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* 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*). 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*[1], also das Umkehren der Kontrolle, eine große Rolle. Diese besagt, dass nicht der geschrieben Code den Ablauf beschreibt, sondern die Kontrolle bei dem Framework[2], welches für die Interaktion zuständig ist, liegt, und dieses entscheidet wiederum wie und wann auf ein Event reagiert wird.

[1] Vgl. [Fow05], https://martinfowler.com/bliki/InversionOfControl.html.

[2] In diesem Fall ein Framework für das Realisieren einer GUI. Vgl. [wik].

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[3]. Ein bewährtes und klassisches Vorgehen bei Anwendungsanforderungen dieser Art ist das schon 1994 von Erich Gamma und seinen Mitstreitern beschriebenes Entwurfsmuster[4], dem *Observer-Pattern*[5]. Man betrachtet hier grundlegend zwei Ansätze: *push* und *pull*. [6]. Auch hier lässt sich wieder die GUI als guten Beispiel heran ziehen. Man betrachtet eine Oberfläche welche die Temperatur im Raum anzeigt. 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 regelmäßig aktualisiert beobachtet werden. Das Objekt, welches den Temperaturwert inne hält wird somit als beobachtbar deklariert, die beiden Objekte, die Ausgabe per Zahlenwert und der graphische Balken, werden bei diesem Objekt als Beobachter registriert. Wenn man den push-Ansatz verfolgt, wird jedem Beobachter benachrichtigt und das Objekt mit den geänderten Daten steht dem Beobachter zur Verfügung. Verfolgt man den pull-Ansatz, werden die Beobachter nur kurz darüber informiert, dass sich Werte innerhalb des zu beobachtenden Objekts geändert haben, müssen jedoch aktiv anfragen um die geänderten Werte zu erhalten. Je nach Vorgehen ergeben sich Vor- und Nachteile. Der push-Ansatz steht eher für lose Kopplung, da der Beobachter keine Details zu dem zu beobachteten Objekt braucht. Jedoch sinkt dadurch die Flexibilität, da Beobachterschnittstellen exakter beschrieben werden müssen damit der Beobachtete 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 Beobachteten verlassen muss.
Nach diesem Abschnitt sollte soweit klar sein, wie man ein reaktives Verhalten mit den schon bekannten Bordmitteln realisiert.

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

[3]TODO Beispiel Event-Handler -> ActionEvent, Click on Button; Quellen angeben und Beschreibung zu Beiden Codebeispielen

[4][GHJV11]: Dieses Buch beschreibt viele noch heute verwendete Entwurfsmuster. Die Autoren sind in die Geschichte als die Gang Of Four (*GoF*) eingegangen.

[5]Hier Beispiel vom OP sowie Übersetzung Beobachtermuster; Observer Pattern und Erklärung

[6]Hier gute Beispiele für push und pull Variante

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[7]. Zur heutigen Zeit jedoch bezieht man sich bei Richtlinien zur Gestaltung reaktiver Anwendungen eher auf das Reactive Manifesto[8]. 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 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, 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 reaktiven Programmieren. Auch aus diesem Grund hat Jonas Bonér einen weiteren Artikel[9] verfasst, der sich dieser Thematik annimmt. Die grundlegenden Unterschiede wurde in dem Artikel wie folgt zusammengefasst.

- Reactive Programming ist eine Teilmenge von Reaktiven Systemen auf Implementierungsebene

[7] [HP85]
[8] [Bon] -> ebenso Bild von Manifest einfügen. Deutsches Manifest in Quellen verlinken
[9] [Bon14]

- 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 reaktives Programmieren 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 genau Unterschied zwischen reaktivem Programmieren und reaktiven System 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 eine Komponente, kann diese unter Zuhilfenahme von reaktivem Programmieren 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 Ereignis-getriebenes 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[10]. 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. Dadurch

[10]Vgl.: [Nur17], Seite 2.

wird programmatisch nicht die Frage nach dem *Wie* gelöst, sondern ein Vorgang wird mit mathematischen Funktionen beschrieben[11]. 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*[12]. Somit kann das FRP 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 OO Programmiersprache[13] Java. 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. 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[14]. 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 Datenströmen. Die Stream-Api wurde wie die Lambda-Funktionalität mit Java 8 eingeführt. Sich veränderte oder neue Daten werden

[11] Verweis auf Lambda-Kalkül

[12] [frp]

[13] Noch Verweis suchen: Viele weitere bekannte Programmiersprache wie C# z.B. unterstützen Lambdas ebenso

[14] Vgl. [Amd07]. Es handelt sich um einen Reprint in der IEEE SSCS News. Auf diesem von Amdahl verfassten Original resultiert das gleichnamige Gesetz.

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 Datenströme- Streams

Das Konzept der Streams stellt eine Abstraktion für Folgen von Bearbeitungsschritten auf Daten dar[15]. Streams erinnern an Collections sind jedoch nur einmal traversierbar und nehmen keine direkten Speicherung der Daten vor. Collections können auch als Datenströme 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.

2.3.2 Massenverarbeitung - 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[16]. Somit kann durch eine Operation zum Beispiel eine Veränderung an jedem Objekt einer Liste ausgeführt werden, oder eben auf jedem Objekt welches einen Datenstrom 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 List 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. Dies gilt äquivalent auch für Streams. Erwähnung sollte noch die unterschiedliche Art von Operation finden. Es wird zwischen drei Arten von Operationen unterschieden: *Erzeugung*, *Berechnung* und *Ergebnisermittlung*[17] die sich wie folgt abbilden lassen:

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

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. Resultieren erhält man einen Datenstrom. 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.

[15] Vgl. [Ind15], Seite 42.
[16] Beispiel Bulk Operation
[17] Vgl.: [Ind15], Seite 42f.

2.3.3 Überdruck - Back pressure

Man nehme an, zwei Datenströme 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 Datenstrom führt nun eine kurze Überprüfung durch der zweite führt eine Berechnung durch. Nach der 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 Datenstroms. 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 die Folge sein.

2.4 Reaktive Datenströme - Reactive Streams

Festhalten lässt sich somit, dass die Verarbeitung von asynchronen, parallelen Datenströmen die Reaktivität generiert. Nach dem anfangs erwähnten Beobachtermuster werden die Ströme zum Beobachteten und die Daten werden den Beobachtern publiziert. In der Java Welt wurde die Initiative der *Reactive Streams*[18] geschaffen, um einen Standard für die Verarbeitung von asynchroner Datenstromverarbeitung mit nicht blockierendem Überdruck zu etablieren. 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[19]. Mit Java 9 wird in der sogenannten *Flow API*[20] eine Implementierung nach den *Reactive Streams*-Kriterien geliefert. Ein Beispiel wird von der Java-Community bereits zur Verfügung gestellt[21].

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.

[18] [rea]
[19] [rs.]
[20] [fl.]
[21] [flo]

Kurze Erläuterung zu der Entstehung von Reactive Extensions

2.5.1 Reactivex.io

Rx Frameworks zu den jeweiligen Sprachen. Frameworks wie z.B. Akka[22].

2.5.2 Übersicht spezielle für die Entwicklung mit Java

RxJava. Reactive Streams Konvention. Java 9 Api Änderung bzgl. Reactive Streams.

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.

[22] [Kar16]

Kapitel 3

Einführung in Reactive Programming mit RxJava

Beschreibung wieso RxJava. Beschreibung was beschrieben wird.

3.1 Wie funktioniert Reactive Programming

Einleitung zum Aufbau: Klassenübersicht des Frameworks mit Erklärung.

3.1.1 Syncronität

Sync vs. Async - was bringt RP in dieser Hinsicht

3.1.2 Parallelisierung

Concurrency vs. Parallelism - was tritt wie wann auf bzw. kann wie wann angewandt werden

3.1.3 Push vs. Pull

Wichtigster Unteschied. Observable als Gegenpart zu Interable - somit Push vs. Pull Vergleich.

3.2 Rx.Observable

Interface Übersicht. Nutzen und Anwendung anhand von Beispiel. Hot vs. Cold

3.3 Rx.Observer

Was kann Observer -> Interface Übersicht

3.3.1 Rx.Subscriber

Was ist speziell am Subscriber -> Interface Übersicht

3.4 Operationen und Transformationen

Erläuterung von den Stadien der Operation von Beginn über Mitte bis Ende.

3.4.1 Exkursion: Streams API Java 8

Beschreibung was Streams darstellen, wie sich Observables im Vergleich verhalten

3.4.2 Operation filter()

Beispiel und Perlenbild. Einsatz beschreiben

3.4.3 Transformation map()

Beispiel und Perlenbild. Einsatz beschreiben

3.4.4 Transformation flatMap()

Beispiel und Perlenbild. Einsatz beschreiben

3.4.5 Operation merge()

Beispiel und Perlenbild. Einsatz beschreiben

3.4.6 Operation zip()

Beispiel und Perlenbild. Einsatz beschreiben Eventuell noch mehr Operationen

Kapitel 4

Beispiel: Implementierung eines Systemmonitors

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

4.1 API für Systemwerte

Framework für die Systemwerte kurz erläutern. Grobes Vorgehen beschreiben wie man es in etwa Umsetzen kann.

4.1.1 Beschreibung Klasse 1

4.1.2 Beschreibung Klasse 2

4.1.3 Beschreibung Klasse 3

4.1.4 Beschreibung Klasse 4

4.2 Client für API: GUI zur Repräsentation der Systemwerte

4.2.1 Beschreibung Klasse 1

4.2.2 Beschreibung Klasse 2

Literaturverzeichnis

[Amd07] 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, Klang V. Jonas: Reactive Programming versus Reactive Systems. https://info.lightbend.com/reactive-programming-versus-reactive-systems.html

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

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

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

[Fow05] Fowler, Martin: *InversionOfControl.* https://martinfowler.com/bliki/InversionOfControl.html. Version: 2005

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

[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] In: Harel, D. ; Pnueli, A.: *On the Development of Reactive Systems.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1985. – ISBN 978–3–642–82453–1, 477–498

[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)

[Nur17] Nurkiewicz, Christensen B. Tomasz: *Reactive programming with RxJava: Creating asynchronous, event-based applications.* Sebastopol, CA : O'Reilly, 2017. – ISBN 9781491931653

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

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

[wik] *List von GUI-Bibliotheken.* https://de.wikipedia.org/wiki/Liste_von_GUI-Bibliotheken

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis