Alles im Fluss

Reactive Extensions in Java

Phillip Ghadir

Reactive Programming ist ein datenflusszentriertes Programmierparadigma. In objektorientierten Programmiersprachen ermöglicht das Observer-Muster den Datenfluss entgegen der gewünschten Aufrufabhängigkeiten: Von einem Subjekt abhängige Beobachter werden automatisch über Änderungen dieses Subjektes informiert und können den aktuellen Zustand erfragen. Mit Reactive Extensions wird das Paradigma so ergänzt, dass sich leicht die parallele Verarbeitung von Daten implementieren lässt.

Das Programmieren von Datenflüssen kann schwierig werden, wenn man die Veränderungen von Daten über die Zeit konsistent halten will. Bibliotheken wie `java.util.concurrent` ermöglichen zwar den synchronisierten Zugriff auf Daten, machen das Schreiben fehlerfreier Software aber zur Tortur, an der auch Anwender teilhaben: Sie kennen Systeme, die immer mal wieder Fehlverhalten zeigen, welches sich nicht einfach reproduzieren lässt beziehungsweise dessen Ursache nicht einfach isoliert werden kann.

Datenfluss und Reactive Programming

Eine Tabellenkalkulation ist ein anschauliches Beispiel für einen Datenfluss-Automaten: Wenn eine Zelle eine Formel enthält, die besagt, dass der Zell-Wert die Summe von A1 und B1 ist, dann enthält zu jedem Zeitpunkt die Zelle den aktuellen Wert von A1 + B1. Wenn wir in Java schreiben:

```
int zelle = A1 + B1;
```

wird die Formel unmittelbar ausgewertet und das Ergebnis in `zelle` abgelegt. Eine nachträgliche Änderung von `A1` schlägt in Java aber nicht auf den Wert von `zelle` durch. Ganz anders als in einer Tabellenkalkulation. Um so ein Problem zu lösen, habe ich in [Ghad11] beschrieben, wie man ein solches Verhalten geeignet kapseln kann. Es geht aber auch generischer.

FRAN – Der Anfang

1997 haben Elliot und Hudak das Paper „Functional Reactive Animation“ (FRAN, [ElHu97]) veröffentlicht und damit den Grundstein zur Forschung zum Functional Reactive Programming gelegt. In dem Beitrag ging es im Wesentlichen darum, die nahezu Echtzeitanforderungen der Präsentation von interaktiven Animationen von der Modellierung von Zustandsänderungen zu trennen. Die These: Das Präsentieren ist eine automatisch und effizient optimierbare Aufgabe, das Modellieren hingegen ist 1997 eine schwierige Aufgabe. Die Essenz der Modellierung von Reactive Animations ist laut dem Paper:

- der Verlauf über die Zeit,
- Ereignismodellierung (sowohl für Eingabe-Ereignisse als auch systeminterne Ereignisse – wie z. B. „Wartezeit auf Antwort zur Anfrage ist abgelaufen“),
- Deklaration von Reaktionen,
- polymorphe Medien.

Die Leistung des Papers bestand darin, die Semantik für Verhalten und Events so zu definieren, dass mit einer Reihe definierter Operatoren eine implementierungsunabhängige Korrektheitsüberprüfung möglich wird. Darauf aufbauend gab es eine Vielzahl von Forschungsaktivitäten, die das Verständnis rund um Functional Reactive Programming vertieften.

Sich verbessernde Werte

Eine bereits im ersten Paper enthaltene Dimension ist die Zeit. Die Entkopplung des Animations- und Reaktionsmodells von dessen Darstellung ermöglicht unterschiedliche Auflösungen (10, 25 oder 60 Bilder pro Sek. oder Ähnliches), ohne das Modell anpassen zu müssen.

Zustandsänderungen im Modell sind losgelöst von Präsentationen. Betrachtet man nun die Präsentation als Funktion in Abhängigkeit von Zuständen und der Zeit, ändern sich für eine Variable die Werte über die Zeit.

Angenommen, wir nehmen als Beispiel das Laden und Präsentieren eines großen XML-Dokuments. Abhängig von dem Zeitpunkt ($t \varepsilon [\geq t_0, \geq t_1, \geq t_2, \geq t_3, \geq t_4]$) sieht der Client unterschiedliche Werte:

- In t_0 haben wir das Dokument bereits geöffnet.
- In t_1 haben wir die ersten 2 KB des Dokuments bereits gelesen.
- In t_2 haben wir die ersten 4 KB des Dokuments bereits gelesen.
- In t_3 haben wir bereits die ersten 5 Kind-Knoten der Dokument-Wurzel vollständig gelesen.
- In t_4 haben wir das Dokument vollständig gelesen und geparst.

Diese Werte können wir als Liste der sich verbessernden Werte zu einem bestimmten Zeitpunkt auffassen. Und für solche Listen haben wir in dieser Kolumne bereits mehrere Konstrukte und Frameworks kennengelernt, mit denen wir sie auch auswerten können. Ob das Interface Iterable oder Iterator, die Bibliothek TotallyLazy oder eine Programmiersprache wie Clojure: Sie iterieren typischerweise über Werte-Mengen, -Listen oder -Sammlungen, ziehen die entsprechenden Werte aus der Liste für die Verarbeitung heraus und rufen dafür einen entsprechenden Block oder eine Funktion auf.

Diese Aufrufrichtung dreht Reactive Extensions um.

Der nächste Schritt: Reactive Extensions

Reactive Extensions – kurz: Rx – transportieren die Ideen des (Functional) Reactive Programming in die OO-Welt. Ursprünglich im C#-Umfeld entstanden, basiert die Kernidee darauf, dass neben den Daten haltenden Klassen (wie z. B. JavaBeans, Entitäten, Transferobjekte oder Ähnliche) auch die Collection-Klassen selbst beobachtbar sind.

Anstatt ständig über Wertmengen (Collections, Listen o. Ä.) zu iterieren, lassen wir einfach unseren Beobachter-Code von einem Observable aufrufen. Anstatt also Werte aus einem Iterable so auszulesen

```
for ( String s : meineStringListe ) {
  // mach etwas Schlaues mit jedem S
}
```

schreiben wir für ein entsprechendes Observable, das die Werte aus **meineStringListe** kapselt:

```
meinStringObservable.subscribe( new Action1() {
  @Override
  public void call(String s) {
    // mach etwas Schlaues mit jedem S
  }
});
```

Dabei kapselt die anonyme innere – von **Action1**-spezialisierte – Klasse die Funktionalität in der Methode **call()**, die zuvor im Block der foreach-Schleife stand. Sowohl die Laufzeitkomplexität als auch die Komplexität beim Schreiben des Codes sind ähnlich.

Für die folgenden Beispiele verwenden wir ein konkretes quelloffenes Framework: RxJava.

Netflix/RxJava

RxJava wurde von Netflix entwickelt und dort laut Angaben der Entwickler produktiv verwendet. Das Framework ist an das Vorbild aus der .NET-Welt angelehnt und bietet ein Fluent-API für die Erzeugung von Datenflussmodellen (s. [RxJava]).

RxJava implementiert in der Klasse **Observable** eine Reihe statischer Methoden, die für das Erzeugen und Komponieren von Observables mit Observern geeignet sind. Generell lassen sich viele Elemente der Standard-Java-Bibliothek wie Iterables, Arrays, gewöhnliche Objekte und Futures mit Hilfe der statischen Erzeugungsmethoden von Observable einfach als Observable kapseln.

Dies geschieht beispielsweise durch Aufrufe wie den Folgenden:

```
Observable<Integer> numbers = Observable.range(10, 100);
Observable<String> people = Observable.from("Tilkov",
  "Starke", "Ghadir");
Observable<List<String>> peopleList =
  Observable.just(Arrays.asList("Tilkov", "Starke", "Ghadir") );
Observable<String> incoming =
  Observable.create( new ErzeugendeFunktion() );
```

Die letzten beiden Aufrufe sind bemerkenswert: Die Methode **just()** erzeugt ein Observable, das die gesamte Liste als einen Wert beobachtet, während mit **create()** eine Funktion verwendet wird, die gemäß Observable/Observer-Kontrakt Werte erzeugt und an sich registrierende Observer übermittelt.

Eine weitere Gruppe von statischen Methoden bilden die typischen Higher-Order-Functions, die wir in dieser Kolumne bereits häufiger gesehen haben, wie filter, map, flatMap, reduce. Darüber hinaus gibt es mehrere subscribe()-Methoden, mit denen sich Observer registrieren können.

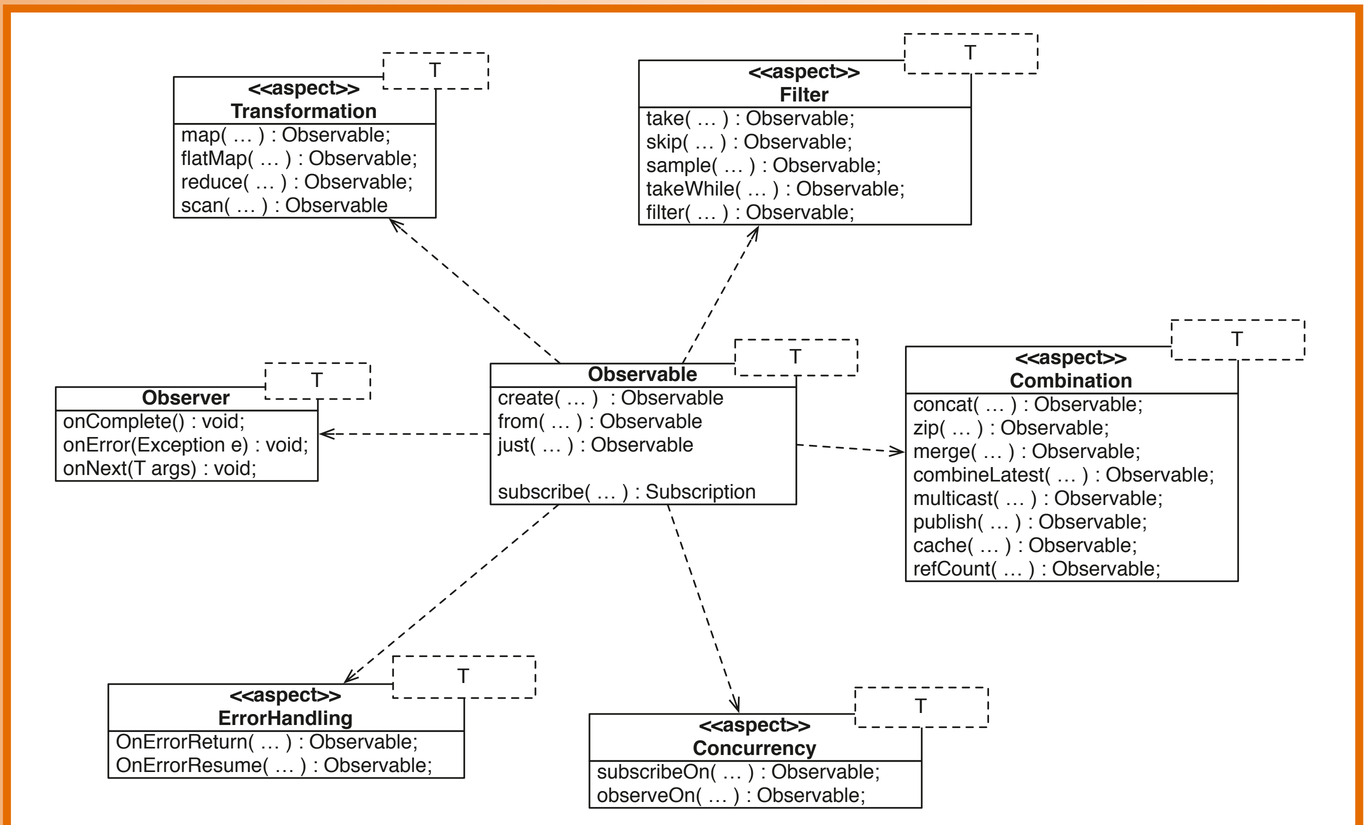


Abb. 1: Logisches Modell der Klasse Observable - In Wirklichkeit umfasst Observable alle Aspekte und implementiert die Methoden selbst

Mit Hilfe der nächsten Gruppe von statischen Methoden lassen sich Observables zu neuen Observables kombinieren. Mit Hilfe dieser Kombinatoren können sehr einfach komplexe Datenflüsse zusammengesteckt werden.

Abbildung 1 skizziert die logische Struktur der Klasse `Observable`. In der tatsächlichen Implementierung des Frameworks sind die Methoden vielfach überladen und erlauben eine Vielzahl unterschiedlicher Parametertypen.

Da die Bibliothek dem Vorbild aus der .NET-Welt sehr ähnlich ist, sind die Video-Tutorials auch für eine Umsetzung in Java sehr hilfreich. Eine Übersicht über die Video-Tutorials gibt es hier [msdnrx].

Je nach Framework ist die Schnittstelle für die Observer im Vergleich zum GoF-Muster [GoF95] erweitert. Ein Observer wird bei Reactive Extensions nicht nur über die Zustandsänderung informiert, sondern auch über Verarbeitungsfehler. Dies ist insbesondere für die asynchrone Verarbeitung sehr nützlich, weil so Verarbeitungsfehler eindeutig transportiert werden können.

Wer hat's erfunden?

Die Forschung zum *Functional Reactive Programming* wurde im Prinzip 1997 durch das FRAN-Paper (Functional Reactive ANimation) von Conal Elliot von Microsoft und Paul Hudak von der Yale University [ElHu97] losgetreten. Es gibt in der funktionalen Welt keine wirklich effizienten Implementierungen, die ein reines Push-Modell verwenden. Jüngere Ansätze kombinieren Push und Pull, um so effizientere Implementierungen in funktionalen Sprachen zu ermöglichen.

Reactive Extensions wurden später im Wesentlichen von Eric Meijer – damals ebenfalls Microsoft – entwickelt. Sie werden gerade stark von Microsoft propagiert.

Heute gibt es Ansätze in verschiedenen Programmiersprachen; unter anderem in .NET, JavaScript und Java. Es gibt mehrere Frameworks in unterschiedlichen Reifegraden und Schwerpunkten.

Datenfluss und Fehlerbehandlung

Nachdem das vorige Beispiel auf „Hello World"-Niveau mit Reactive Extensions genau so komplex wie die normale Implementierung mit Iterator ist, sollten wir uns ein komplexeres Beispiel anschauen. Der folgende Code zeigt die Definition der statischen Struktur eines Datenfluss-Automaten:

```
Observable<String> tags = Observable.from("innoQ",
  "Softwarearchitektur", "hidden product placement");
Observable<String> people =
  Observable.from("Tilkov", "Starke", "Ghadir");

Observable derDatenflussAutomat =
Observable
  .merge( tags, people, fetchDocuments() )
  .onErrorReturn(new Func1<Throwable, String>() {
    @Override
    public String call(Throwable s) {
      return "Actually aborted during stuff... " + s ;
    }
  }).finallyDo(new Action0() {
    @Override
    public void call() {
     System.out.println("MainMergeMapAndFilter.call -- finally done!");
  }
});
```

Dieser komplexere Datenfluss-Automaten `derDatenflussAutomat` hängt die Daten von verschiedenen Quellen per `merge()` aneinander. Die Methode `fetchDocuments()` liefert selbst einen `Observer<String>` zurück, sodass dessen Werte genauso angefügt werden, selbst, wenn sie von einem externen System ermittelt werden müssen. Der Datenfluss-Automat kommt ohne try-catch-finally aus, legt aber eine ähnliche Semantik an den Tag. Tritt während der Verarbeitung des Observables eine Exception auf, wird für diese die anonyme Fehlerbehandlungsfunktion aufgerufen, die für ein Throwable eine entsprechende Meldung zurückliefert. Eine solche Fehlermeldung wird ebenfalls in dem Observable dann als Datum gesehen und darüber werden dann Beobachter informiert.

Am Ende der Verarbeitung aller Elemente wird die Action ausgeführt, die einfach nur das Ergebnis ausgibt. Allerdings wird die Verarbeitung noch gar nicht angestoßen. Der obige Code zeigt noch keinerlei Wirkung, sondern definiert erst einmal die Struktur der am Datenfluss beteiligten Komponenten. Erst, wenn ein Subscriber am Ergebnis interessiert ist (also wenn er sich per `subscribe()` registriert), fließen die Daten.

Wird im Anschluss an die obigen Anweisungen das Folgende aufgerufen:

```
derDatenflussAutomat.subscribe( new TraceObserver() );
```

wird der `TraceObserver` mit den Daten benachrichtigt.

Interessant wird es, wenn man die im Beobachter-Muster genannten Subjekte genauer betrachten möchte.

Subjekte

Eine spezielle Form der Observables sind die Subjekte. Sie leiten von Observable ab und implementieren gleichzeitig noch die Observer-Schnittstelle.

Sich verbessernde Werte können wir beispielsweise mit Hilfe eines `BehaviorSubject<T>` realisieren. Ein `BehaviorSubject` benachrichtigt Beobachter bei dessen Registrierung über den aktuell gültigen Wert des Subjekts und anschließend über jede weitere Wertänderung. Genau dieses Verhalten wünscht man sich, wenn man im Spreadsheet für eine Zelle die Formel „= A1 + B1" hinterlegt.

Also können wir in Java schreiben:

```
BehaviorSubject<Integer> a1 =
  BehaviorSubject.createWithDefaultValue(0);
BehaviorSubject<Integer> b1 =
  BehaviorSubject.createWithDefaultValue(0);
Observable<Integer> zelle =
  Observable.combineLatest( a1, b1, new Adder() );
```

Dabei ist `Adder` die selbst definierte binäre Additionsfunktion, die die von RxJava definierte Schnittstelle für binäre Funktionen realisiert:

```
public class Adder implements Func2<Integer, Integer, Integer> {
  @Override
  public Integer call(Integer i1, Integer i2) {
    return i1 + i2;
  }
}
```

Jetzt reicht ein Horchen auf der Zelle und man bekommt stets den korrekten Wert mit. Damit das auch innerhalb eines simplen Testprogramms geht, verwende ich dafür direkt einen RxJava-Scheduler, um nebenläufig Daten zu manipulieren und mich darüber benachrichtigen zu lassen.

Datenänderungen im anderen Thread

In diesem Fall, in dem ich in einem Test-Programm nur beobachten will, welche Werte die Zelle (`zelle`) annimmt, lasse ich das Rx-Framework einen separaten Thread dafür verwenden.

Ich füge also an die obige Deklaration der lokalen Variablen `a1`, `b2` und `zelle` in derselben Methode an:

```
zelle.observeOn( Schedulers.newThread()).subscribe(
  new TraceObserver( "zelle: " ) );
a1.onNext( 1 );
b1.onNext( 5 );
a1.onNext( 2 );
a1.onNext( 3 );
```

Dabei ist der `TraceObserver` eine selbstgeschriebene Klasse, die jeden Aufruf der Methoden `onNext()`, `onError()` oder `onComplete()` protokolliert. Die Methode `observeOn( Scheduler )` bekommt einen speziellen Scheduler, um darüber die Benachrichtigung im separaten Thread abzusetzen. Dazu später mehr.

Die Ausgabe des Programms überrascht nicht. Es ist lediglich bemerkenswert, dass bereits direkt bei der Registrierung, bevor die Werte von `a1` und `b1` manipuliert werden, der Beobachter über den aktuellen Zustand der Zelle benachrichtigt wird.

```
zelle: TraceObserver.onNext -- 0
zelle: TraceObserver.onNext -- 1
zelle: TraceObserver.onNext -- 6
zelle: TraceObserver.onNext -- 7
zelle: TraceObserver.onNext -- 8
```

Asynchronität und Zeitbezug in Rx

Damit nicht synchron bereits beim Registrieren (per `subscribe()`) alle Observer aufgerufen werden, erlaubt das Rx-Framework eine asynchrone Kopplung von Observable und Observern.

Dazu bietet das Framework sogenannte Scheduler, die über den zeitlichen Verlauf und die Zuordnung zu Threads entscheiden. Im Wesentlichen enthält die abstrakte Klasse `Scheduler` Methoden, um Aufgaben zeitlich festzulegen oder die aus Sicht des Schedulers aktuelle Zeit abzufragen.

RxJava bringt verschiedene konkrete Implementierungen mit, die der API-Dokumentation entnommen werden können. Alternativ zur gezeigten Implementierung, in der die Observer durch einen zusätzlichen Thread benachrichtigt werden, kann beispielsweise über einen ExecutorScheduler ein Standard-ExecutorService und damit ein JVM-interner Timer verwendet werden. Interessant ist vielleicht auch der TestScheduler, über den die Zeit „vorgespult" werden kann, um so Ereignisse über größere Zeiträume in kurzen Intervallen zu testen.

Fazit

Reactive Extensions basieren im Wesentlichen auf der einheitlichen Abstraktion von Einzelwerten und Collections zu einem Observable, auf dem einerseits Funktionen zum Erzeugen, Filtern, Transformieren und Kombinieren vorhanden sind, sich aber andererseits auch Funktionen zur asynchronen Verarbeitung und Fehlerbehandlung befinden. Observables können mit einem erweiterten Observer-Muster beobachtet werden, bei dem die Beobachter nicht nur über Änderungen von Einzelwerten informiert werden, sondern auch am Ende der Verarbeitung aller Werte eines Observables oder im Falle eines Fehlers während der Verarbeitung.

Es gibt noch keine Version 1.0 von RxJava, dennoch soll die Bibliothek bei Netflix bereits längere Zeit produktiv im Einsatz sein. Die Komposition von Funktionen erinnert sehr stark an die Möglichkeiten, die man in funktionalen Programmiersprachen mit den Higher-Order-Functions hat.

Allerdings ermöglicht die Invertierung der Aufrufabhängigkeit entgegen des üblichen Datenflusses die Konstruktion von Services, bei denen die Komplexität über die Datenbeschaffung, -synchronisierung und -traversierung vor den Clients verborgen werden kann.

Eine dazu interessante alternative Perspektive bringt vielleicht [MaOd12] rein, die mit Scala.React das Observer-Muster überflüssig machen wollen. Aber das ist eine ganz andere Geschichte.

Literatur und Links

[Amsd11] E. Amsden, A Survey of Functional Reactive Programming, Rochester Institute of Technology, Independent Study on FRP, 2011

[ChHu13] B. Christensen, J. Husain, Functional Reactive in the Netflix API with RxJava, 4.2.2013,
http://techblog.netflix.com/2013/02/rxjava-netflix-api.html

[ElHu97] C. Elliot, P. Hudak, Functional Reactive Animation, in: ICFP '97 Proc. of the 2nd ACM SIGPLAN International Conference on Functional Programming

[Frappe] A. Courtney, Frappé: Function Reactive Programming in Java, in: Proc. PADL '01, s. a.:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.4772&rep=rep1&type=pdf

[FRP] http://en.wikipedia.org/wiki/Functional_reactive_programming

[Ghad11] P. Ghadir, Domänenmodelle und Interaktion, in: JavaSPEKTRUM, 5/2011, s. a.:
http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2011/05/ghadir_JS_05_11.pdf

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1995

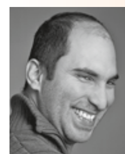
[MaOd12] I. Maier, M. Odersky, Deprecating the Observer Pattern with Scala.React, EPFL-Report-176887

[msdnrx] MSDN Video Tutorials zu Rx,
http://www.microsoft.com/germany/MSDN/webcasts/library.aspx?id=1032487439

[RP] http://en.wikipedia.org/wiki/Reactive_programming

[Rx] Reactive Extensions, https://rx.codeplex.com/

[RxJava] https://github.com/Netflix/RxJava



Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com