Vergleich einer reaktiven und imperativen Implementierung eines Demoprojektes zum Filtern von Live-Daten

Wilhelm Büchner Hochschule

Alexander Pann

850621

25. April 2021

Inhaltsverzeichnis

1 Einführung

In dieser Arbeit wird ein Demoprojekt mit dem reaktiven Framework RxJS [24][26] implementiert. Das Ziel ist es, dass ein Benutzer Tweets in Echtzeit nach Schlüsselwörter filtern kann. Es soll gezeigt werden, wie Probleme der nebenläufigen und parallelen Programmierung wie Verklemmungen, Verhungern und Livelocks mit reaktiver Programmierung elegant gelöst werden können oder zumindest die Anzahl der Probleme reduziert werden können. Die Datenflussorientierte Herangehensweise an das Problem soll anhand des Demoprojektes ebenfalls dargestellt werden. Die fertige Anwendung kann in der Grafik 1 gesehen werden.

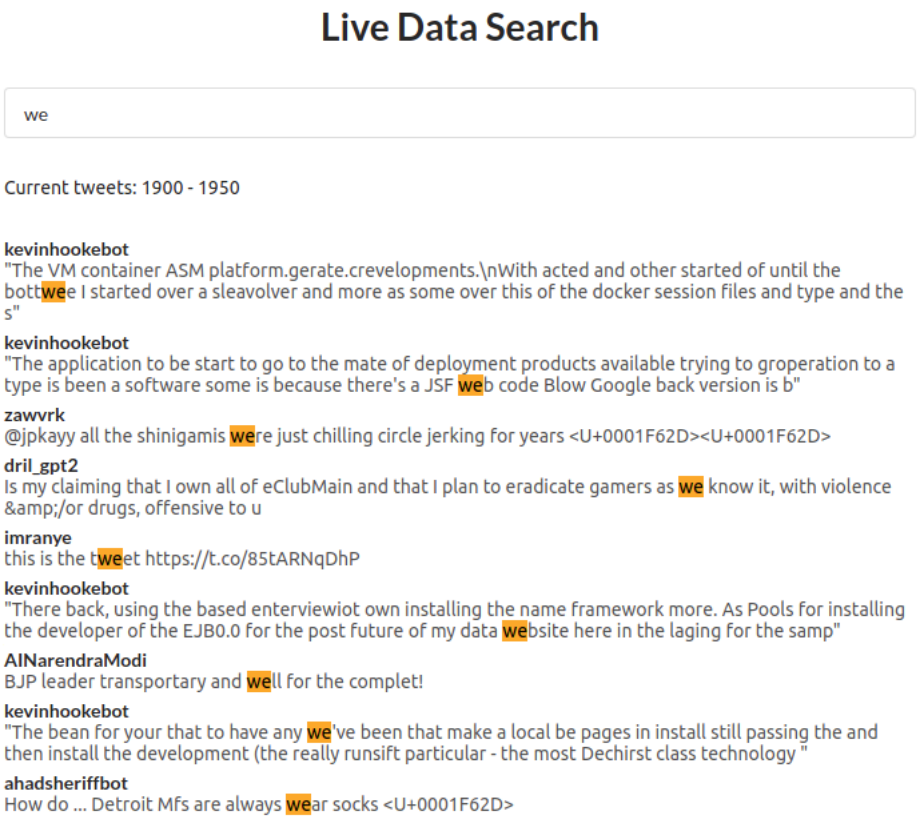


Abbildung 1: Demoprojekt

Es werden alle 2 Sekunden 50 neue Tweets aus einer Testdatei mit ca. 2500 Tweets gelesen und in einem Datenstrom veröffentlicht. Der Benutzer kann im Suchfeld ein oder mehrere Begriffe eingeben. Alle aktuell geposteten Tweets (50 Stück) werden nach den Begriffen durchsucht, die gefundenen Tweets angezeigt und die Suchbegriffe in Orange markiert. Die Testdatei wurde dabei bereits im Vorfeld eingelesen. Auf eine Variante, bei der die Datei live auf Änderungen durchsucht wird, wurde aus technischen Gründen verzichtet, um den Umfang des Projektes gering zu halten.

2 Reaktive Programmierung

Viele Softwaresysteme sind reaktiv: Sie reagieren auf auftretende Ereignisse und führen eine Berechnung aus. Es gibt verschiedenste Ereignisarten wie zum Beispiel Ereignisse der grafischen Benutzeroberfläche, die auf Benutzereingaben reagieren sowie eingebetteten Systemen, die auf Signale reagieren. Reaktiver Code wird asynchron durch Ereignisse ausgelöst. Die Ausführung ist dabei nicht blockierend, d. h. ausführende konkurrierende Threads müssen nicht auf gemeinsame Ressourcen durch Blocken warten. Er ist daher schwer zum Nachvollziehen und der Kontrollfluss des gesamten Systems ist nur schwer zu verstehen [23].

Die reaktive Programmierung versucht das Problem, reaktive Software zu entwerfen, implementieren und zu warten, zu vereinfachen. Es ist ein deklaratives Programmierparadigma, dass sich mit Datenströmen und deren Ausbreitung von Änderungen beschäftigt [35]. Statische oder dynamische Daten können dabei als eine Abfolge von Werte ausgedrückt werden, die sich mit der Zeit ändern (auch manchmal als Signale oder Verhalten bezeichnet). Abhängigkeiten werden auch automatisch verfolgt, sowie Ereignisströme benutzt, mit denen diskrete Aktualisierungen modelliert werden können.

Reaktive Programme werden oft durch das Beobachter-Muster [12] umgesetzt. Wie in Abbildung 2 abgebildet, gibt es zwei Schnittstellen, die implementiert werden müssen: Ein Subjekt, welches beobachtet wird sowie ein oder mehrere Beobachter, die bei Änderungen benachrichtigt werden.

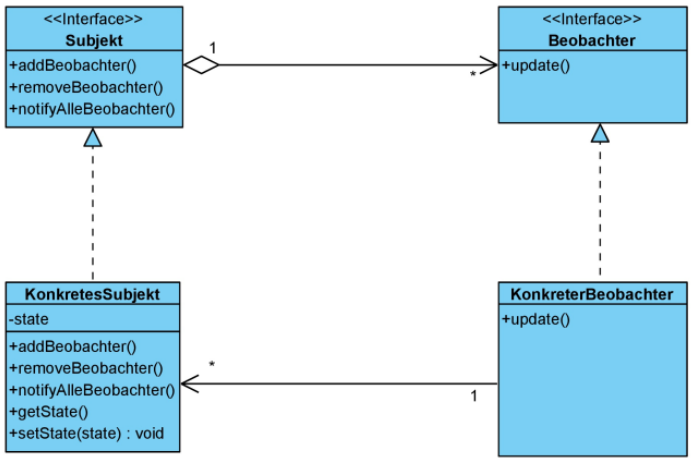


Abbildung 2: Beobachter-Muster [38]

2.1 Abgrenzung zu anderen Paradigmen/Ansätzen

Gemeinsam mit flussbasierter Programmierung, funktional reaktiver Programmierung und ontologiebasierenden Ansätzen gehört das reaktive Paradigma zur Programmierung, die auf Datenflüsse basieren. Sie steht dadurch auch in Kontrast zum imperativen Paradigma, bei der die Änderung von Zuständen im Fokus des Programmes steht. Dadurch ist sie auch eine Unterordnung der deklarativen Programmierung. Es gibt auch Umsetzungen mit Rückruffunktionen.

Gleich wie bei einem Ereignis getriebenen Architektur liegt die asynchrone Kommunikation im Mittelpunkt, um Komponenten voneinander zu trennen [19]. Nachrichten haben hier eine ähnliche Notation wie Ereignisse. Die beiden Begriffe sind nur in der Absicht unterschiedlich, die durch den jeweiligen Begriff hervorgehoben wird. Der reaktive Begriff wurde im *reaktiven Manifest* beschrieben, dass erstmals 2013 von einer Gruppe von Entwicklern veröffentlicht wurde, die von einem Entwickler namens Jonas Boner geleitet wurde [4]. Während das reaktive Manifest die Qualitätsattribute hervorhebt, die sich darauf beziehen, wie sich das System zur Laufzeit verhält und dynamisch reagiert, konzentriert sich die ereignisgesteuerte Architektur darauf, wie ein System aufgegliedert ist und wie seine Teile im Allgemeinen interagieren. Das reaktive Paradigma steht somit auch in Kontrast zu reaktiven Systemen, die Ereignis basiert sind. Das Manifest beschreibt reaktive Systeme wie folgt:

Antwortbereit
: Das System antwortet immer zeitgerecht, sofern es möglich ist. Auch im Fehlerfall muss eine Antwort gesendet werden. Die Antwort muss innerhalb eines bestimmten Timeouts erfolgen, damit eine Fehlererkennung und -behandlung möglich ist. Durch konsistente Antwortzeiten wird mehr Vertrauen in das System gesteckt und die Interaktivität erhöht sich.

Widerstandsfähig Das System antwortet auch im Falle eines Hardware- oder Softwaredefekts. Ein System, das nicht widerstandsfähig ist, kann auch nicht antwortbereit sein. Dies kann durch Replikation der Funktionalität, Eindämmung von Fehlerquellen, Isolation von Komponenten sowie Delegierung von Verantwortung erreicht werden. Dadurch sollte im Fehlerfall nur ein Teil des Systems betroffen sein.

Elastisch Durch Verteilung und Replikation von Funktionalität kann das System auch bei veränderten Lastbedingungen antworten. Bei Änderung der Last werden automatisch die Replikationsfaktoren und damit die genutzten Ressourcen angepasst. Das System darf dafür keine Engpässe enthalten. Die Aufgabe kann dann auf beliebig viele Ressourcen verteilt

werden.

Nachrichtenorientiert Die Nachrichtenübertragung zwischen den Komponenten des Systems erfolgt asynchron. Die explizite Übermittlung der Nachrichten führt zu einer ortsunabhängigen Formulierung des Programms und eine transparente Skalierung von Komponenten ist dadurch möglich. Das Überprüfen der Nachrichtenpuffer kann zur Diagnose, zur automatischen Ressourcensteuerung, Priorisierung und Kontrolle verwendet werden. Ortsunabhängigkeit bedeutet hier, dass Code und Semantik eines Programms nicht davon abhängig sind, ob sie lokale oder verteilt über ein Netzwerk ausgeführt werden.

2.2 Ansätze

Es gibt vier Hauptansätzen, wie das reaktive Paradigma umgesetzt werden kann.

2.2.1 Imperativ

Reaktive Programmierung kann mit imperativer Programmierung kombiniert werden [8]. Dabei arbeiten imperative Programme mit reaktiven Datenstrukturen. Bei der imperativen Programmierung wird durch die Reihenfolge der Befehlt bestimmt, welche Tätigkeiten der Computer auszuführen hat.

2.2.2 Objektorientiert

Objektorientierte reaktive Programmierung kombiniert die objektorientierte Programmierung mit der reaktiven Programmierung. Anstelle der Sicht auf Objekte in der klassischen Sicht durch Methoden und Felder, können Objekte hier auf Ereignisse reagieren [32]. Dies wird durch Methoden wie Neuberechnungen bei Abruf, Caching, Verfolgen von Abhängigkeiten und Akkumulieren von Änderungen erreicht.

2.2.3 Funktional

Die funktionale reaktive Programmierung [18] kombiniert die funktionale Programmierung, die auf der Idee basiert, dass alles durch reine Funktionen ausgedrückt werden kann, mit dem reaktiven Programmierparadigma, dass auf der Idee basiert, dass alles ein Strom (Beobachter-Muster) ist.

2.2.4 Regelbasiert

Bei diesem Ansatz stehen Regeln im Vordergrund. In diesen Programmiersprachen gibt es Reaktionen auf Ereignisse, die alle Bedingungen erfüllen. Dadurch kann die Korrektheit der Software überprüft werden und Reaktionen können ereignisgesteuert sein. Eine relativ neue Kategorie von Programmiersprachen verwendet Einschränkungen als Hauptprogrammierkonzept. Es besteht aus Reaktionen auf Ereignisse, die alle Einschränkungen erfüllt halten. Dies erleichtert nicht nur ereignisgesteuert Reaktionen, sondern macht reaktive Programme zu einem wichtigen Faktor für die Korrektheit von Software. Ein Beispiel für eine regelbasierte reaktive Programmiersprache ist Ampersand, die in der Relationenalgebra begründet ist [22].

2.3 Herausforderungen

Durch eine temporäre Inkonsistenz im Datenflussgraphen kann ein sogenannter *Glitch* entstehen [17]. Da Aktualisierungen nicht sofort stattfinden, sondern Zeit für die Berechnung benötigen, können die Werte innerhalb des Systems während der Aktualisierung vorübergehend nicht synchron sein. Außerdem ist es je nach Art des Systems möglich, dass Knoten mehr als einmal in einer Propagation aktualisiert werden.

Zur Sortierung von Abhängigkeiten muss als Abhängigkeitsgraph ein gerichteter azyklischer Graph verwendet werden. Dieser Graph kann in der Praxis Zyklen enthalten. Diese Zyklen werden bei reaktiven Sprachen entfernt, indem ein Element entlang der Rückkante des Graphen platziert wird, damit die Aktualisierung beendet werden kann. Dies kann z. B. durch einen Verzögerungs-Operator erreicht werden, durch den die nachfolgende Berechnung erst im nächsten Zeitschritt ausgewertet wird und wodurch somit die aktuelle Auswertung abgeschlossen werden kann. Es gibt auch Sprachen, wie Reactive Async [16], die zyklische Abhängigkeiten automatisch erkennen können.

Reaktive Sprachen gehen davon aus, dass Ausdrücke rein funktional sind, also keine Nebenwirkungen zulassen. Durch das Prinzip der referentiellen Transparenz [37] kann beim Aktualisierungsmechanismus die Reihenfolge frei gewählt werden, in der die Aktualisierung stattfinden, um Optimierungen zu ermöglichen. Bei imperative Programmiersprachen, die reaktive Sprachkonstrukte anbieten, führt dies zu Problemen. Es gibt dazu nur Teillösungen wie sogenannte veränderbare Zellen, die dem reaktiven System bekannt sind, sodass Änderungen an der Zelle sich auf den Rest des reaktiven Programms ausbreiten. Auch durch das ordnungsgemäße Kapseln von objektorientierten Bibliotheken ist eine Integration mit einem reaktiven Teil möglich. Beide Ansätze werden in der

Programmiersprache FrTime beispielsweise eingesetzt [7] [20].

Auch die dynamische Aktualisierung des Abhängigkeitsgraphen muss bedacht werden. In manchen reaktiven Sprachen ist dieser Graph statisch und ist somit während der gesamten Programmausführung fest. In anderen Sprachen kann er dynamisch sein und bietet somit eine erhebliche Ausdruckskraft. Dynamische Abhängigkeiten treten beispielsweise bei Benutzeroberflächen häufig auf. Das Aktualisierungssystem muss jedoch entscheiden, ob Ausdrücke immer rekonstruiert werden sollen oder ob die Knoten eines Ausdrucks konstruiert, aber inaktiv bleiben sollen. Im letzteren Fall dürfen diese an Berechnung nicht teilnehmen, wenn sie nicht aktiv sein sollen.

2.4 Vor- und Nachteile dieses Paradigmas

Einige Nachteile wurden bereits im vorherigen Kapitel besprochen. Eine empirische Studie [34] zum Programmverstehen mit reaktiver Programmierung, die Subjekte in eine reaktive und objektorientierte Gruppe teilte, kam zu folgenden Schlüssen: Die Korrektheit des Programmverständnisses erhöht sich und das Verstehen von Programmen im reaktiven Stil benötigt nicht mehr Zeit als das Verstehen des objektorientierten Äquivalents. Dies wurde auch in [36] gezeigt.

Auch sind keine Programmierkenntnisse erforderlich, um reaktive Anwendungen zu verstehen, im Gegensatz zur Objektorientierung, wie die Ergebnisse mit den Programmierkenntnissen korreliert sind. Dies legt nah, dass reaktive Programmierung, die Einstiegshürde zum Verständnis von reaktiven Applikationen senkt. [33] konnten auch zeigen, dass sich der reaktive Ansatz auch für verteilte Applikationen eignet.

3 Demoprojekt

3.1 Theorie

In diesem Abschnitt werden die theoretischen Grundlagen erklärt, die die reaktive Programmierung von anderen Ansätzen trennt und wie manche Probleme der nebenläufigen/parallelen Programmierung elegant gelöst werden können. Diese Arbeit orientiert sich an der Definition des Begriffes Nebenläufigkeit in [3]:

Nebenläufigkeit 3.1.1. *Nebenläufigkeit bezeichnet die Fähigkeit eines Systems, Computerprogramme zur Laufzeit (Prozesse) voneinander unabhängig zu bearbeiten bzw. auszuführen.*

Wie auch im Artikel wird hier zwischen echter Parallelität und einer scheinbaren Parallelität unterschieden. [29] definiert Parallelität wie folgt:

Parallelität 3.1.1. *Arbeitsabläufe bzw. deren Einzelschritte heißen parallel, wenn sie gleichzeitig und unabhängig voneinander durchgeführt werden können.*

Wie in [45] beschrieben, kann ein Prozessor immer nur eine Anweisung ausführen. Nebenläufigkeit kann durch sogenannte Threads umgesetzt werden, indem das Betriebssystem schnell zwischen Threads hin und her wechselt und somit eine Pseudoparallelität erreicht. Ein Thread ist laut [43] wie folgt definiert:

Thread 3.1.1. *Ausführungsstrang oder Ausführungsreihenfolge in der Abarbeitung eines Programms, der Teil eines Prozesses ist.*

In [40] wird ein Thread aus dieser Sicht eines Benutzers auch als leichtgewichtiger Prozess bezeichnet. Die Alternative dazu ist der Einsatz von mehreren Prozessoren. In diesem Projekt wurde die Programmiersprache JavaScript verwendet. JavaScript ist grundsätzlich nicht parallelisierbar, da zur Ausführung nur ein einzelner Thread verwendet wird.

3.1.1 Synchronisation

Eine der Hauptprobleme bei der Arbeit mit Threads ist die Synchronisation mehrere Threads. Es wird hier die Definition von Prozesssynchronisation in [21] auch für Threads verwendet:

Synchronisation 3.1.1. *Unter Prozesssynchronisation (oder kurz Synchronisation) versteht man die Koordinierung des zeitlichen Ablaufs mehrerer nebenläufiger Prozesse.*

Das Wechseln zwischen mehreren Threads wird als Scheduling bezeichnet und ist oft nichtdeterministisch. Dies führt oft zu fehlerhaften Verhalten einer Anwendung. Die Fähigkeit einer nebenläufigen Anwendung, zeitgerecht ausgeführt zu werden, wird als Lebendigkeit bezeichnet [28]. Das Gegenteil, bei dem ein Thread unter Umständen nie zur Ausführung kommt, bezeichnet man als Verhungeren. Es gibt dabei drei häufige Probleme: Verklemmungen, Verhungern und Livelock.

Eine Verklemmung beschreibt eine Situation, in der zwei oder mehr Threads für immer blockiert sind und aufeinander warten. Dies kann geschehen, wenn mehrere Theads zur selben Zeit Betriebsmittel anfordern. Wenn ein Thread ein Betriebsmittel reserviert, das ein anderer Thread benötigt und umgekehrt, kommt es zu einer Verklemmung. Beim Verhungern ist ein Thread nicht in der Lage, regelmäßigen Zugriff auf eine gemeinsam genutztes Betriebsmittel zu erhalten und kann demnach keine Fortschritte machen. Dies geschieht, wenn ein Betriebsmittel durch gierige Threads für lange Zeiträume nicht verfügbar gemacht wird. Threads agieren oft auch als Reaktion auf eine Aktion von anderen Threads. Wenn die Aktion des anderen Threads auch eine Reaktion auf die Aktion eines anderen Threads ist, kann es zu einem Livelock kommen. Wie bei Verklemmungen, können diese Threads keine weiteren Fortschritte machen. Die Threads sind jedoch nicht blockiert, sondern sind zu sehr beschäftigt gegenseitig zu reagieren, um ihre Arbeit fortzusetzen. In Bezug zu diesem Thema muss auch die sogenannte Wettlaufsituation erwähnt werden. Dies geschieht, wenn mehrere Threads in fehlerhafter Weise interagieren, wobei die Reihenfolge der Ausführung der verschiedenen Anweisungen entscheidend ist.

3.1.2 Warteschlangen

Im Bezug auf das Demoprojekt müssen noch Warteschlangen erwähnt werden. Warteschlangen sind spezielle Datenstrukturen, bei der die Speicherung nach dem sogenannten FIFO-Prinzip (first in, last out) funktioniert: Das Element das zuerst in der Warteschlange ist, wird auch zuerst wieder entfernt. Warteschlangen können zum Austausch von Daten verwendet werden. Sie können beim Auslesen und Einfügen von Elementen blockierend, nicht blockierend und zeitlich blockierend sein. [45] nennt einige Beispiele für Warteschlangen. Grafische Benutzeroberflächen verwenden Warteschlangen, um Benutzeraktionen

darzustellen. Die Anwendung kann diese dort abrufen und verarbeiten. Sie werden auch als sogenannte Spooler genutzt, die Aufträge zwischenspeichern. Ein bekannter Spooler ist der Druckerspooler, der Druckaufträge zwischenspeichert und einzeln an den Drucker weitergibt, damit dieser den Auftrag abarbeiten kann. Auch in sogenannten Message-Queueing-Systeme werden Warteschlangen verwendet. Sie verbinden mehrere Anwendungen, die durch Austausch von Nachricht über diesem System kommunizieren. Dabei erfolgt der Empfang und die Weiterleitung von Nachrichten erst dann, wenn der Empfänger verfügbar ist. Die einzelnen Nachrichten können dabei auch nach Prioritäten sortiert, weitergeleitet werden. Diese Prioritätswarteschlangen gibt es bereits seit einigen Jahrzehnten [41].

3.1.3 Modellierung

In der Modellierungsphase eines Projektes werden manchmal Diagramme des Softwaresystems erstellt. Ein bekanntes Werkzeug dafür sind UML-Diagramme [31]. Das UML-Sequenzdiagramm eignet sich gut, um eine Interaktion zwischen mehreren Objekten oder Komponenten darzustellen. Es hat eine Semantik, die den Unterschied zwischen synchroner und asynchroner Kommunikation deutlich macht. Sie können sogar Verzögerungen bei der Nachrichtenübergabe darstellen. Für komplexere Interaktionen zwischen Systemen sollte jedoch ein BPMN-Kollaborationsdiagramm verwendet werden [1]. Sie wurden speziell für diesen Zweck entwickelt. Sie haben den Vorteil, dass sie die Darstellung erleichtern, wie jedes System auf Ereignisse in einem Fluss von zusammenhängenden Aufgaben reagiert, einschließlich außergewöhnlicher Ereignisse wie Zeitüberschreitungen und Fehler. Ein einfaches Sequenzdiagramm ist in Abbildung 3 dargestellt.

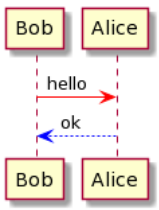


Abbildung 3: UML-Sequendiagramm

Die Limitierungen von UML werden oftmals in Diskussionen im Internet besprochen [39]. Zwei Kritikpunkte sind die fehlende Ausdruckskraft und das Problem, die Diagramme synchron mit dem Quellcode zu halten. Rx verwendet zur Darstellung von Operationen die in Abbildung dargestellten 4 Marble-

Diagramme.

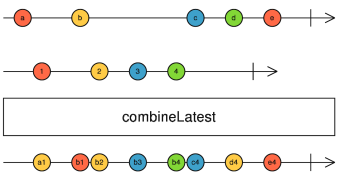


Abbildung 4: combineLatest-Operator

Diese Diagramme sind sehr anschaulich und erklären konkrete Operatoren im Gegensatz zu UML-Diagramme, die häufig auf einer noch höheren Abstraktionsebene arbeiten, allerdings sind sie von den gleichen Problemen geplagt. Sie können auch nicht einfach kombiniert werden und sie sind nicht genormt wie UML, dass in ISO/IEC 19505 spezifiziert wurde.

3.1.4 Aktualisierung grafischer Oberflächen

Grafische Oberfläche in vielen Anwendungen wie Android-Apps, Applikationen in Java [30] oder auch iOS-Anwendungen [9] verwenden einen dedizierten GUI-Thread, in dem alle Aufgaben abgearbeitet werden, die mit der grafischen Oberfläche zu tun haben wie z. B. neue Ausrichten und Zeichnen von grafischen Elementen oder auch das Aktualisieren eines Eingabefeldes. Zur Kommunikation mit diesem Thread gibt es verschiedenste Ansätze in unterschiedlichen Programmiersprachen, um Anweisungen aus anderen Threads auf dem GUI-Thread auszuführen. Der GUI-Thread ist auch oft gleichzeitig der Main-Thread, der beim Programmstart immer läuft. Dies hat zur Folge, dass lange Operationen in andere Threads verschoben werden müssen. Ansonsten wird der GUI-Thread/Main-Thread zu lange blockiert und die grafische Benutzeroberfläche scheint eingefroren zu sein und reagiert auf keine Benutzereingaben und Ereignisse mehr. Auch sind die Bibliotheken zur Erstellung von grafischen Oberflächen nicht immer threadsicher. Dies sind Probleme, die auch die reaktive Programmierung nicht lösen kann, allerdings kann die Datenweitergabe an den UI-Thread von einem anderen Thread durch einen besseren Datenfluss verbessert werden.

Zur Ausgabe von User-Interface-Komponenten wird in dem Demoprojekt React [11] verwendet. In React ist jedes UI-Teil eine Komponente, und jede Komponente hat einen Zustand. React folgt dem Observable-Muster und hört auf Zustandsänderungen. Wenn sich der Zustand einer Komponente ändert, aktualisiert React einen virtuellen DOM-Baum. DOM steht für Document Objekt Model und ist eine plattform- und sprachunabhängige Schnittstelle, die

ein XML- oder HTML-Dokument als Baumstruktur behandelt, wobei jeder Knoten ein Objekt ist, das einen Teil des Dokuments darstellt. Sobald das virtuelle DOM aktualisiert wurde, vergleicht React die aktuelle Version des virtuellen DOM mit der vorherigen Version des virtuellen DOM. Dieser Prozess wird als *diffing* bezeichnet. Sobald React bekannt ist, welche virtuellen DOM-Objekte sich geändert haben, aktualisiert React nur noch diese Objekte im realen Dom. Dies verbessert die Geschwindigkeit deutlich im Vergleich zur direkten Manipulation des realen DOMs.

Abschließend ist noch zu erwähnen, dass der Name *React* hier irreführend ist. [42] beschreibt in seinen Vortrag, den Unterschied zwischen funktional-reaktiven Bibliotheken wie RxJS und React. In RxJS bietet Werkzeuge, um Ereignisse deklarativ zu behandeln und den Zustand zu verwalten. Während die Render-Funktion auf Zustandsänderungen reagiert, ist React selbst nicht reaktiv. Es werden stattdessen imperative Event-Handler geschrieben, die von sämtlichen Synchronisationsproblemen, die bereits ausgeführt wurden, geplagt sind. In dem Vortrag wird gezeigt, dass eine reaktive Implementierung von React möglich ist.

3.2 Praxis

Zur Umsetzung des Projektes wurde RxJS verwendet, die Implementierung in JavaScript des Frameworks Reactive Extensions. Die grafische Oberfläche wurde mit dem JavaScript-Framework React [10] erstellt. Wie in Abbildung 5 abgebildet, werden drei verschiedene Datenströme verwendet: *InputStream*, *CSVStream* und *CombinedStream*. In *InputStream* wird ein Eingabeelement mit Hilfe von React-Hooks mit Reactive Extensions verbunden, um aus dem Textfeld ein *Observer* zu erstellen. In der Funktion *parse* in *CSVStream* wird der Event-Handler des Einlesens der CSV-Datei in ein Promise umgewandelt, um eine asynchrone Ausführung zu erlauben. Die Methode wird dann in *create()* aufgerufen und mit Hilfe des *await* Schlüsselwortes, in einen synchronen Aufruf umgewandelt. Die einzelnen Tweets werden dann aus der CSV-Datei gelesen und Blöcke der Größe n aufgeteilt.

Damit sich der Strom wie ein echter Twitterfeed verhält, wird jedem Block eine Zeitverzögerung hinzugefügt. Der *CombinedStream* verbindet die beiden anderen Ströme, indem bei Eingabe eines Zeichens, die neuesten Daten der beiden Ströme ausgelesen werden und die Tweets für die grafische Darstellung vorbereiteten werden: Zuerst werden die Tweets nach Schlüsselwörtern durchsucht, danach werden die eingelesenen CSV-Dateien in einfache JSON-Objekte umgewandelt. Zuletzt kann der *CombinedStream* abonniert werden

und die ausgelesenen Tweets grafisch dargestellt werden.

Wie in der Einführung zu RxJS beschrieben, kann es auch in der reaktiven Programmierung zu den Tücken der Gleichzeitigkeit kommen [5]. Dies kann zu Wartungsproblemen führen, die sich in dem Bereich Debugging, Testen und Refaktorisierung bemerkbar machen. Diese oft durch unvorhersehbares Timing hervorgerufenen Probleme, werden oft durch Entwickler stillschweigend als Fehler eingeführt. Diese Fehler werden oft bei der Entwicklung, der Qualitätskontrolle und Akzeptanztesten nicht gefunden und werden erst in der Produktion sichtbar. Reaktive Ansätze wie Rx vereinfachen die simultane Verarbeitung von beobachtbaren Sequenzen jedoch ausgezeichnet, sodass viele dieser Probleme nur in geringen Maß auftreten. Durch das Einhalten von Richtlinien können solche Probleme stark reduziert werden.

Es wird hier vergleichsweise auf Synchronisationsmechanismen der Allzweckprogrammiersprache Java eingegangen, wie sie beispielsweise in [44] beschrieben sind. Zur Umsetzung der Nebenläufigkeit werden Threads verwendet.

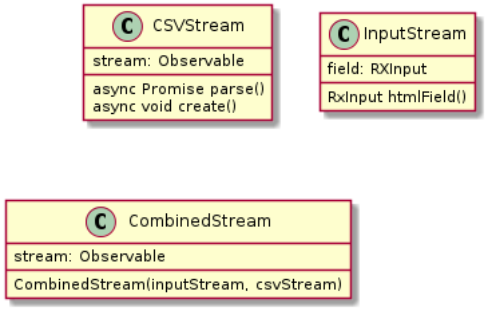


Abbildung 5: Streams (Beobachter-Muster)

Eine Variante, um Methoden in Java threadsicher zu machen, ist diese mit dem *synchronized* Schlüsselwort zu versehen, damit diese Methode nur von einem Thread gleichzeitig ausgeführt werden kann. Dies ist in Listing 10 dargestellt.

```
1 synchronized void incrementSync() {
2     count = count + 1;
3 }
```

Listing 1: Synchronized Methode in Java

Einzelne Abschnitte des Programms können auch wie in Listing 2 mit verschiedenen Arten von Locks gesperrt werden.

```
1  ReentrantLock lock = new ReentrantLock();
2  int count = 0;
3  void increment() {
4      lock.lock();
5      try {
6          count++;
7      } finally {
8          lock.unlock();
9      }
10 }
```

Listing 2: Lock in Java

Zusätzlich gibt es atomare Variablen und Datenstrukturen, die von mehreren Threads gleichzeitig ausgeführt bzw. verwendet werden können. Ein Beispiel für atomare Ganzzahlen ist in Listing 3 dargestellt.

```
1  AtomicInteger atomicInt = new AtomicInteger(0);
2
3  ExecutorService executor = Executors.newFixedThreadPool(2);
4
5  IntStream.range(0, 1000)
6      .forEach(i ->
           executor.submit(atomicInt::incrementAndGet));
7
8  stop(executor);
9
10 System.out.println(atomicInt.get());
```

Listing 3: AtomicInteger in Java

Dies lässt sich in reaktiver Programmierung eleganter lösen. In diesem Projekt werden Threads durch RxJS-Ströme abstrahiert, die das Beobachter-Muster implementieren. Die Daten von mehreren Strömen können auch kombiniert werden. In Listing 4 wird der Eingabestrom, der die Eingabe des Eingabefeldes enthält, mit dem Twitterdatenstrom kombiniert.

```
return inputStream.field.onValueChanges$
.pipe(
    withLatestFrom(csvStream.stream),
    map(([words, second]) => {
        return second.filter(findWords).map(formatTweetData)
    })
)
```

Listing 4: CombinedStream

Wie in den letzten Listings erkennbar ist, sind Synchronisationsmechanismen außerhalb der Observables meist nicht notwendig, da nebenläufige Programmteile voneinander getrennt werden und keine gemeinsamen Daten verarbeiten. Stattdessen werden Datenströme verändert oder kombiniert. Dies wird mithilfe von reinen Funktionen umgesetzt, damit durch einen Funktionsaufruf keine Daten wie in der imperativen Programmierung verändert werden, sondern nur mit Kopien gearbeitet wird. Diese Funktionen werden in Rx Operatoren genannt und wandeln ein Observable in ein neues Observable um. Die asynchronen Ströme können nun abonniert werden und beim Eintreffen von Daten synchron oder asynchron weiterverarbeitet werden. Es können auch spezielle Observables erstellt werden, die von mehreren Beobachtern unabhängig voneinander abonniert werden können. Dies entspricht in der Telekommunikation einem Multicast.

Der bereits erwähnte Datenstrom mit Twitter-Nachrichten entspricht einer nebenläufigen Warteschlange. Zu Testzwecken wurde kein echter Twitter-Feed verwendet, sondern die Tweets wurden einer CSV-Datei mit Test-Tweets entnommen und dem Strom hinzugefügt. Damit sich der Strom wie ein echter Feed verhält, wurden die Tweets zu fixen Längen gruppiert und in fixen Intervallen im Strom veröffentlicht. Normalerweise muss dazu eine spezielle Datenstruktur wie z.B. in BlockingQueue in Java verwendet werden und Code-Abschnitte müssen synchronisiert werden. Diese Herausforderungen fallen in RxJs weg und können elegant mit Strömen sowie einfachen Operationen wie *delay* und *bufferCount* umgesetzt werden, die eine Verzögerung der Datenausgabe im Strom sowie einer Pufferung der Daten hervorrufen.

Der bereits erwähnte kombinierte Strom wird wie in Listing 5 abonniert und die neu eintreffenden Daten werden an die React-Anwendung zur Anzeige übergeben.

```
1  async componentDidMount() {
2    this.inputStream.field.onValueChanges$.subscribe(input
       => this.setState({keyword: input}))
3    await this.csvStream.create()
4    this.csvStream.stream.subscribe((tweets) => {
5      this.setState({ offset: this.state.offset + 1})
6    })
7    this.combinedStream = new
       CombinedStream(this.inputStream, this.csvStream)
8    this.combinedStream.stream.subscribe((tweets) => {
9      this.setState({tweets:tweets})
10   })
11 }
```

Listing 5: Mount-Methode der Haupt-Komponente

3.3 Parallelisierung des Projektes

Die hier angeführten Parallelisierungstechniken beziehen sich nicht nur auf dieses Projekt, sondern sollen allgemein zeigen, wie JavaScriptprojekte mit RxJS parallesiert werden können.

JavaScript benutzt nur einen einzelnen Thread. Es können allerdings sogenannte Web Worker [14] verwendet werden, dies es ermöglichen, Skripte in einem Hintergrund-Thread getrennt vom Haupt-Thread auszuführen. Rechneintensive Operationen können somit im Hintergrund ausgeführt werden, ohne dass der Haupt-Thread, der unter anderem für die grafische Oberfläche zuständig ist, blockiert wird. Das Auslesen der CSV-Datei kann in diesem Projekt nicht parallelisiert werden, allerdings kann mithilfe eines Web Worker der Vorgang im Hintergrund ausgeführt werden, um den Hauptthread nicht zu belasten.

Alternativ gibt es die Worklet-Schnittstelle, eine abgespeckte Version von Web Workers. Sie gibt Entwicklern Zugriff auf einen Low-Level der Rendering-Pipeline [27]. Mit Worklets kann JavaScript- und WebAssembly-Code ausgeführt werden, um Grafikrendering, Audioverarbeitung, Animationen etc. durchzuführen, wenn hohe Leistung erforderlich ist. Dies ist hier nicht der Fall und wurde nur der Vollständigkeit halber erwähnt. Eine weitere experimentelle Technologie sind Service Workers [13]. Ein Service Worker ist ein Skript, das im Browser im Hintergrund ausführt wird z. B. die Tür zu Funktionen öffnet, die keine Webseite oder Benutzerinteraktion benötigen. Sie werden z. B. für Push-Benachrichtigungen und Synchronisationen im Hintergrund bereits verwendet.

3.4 Verwendete RxJS Muster

RxJS verwendet verschiedene Operatoren, die auch in anderen Frameworks, Bibliotheken und Programmiersprachen verwendet werden und aus der reaktiven oder funktionalen Programmierung stammen. In diesem Abschnitt werden die verwendeten Muster vorgestellt. Die Marble-Diagramme wurden der offiziellen Dokumentation entnommen.

Piping

Beim Piping werden einzelne Operatoren zu einem großen Operator zusammengefügt. Da Operatoren Funktionen sind, entspricht dies einer Funktionskomposition. Bsp: *obs.pipe(op1(), op2(), op3())*

WithLatestFrom

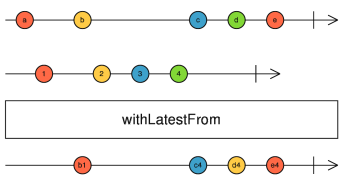


Abbildung 6: withLatestFrom

Kombiniert die Beobachtungsvariable der Eingabe mit anderen Beobachtungsvariablen, um eine Beobachtungsvariable zu erstellen, deren Werte aus den jeweils neuesten Werten berechnet werden, sofern die Quelle Daten sendet.

Delay

Verzögert das Senden von Elementen aus dem Observable um ein bestimmtes Timeout. Bsp: *obs.delay(1000)*

BufferCount

Puffert die Quellwerte von Observable, bis die Größe die angegebene maximale Puffergröße erreicht. Bsp: *obs.bufferCount(5)*

ConcatMap

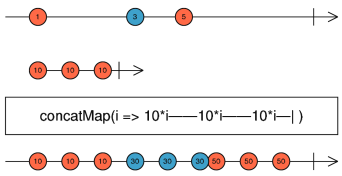


Abbildung 7: concatMap

Wandelt jeden Wert in ein Observable um und abonniert das Observable.

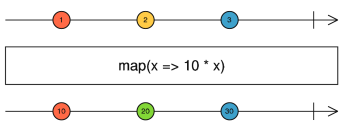
Take

Gibt nur die ersten n Werte aus, die vom Observable ausgegeben werden. Bsp: *obs.take(5)*

Skip

Gibt ein Observable zurück, das die ersten n ausgegebenden Werte vom Observable überspringt. Bsp: *obs.skip(2)*

Map



Wendet eine gegebene Funktion auf jeden Wert an, der vom Observable ausgegeben wird, und gibt die resultierenden Werte als Observable aus.

Abbildung 8: map

Es gibt viele weitere Operatoren, die insgesamt in die folgenden Kategorien aufgeteilt werden können (es wurden hier die Originalbezeichnungen übernommen): Creation Operators, Join Creation Operators, Transformation Operators, Filtering Operators, Join Operators, Multicasting Operators, Error Handling Operators, Utility Operators, Conditional and Boolean Operators sowie Mathematical and Aggregate Operators.

3.5 Koroutinen als alternative Implementierung

kann Speziell in der Java- und Androidwelt setzen sich sogenannte Koroutinen [25] gegenüber Rx durch. Dies sind Funktionen, die ihre Ausführung unterbrechen können und später wieder ihre Arbeit aufnehmen können. Sie können auch mit Datenströmen wie Rx umgehen und werden hier aufgrund der hohen Popularität kurz erklärt.

Koroutines gibt es schon sehr lang und werden in verschiedenen Programmiersprachen umgesetzt. Es wird hier die Implementierung der Programmiersprache Kotlin betrachtet, dessen Programme nach JavaScript transpiliert werden können. Durch das Hinzufügen des Schlüsselwortes *suspend* vor der Funktionsdeklaration kann eine Funktion unterbrochen werden.

Es gibt auch die Couroutine Flow API, sodass der Code ähnlich wie bei Rx aussieht. Ein Beispiel dafür ist in Listing 6 zu sehen, in dem 10 Zahlen hintereinander mit 100 Millisekunden ausgegeben werden, der Auftrag aber bereits nach 400 Millisekunden abgebrochen wird.

```
fun main() = runBlocking {
    val job = launch {
        flow.collect {
            println("Item: $it")
        }
    }
    delay(400)
    job.cancel()
}
val flow = flow {
    for (i in 1..10) {
        delay(100)
        emit(i)
    }
}
// Output:
// Item: 1
// Item: 2
// Item: 3
```

Listing 6: Beispiel in Flow

Koroutinen wurden mit Rx in [6] von mehreren Autoren verglichen. Koroutinen sind demnach nur ein dünnes Feature der Programmiersprache mit ein paar grundlegenden Funktionen, während Rx eine umfangreiche Bibliothek mit vielen fertigen Operatoren ist.

Rx bringt einen bestimmten funktionalen Programmierstil mit sich, der in jeder Programmiersprache ohne besondere Unterstützung umgesetzt werden kann. Es eignet sich für Probleme, die sich in eine Folge von Standardoperatoren zerlegen lassen. Koroutinen in Kotlin sind ein Feature der Sprache, mit dem verschiedene asynchrone Programmierstile implementiert werden können und ist nicht beschränkt auf den funktional reaktiven Stil wie Rx. Der asynchrone Code kann daher auch im imperativen Stil, mit Promises/Futures [15] oder mit Aktoren [2] usw. geschrieben werden.

Ein Vorteil, der durch den Einsatz des Schlüsselworts *suspend* entsteht, ist, dass der Code als synchron empfunden wird. Es fällt auch die Rückruffunktion weg, die als Parameter an die Funktion normalerweise übergeben wird, da ein direkter Sprung zurück in die Funktion möglich ist.

4 Fazit

In dieser Arbeit wurde nach einer kurzen Einführung im folgenden Kapitel die reaktive Programmierung näher erläutert. Die reaktive Programmierung wurde zu anderen Paradigmen abgegrenzt und es wurden die verschiedenen Ansätze vorgestellt, wie reaktive Programmierung umgesetzt werden kann. Nachdem die Herausforderungen erklärt wurden, wurden die Vor- und Nachteile dieses Paradigmas aufgezählt.

Der Hauptteil dieser Arbeit beschäftigte sich mit einem Demoprojekt, in dem ein Twitter-Feed simuliert wurde, der live nach Stichwörtern durchsucht werden kann. Das Projekt wurde mit dem reaktiven Framework Reactive Extensions in JavaScript umgesetzt (RxJS). Die gefilterten Daten wurden mit der Javascript-Bibliothek React live dargestellt.

Nachdem theoretische Themen wie Synchronisation, Warteschlangen, Modellierung sowie Aktualisierung grafischer Oberflächen durchgenommen wurden und auch Vergleiche zu einer imperativen Programmiersprache (Java) hergestellt wurden, wurde die Implementierung des Demoprojektes in der Praxis unter die Lupe genommen. Es wurde auch festgestellt, dass Parallelisierung kein wichtiger Aspekt bei der reaktiven Programmierung ist und auch nicht das Hauptkriterium zur Wahl dieses Paradigmas sein sollte. Vielmehr steht das Vermeiden von Fehlern in der nebenläufigen Programmierung sowie Datenflüsse im Vordergrund.

Als Nächstes wurden einige Operationen kurz erklärt, die typisch für Anwendungen sind, die Reactive Extensions verwenden. Diese Muster/Funktionen können auch in anderen Bibliotheken und Programmiersprache gefunden werden. Die Arbeit hat gezeigt, dass das reaktive Paradigma oft eine gute Wahl ist, wenn Daten als Datenströme gesehen werden können und diese verarbeitet und mit anderen Strömen kombiniert werden müssen. Es wurde auch gezeigt, dass einige Fehlerquellen der nebenläufigen Programmierung vermindert bzw. verhindert werden können. Zuletzt wurden Koroutinen als alternative Implementierungsweise vorgestellt.

Trotz der Beliebtheit von Reactive Extensions und der reaktiven Programmierung darf nicht vergessen werden, dass es nur ein spezielles Paradigma ist, um Datensequenzen verarbeiten zu können. Es können beispielsweise auch alle Probleme mit den bereits erwähnten Koroutinen gelöst werden.

Listings

Literatur

[1] Chris Adams. What is a bpmn collaboration diagram? https://www.modernanalyst.com/Careers/InterviewQuestions/tabid/128/ID/2529/What-is-a-BPMN-Collaboration-Diagram.aspx. [Online; abgerufen am 15.03.2021].

[2] Gul Agha. Concurrent programming using actors. *Object-oriented concurrent programming*, pages 37–53, 1987.

[3] Stephan Augsten. Was ist nebenläufigkeit? https://www.dev-insider.de/was-ist-nebenlaeufigkeit-a-926206. [Online; abgerufen am 20.03.2021].

[4] Jonas Boner. Das reaktive manifest. https://www.reactivemanifesto.org/de. [Online; abgerufen am 4.03.2021].

[5] Lee Campbell. Intro to rx - scheduling and threading. http://introtorx.com/Content/v1.0.10621.0/15_SchedulingAndThreading.html. [Online; abgerufen am 15.03.2021].

[6] charlie_pl. How kotlin coroutines are better than rxkotlin? - stack overflow. https://stackoverflow.com/questions/42066066/how-kotlin-coroutines-are-better-than-rxkotlin. [Online; abgerufen am 15.03.2021].

[7] Gregory H Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308. Springer, 2006.

[8] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. *ACM SIGPLAN Notices*, 46(10):407–426, 2011.

[9] Dywanedu. ios: Why the ui need to be updated on main thread — medium. https://medium.com/@duwei199714/ios-why-the-ui-need-to-be-updated-on-main-thread-fd0fef070e7f. [Online; abgerufen am 15.03.2021].

[10] Artemij Fedosejev. *React. js essentials*. Packt Publishing Ltd, 2015.

[11] Cory Gackenheimer. *Introduction to React*. Apress, 2015.

[12] Erich Gamma and Richard Helm. Ralph johnson and john vlissides. entwurfsmuster, 1996.

[13] Matt Gaunt. Service workers: an introduction — web fundamentals. https://developers.google.com/web/fundamentals/primers/service-workers. [Online; abgerufen am 15.03.2021].

[14] Ido Green. *Web workers: Multithreaded programs in javascript*. O'Reilly Media, Inc., 2012.

[15] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and promises. *URL: http://docs. scala-lang. org/overviews/core/futures. html*, 2012.

[16] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. Reactive async: expressive deterministic concurrency. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pages 11–20, 2016.

[17] Li Haoyi. lihaoyi/scala.rx — glitchiness and redundant computation. https://github.com/lihaoyi/scala.rx#glitchiness-and-redundant-computation. [Online; abgerufen am 4.03.2021].

[18] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *International School on Advanced Functional Programming*, pages 159–187. Springer, 2002.

[19] Benutzer 'Hyggenbodden'. What's the difference between reactive programming and event driven architecture? https://softwareengineering.stackexchange.com/a/405955. [Online; abgerufen am 4.03.2021].

[20] Daniel Ignatoff, Gregory H Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *International Symposium on Functional and Logic Programming*, pages 259–276. Springer, 2006.

[21] Günther Jena. Synchronisation. https://www.semiversus.com/dic/komplexe_digitale_systeme/synchronisation.html. [Online; abgerufen am 20.03.2021].

[22] Stef Joosten. Relation algebra as programming language using the ampersand compiler. *Journal of Logical and Algebraic Methods in Programming*, 100:113–129, 2018.

[23] Ingo Maier and Martin Odersky. Deprecating the observer pattern with scala. react. Technical report, 2012.

[24] Sergi Mansilla. *Reactive Programming with RxJS 5: Untangle Your Asynchronous JavaScript Code*. Pragmatic Bookshelf, 2018.

[25] Christopher D Marlin. *Coroutines: a programming methodology, a language design and an implementation*. Number 95. Springer Science & Business Media, 1980.

[26] Erik Meijer. Reactive extensions (rx) curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, pages 1–1. 2010.

[27] Mozilla. Worklet - web apis — mdn. `https://developer.mozilla.org/en-US/docs/Web/API/Worklet`. [Online; abgerufen am 15.03.2021].

[28] Oracle. Lesson: Concurrency (the java™ tutorials ¿ essential classes). `https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`. [Online; abgerufen am 15.03.2021].

[29] Universität Potsdam. Definition nebenläufigkeit - parallelität. `http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/SeminarDidaktik/Nebenlaeufigkeit/nebenlaeufig.htm`. [Online; abgerufen am 20.03.2021].

[30] QuarkPhysics. Java basics: main and the gui event dispatch thread. `http://quarkphysics.ca/ICS3U1/LeepointJava/JavaBasics/gui/gui-commentary/guicom-main-thread.html`. [Online; abgerufen am 15.03.2021].

[31] Bernhard Rumpe. *Modellierung mit UML*. Springer Berlin Heidelberg, 2011.

[32] Guido Salvaneschi and Mira Mezini. Towards reactive programming for object-oriented applications. In *Transactions on Aspect-Oriented Software Development XI*, pages 227–261. Springer, 2014.

[33] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. Towards distributed reactive programming. In *International Conference on Coordination Languages and Models*, pages 226–235. Springer, 2013.

[34] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming.

In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 564–575, 2014.

[35] Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. Reactive programming: A walkthrough. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 953–954. IEEE, 2015.

[36] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017.

[37] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990.

[38] Christoph Tornau. Das observer-pattern / beobachter-muster. `http://www.tornau.name/2014/02/das-observer-pattern-beobachter-muster/`. [Online; abgerufen am 4.03.2021].

[39] user4051. Limitations of uml? - software engineering stack exchange. `https://softwareengineering.stackexchange.com/questions/145949/limitations-of-uml/227763`. [Online; abgerufen am 15.03.2021].

[40] Uresh Vahalia. *UNIX internals: the new frontiers*. Pearson Education India, 1996.

[41] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10 (1):99–127, 1976.

[42] Shawn Swyx Wang. Why react is not reactive - react rally cfp · github. `https://gist.github.com/sw-yx/9bf1fad03185613a4c19ef5352d90a09`. [Online; abgerufen am 15.03.2021].

[43] Wiktionary. Thread – wiktionary. `https://de.wiktionary.org/wiki/Thread`. [Online; abgerufen am 20.03.2021].

[44] Benjamin Winterberg. Java 8 concurrency tutorial: Synchronization and locks. `https://winterbe.com/posts/2015/04/30/`

java8-concurrency-tutorial-synchronized-locks-examples/. [Online; abgerufen am 15.03.2021].

[45] Peter Ziesche and Doga Arinir. *Java: nebenläufige & verteilte Programmierung: Konzepte, UML-2-Modellierung, Realisierung in Java*. W3l GmbH, 2010.

```
1
2  import csvFile from '../data/test.csv'
3
4  import Papa from 'papaparse'
5  import { from, of } from 'rxjs';
6  import { concatMap, delay,take,
       withLatestFrom,skip,bufferCount } from 'rxjs/operators';
7
8
9  export default class CSVStream {
10
11   constructor() {
12     this.size = 50
13   }
14
15     async parse() {
16         return new Promise(resolve => {
17           Papa.parse(csvFile, {
18             download: true,
19             complete: results => {
20               resolve(results.data);
21             }
22           });
23         });
24       };
25
26     async create() {
27       let csvData = await this.parse()
28         let csvFile = from(csvData)
29         let header = csvFile.pipe(take(1))
30         let rows = csvFile.pipe(skip(1))
31         this.stream = rows.pipe(withLatestFrom(header,
              (row, header) => {
32           return row.reduce((rowObj, cell, i) => {
33               rowObj[header[i]] = cell;
34               return rowObj;
35           }, {});
36       }),bufferCount(this.size)).pipe(concatMap(val =>
            of(val).pipe(delay(2000))))
37     }
38 }
```

Listing 7: csvStream.js

```
import { withLatestFrom, map } from 'rxjs/operators';


export default class CombinedStream {
    constructor(inputStream, csvStream) {
        this.stream = this.create(inputStream, csvStream);
    }

    create(inputStream, csvStream) {
        return inputStream.field.onValueChanges$.pipe(
        withLatestFrom(csvStream.stream),
        map(([words, second]) => {
          return second.filter(tweet =>
              words.split(/\s+/).some(word => word !== "" &&
              tweet.text.includes(word))).map(tweet => {
            let obj = { author: tweet.screen_name, text:
                tweet.text}
            return obj
          })
        })
      )
    }
}
```

Listing 8: combinedStream.js

```
import { rxInput} from 'reactivehooks'

export default class InputStream {
    constructor() {
        this.field = rxInput("text")
    }

    get htmlField() {
        return this.field;
    }
}
```

Listing 9: inputStream.js

```
import './App.css';

import React from 'react'
import { Grid, Header } from 'semantic-ui-react'

import InputStream from './streams/inputStream'
import CSVStream from './streams/csvStream'
import CombinedStream from './streams/combinedStream'

import Tweets from './components/tweets'

class App extends React.Component {

  constructor(props) {
    super(props)
    this.state = {
      offset: 0,
      tweets: []
    }
    this.inputStream = new InputStream()
    this.csvStream = new CSVStream()
  }

  async componentDidMount() {
    this.inputStream.field.onValueChanges$.subscribe(input =>
        this.setState({keyword: input}))
    await this.csvStream.create()
    this.csvStream.stream.subscribe((tweets) => {
      this.setState({ offset: this.state.offset + 1})
    })
    this.combinedStream = new CombinedStream(this.inputStream, this.csvStream)
    this.combinedStream.stream.subscribe((tweets) => {
      this.setState({tweets:tweets})
    })
  }

  render() {
    let InputField = this.inputStream.htmlField
    return (
      <Grid padded centered>
        <Grid.Row>
          <Header as="h1">Live Data Search</Header>
        </Grid.Row>
        <Grid.Row>
          <Grid.Column>
            <div className="ui fluid input"><InputField /></div>
          </Grid.Column>
        </Grid.Row>
        <Grid.Row>
          <Grid.Column>Current tweets:
              {this.state.offset*this.csvStream.size} -
              {(this.state.offset+1)*this.csvStream.size}</Grid.Column>
        </Grid.Row>
        <Grid.Row>
          <Grid.Column><Tweets tweets={this.state.tweets}
              keyword={this.state.keyword}/></Grid.Column>
        </Grid.Row>
      </Grid>
    )
  }
}

export default App
```


Listing 10: App.js