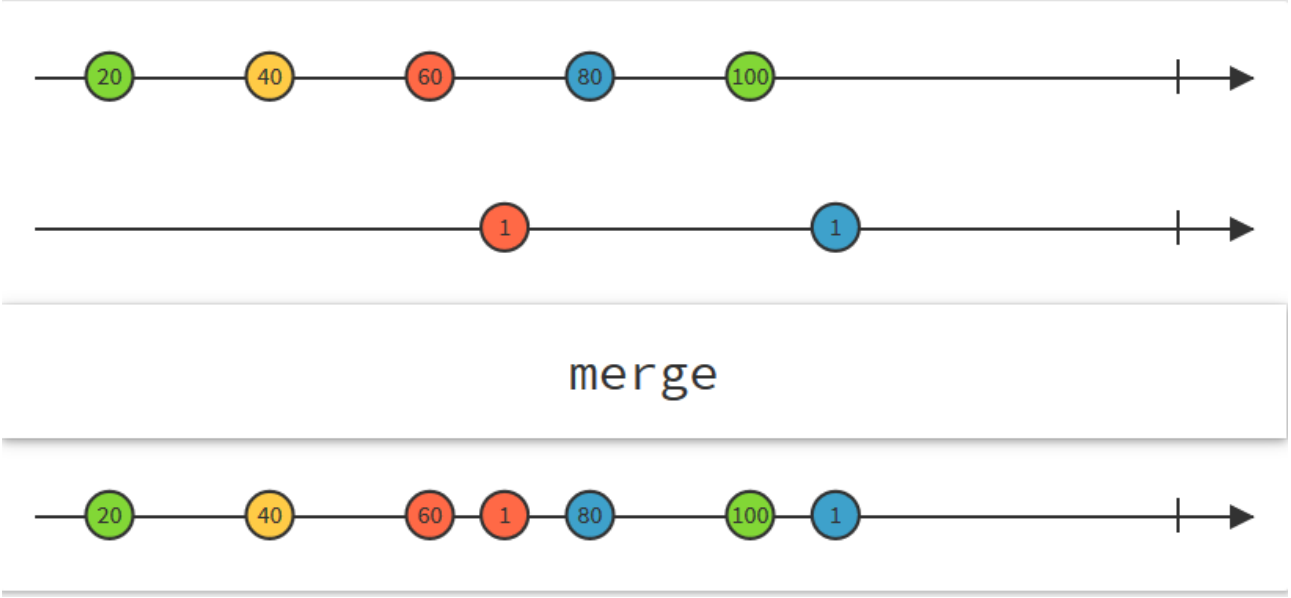

Abbildung 3.3: Datenfluss Merge-Operation

```
obs1.map( x -> x * 10 ).subscribe( x -> System.out.println( x ) );
```

Listing 3.9: Beispiel Map Operation

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

3.10.3 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 3.3 wird deutlich wie die Komposition stattfindet. Zu jedem Zeitpunkt eines der Observables ein Element ausgibt wird es an das zusammengeführte Observable weitergegeben. Somit wird die

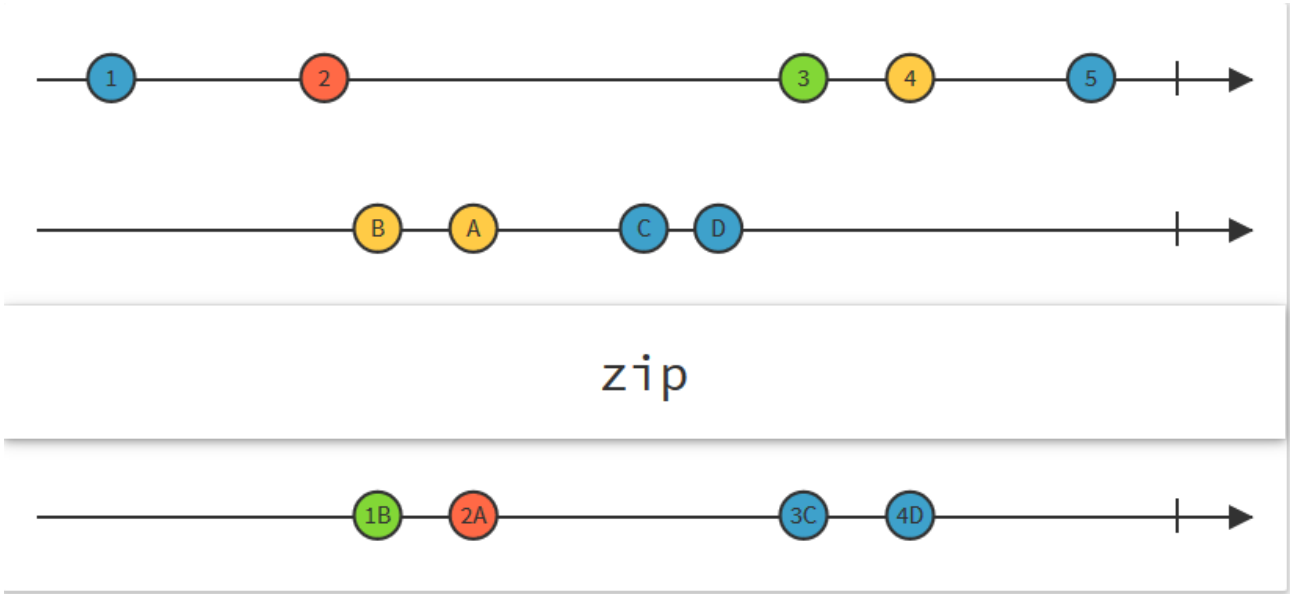

Abbildung 3.4: Datenfluss Zip-Operation

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

Listing 3.10: Beispiel Merge Operation

Im Listing 3.10 ist die entsprechende Implementierung zu erkennen. 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. Dadurch dass das Intervall jede Nanosekunde Werte ausgibt und die Subscription erst in dem Moment endet, in welchem alle Observables abgeschlossen sind, handelt es sich in diesem Beispiel um ein unendliches Observable. Wie vorab schon erwähnt ist die Unendlichkeit immer mit Vorsicht zu genießen hinsichtlich der Verwertbarkeit des Observerables und der Performanz einer Anwendung.

3.10.4 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 3.4 erläutert diese Weise genauer. Die Elemente der beiden Observables sind unterschiedlichen Typs und werden zu unterschiedlichen Zeitpunkten ausgegeben. 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 3.11: Beispiel Zip Operation

Um es sich wieder von Entwicklungsseite zu veranschaulichen ist in Listing 3.11 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, welche Objekte gezippt werden sollen, ebenso muss eine Funktion beschrieben werden, die definiert wie die Elemente miteinander verknüpft werden sollen. Hier ist wieder die Verwendung von Lambdas möglich, wie es auch im Listing stattfindet, oder es muss eine explizite Funktion definiert werden, als Objekt oder anonym. Auch hier ist wieder das Verhalten zu beachten, das erst eine Unsubscription stattfindet wenn alle verknüpften Streams beendet wurden.

Kapitel 4

Beispiel: Implementierung eines Systemmonitors

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

4.1 Klassenbeschreibungen SServerSSeite

4.1.1 API für Systemwerte

Framework für die Systemwerte kurz erläutern.

4.2 Klassenbeschreibungen "ClientSSeite

4.2.1 RxJavaFx

Kurzbeschreibung mit Verweis auf gitbook

4.3 Hier eventuell Testing noch beschreiben

Kapitel 5

Evaluierung

Hier Eval schreiben

5.1 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

[Kar16] Karnok, Dávid: *Operator-fusion (Part 1)*. https://akarnokd.blogspot.de/2016/03/operator-fusion-part-1.html. Version: 2016 (Advanced Reactive Java)

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

[mai] *ReactiveX - An API for asynchronous programming with observable streams*. http://reactivex.io/intro

[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/

[rxa] *RxJava2: Api Doc*. http://reactivex.io/RxJava/2.x/javadoc/

[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

Tabellenverzeichnis

Listingverzeichnis