Neuer reaktiver Ansatz für die GUI-Programmierung mit Sodium

Sven Reinck, FLUXparticle

In jüngster Zeit ist der Begriff „reaktive Programmierung“ in aller Munde. Es besteht allerdings noch viel Uneinigkeit darüber, was reaktive Programmierung überhaupt ist. Der Artikel bringt etwas Licht ins Dunkel, denn der Autor ist der Meinung, dass dieses Paradigma nach der funktionalen Programmierung das nächste sein wird, das in der breiten Masse Einzug hält.

Neben der reaktiven Programmierung gibt es auch die funktionale reaktive Programmierung (FRP). Auch wenn die Lambdas, die beim reaktiven Framework RxJava zum Einsatz kommen, frei von Nebeneffekten sind und somit als funktional bezeichnet werden könnten, lohnt es sich dennoch, ein funktional reaktives Framework wie Sodium anzuschauen, um nochmal ein gänzlich anderes Konzept kennenzulernen, das auch seine Anwendungsfelder hat.

RxJava ist zwar für die Verarbeitung von Events in einer GUI interessant, allerdings ist hier das Konzept von Sodium besser. Deshalb stellt der Autor seine Lösung für die ersten fünf GUIs des 7GUIs-Benchmarks von Eugen Kiss *(siehe „https://eugenkiss.github.io/7guis/tasks“)* vor, die er mit seinem Groovy-Wrapper für Sodium *(siehe „https://github.com/SodiumFRP/sodium“)* namens „Fenja“ *(siehe „https://github.com/FLUXparticle/fenja“)* erstellt hat. Im 7GUIs-Benchmark geht es darum, sieben typische Aufgaben der GUI-Programmierung in einem bestimmten Framework zu lösen, um anschließend verschiedene Frameworks hinsichtlich ihrer Funktionalität und Effektivität vergleichen zu können.

Zunächst ein Blick darauf, was „Functional Reactive Programming“ mit Sodium von reaktiver Programmierung mit RxJava unterscheidet. In RxJava gibt es sogenannte „Observables“, die einen oder mehrere Werte ausgeben können; ähnlich zu den Streams in Java 8. Diese können dann gefiltert, gemappt oder sonst wie verarbeitet werden. Zum Beispiel lassen sich mithilfe dieser Observables die Anfragen an einen REST-Service sehr bequem und mit nur wenigen Zeilen parallelisieren, und zwar besser als es mit Streams möglich wäre.

In einer GUI ist die Situation allerdings etwas anders gelagert. Hier gibt es zwei verschiedene Arten von Funktionalitäten, mit denen man umgehen muss. Auf der einen Seite sind da beispielsweise Textfelder, die zu jeder Zeit einen bestimmten Wert beinhalten. Diese sind in Fenja als „Value“ abgebildet *(siehe Abbildung 1)*. Auf der anderen Seite gibt es auch Buttons, die zu einzelnen Zeitpunkten Ereignisse auslösen. Sie kommen in Fenja als „EventStream“ vor *(siehe Abbildung 2)*.

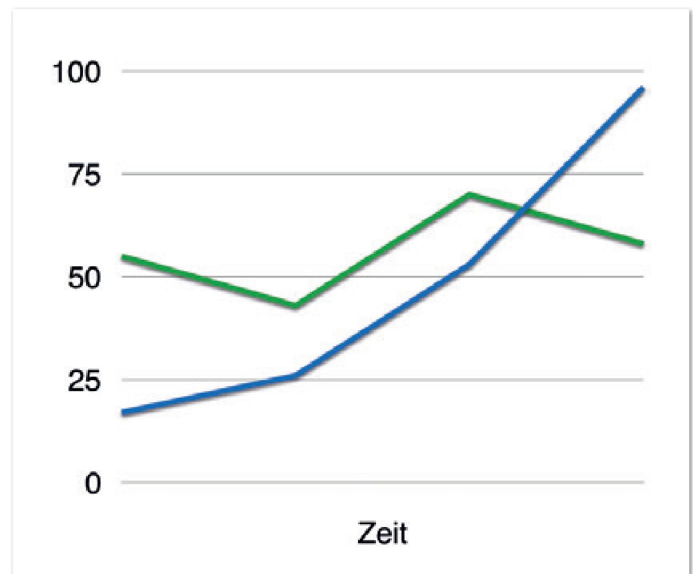

Abbildung 1: Darstellung von Werten, die sich über die Zeit verändern

Grundoperationen

Sowohl „Value“ als auch „EventStream“ können durch „map()“ transformiert werden. Dagegen unterstützt nur „EventStream“ die Operation „filter()“. Würde diese Operation auch auf „Value“ funktionieren, hätte der resultierende „Value“ in bestimmten Fällen keinen Wert, und genau das würde der Definition eines „Value“ widersprechen. Es gibt noch einige weitere Grundoperationen, die jedoch hier nicht wichtig beziehungsweise in den nachfolgenden Beispielen erklärt sind. Durch die Verknüpfung dieser Typen und Operationen entsteht ein Graph, der jeweils bei den Beispielen abgebildet ist.

Erste GUI: Ein Counter

Der Counter ist die einfachste Aufgabe aus dem Benchmark. Er besteht lediglich aus einem Textfeld und einem Button *(siehe Abbildung 3)*. Immer wenn man auf den Button klickt, wird die Zahl im Textfeld um „1“ erhöht.

Der Graph ist auch entsprechend einfach und ermöglicht es damit, erstmal ein paar Konventionen für die Grafiken einzuführen. Die „EventStreams“ sind als dicke, blaue Pfeile dargestellt und ihre Bezeichnung beginnt mit einem kleinen „s“. Der Button erzeugt einen solchen „EventStream“, der bei jedem Klick ein Event auslöst. Aus

Abbildung 2: Darstellung von Ereignissen zu unterschiedlichen Zeitpunkten

diesem „EventStream" und einem „Value" entsteht eine Schleife, die bei jedem Event den Wert des „Value" nimmt, um „1" erhöht und wieder in das „Value" zurückschreibt.

So eine Schleife würde bei einem herkömmlichen reaktiven System nicht funktionieren, da das Erhöhen des Zählers ja sofort wieder die Schleife triggern und somit der Zähler unendlich oft erhöht werden würde. Sodium verwendet dafür Transaktionen; jede Veränderung, die durch ein Ereignis ausgelöst wurde, führt erst in der nächsten Transaktion zu einer tatsächlichen Veränderung. Damit sind solche Schleifen ohne Weiteres möglich. Zu guter Letzt wird die Zahl im „vCounter" noch mit „map()" in einen „String" umgewandelt, der in JavaFX ganz einfach an die „textProperty()" des Textfelds gebunden werden kann. *Listing 1* zeigt, wie dieser Graph mithilfe von Fenja in Groovy dargestellt wird.

Alle hier vorgestellten Codes sind nach dem EVA-Prinzip unterteilt, um mehr Übersicht zu erreichen. Der erste Teil ist die Eingabe, bei der mithilfe der Convenience-Methode „streamOf()" ein „EventStream" aus dem „ActionEvent" des Buttons erzeugt wird. Im dritten Teil, der Ausgabe, wird mithilfe des überladenen Operators „<<" (die Idee stammt aus C++) der Counter an „textProperty()" des Textfelds gebunden.

Das Interessanteste ist der mittlere Teil; dort findet die eigentliche Verarbeitung statt – hier ist auch die Schleife zu erkennen –, denn die Variable „vCount" kommt dort vor, bevor sie überhaupt erzeugt wurde. Das würde normalerweise gar nicht gehen – hier kommt jetzt der Grund zum Tragen, aus dem der Autor überhaupt Groovy verwendet hat, um einen Wrapper für Sodium zu bauen.

Da Schleifen in FRP-Graphen etwas ganz Normales sind, hat Sodium dafür extra Datentypen geschaffen, nämlich eine „EventStreamLoop" und eine „ValueLoop". Diese dienen als Platzhalter für die echten Datentypen und werden erst später mit einem Wert belegt. Hier bietet Groovy die Möglichkeit, definieren zu können, was passieren soll, wenn eine Variable benutzt wird, die es noch gar nicht gibt. In dem Fall kommt einfach ein Loop-Objekt statt eines richtigen Ob-

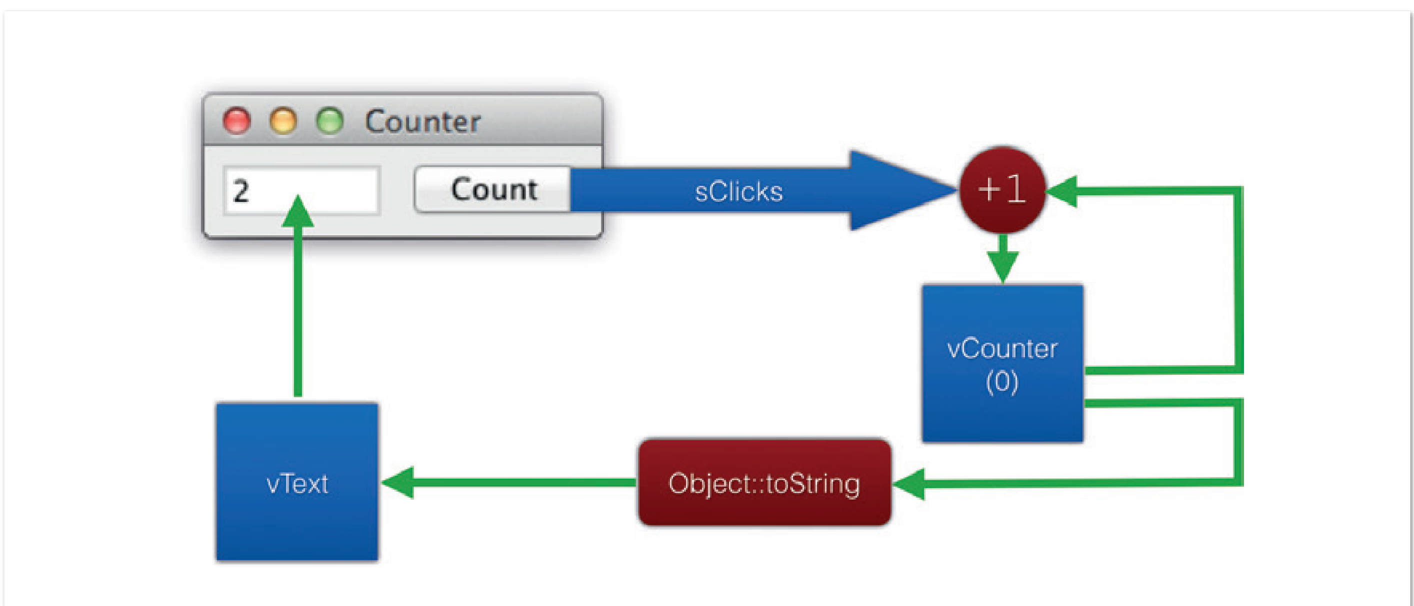

Abbildung 3: Darstellung des Graphen für den Counter

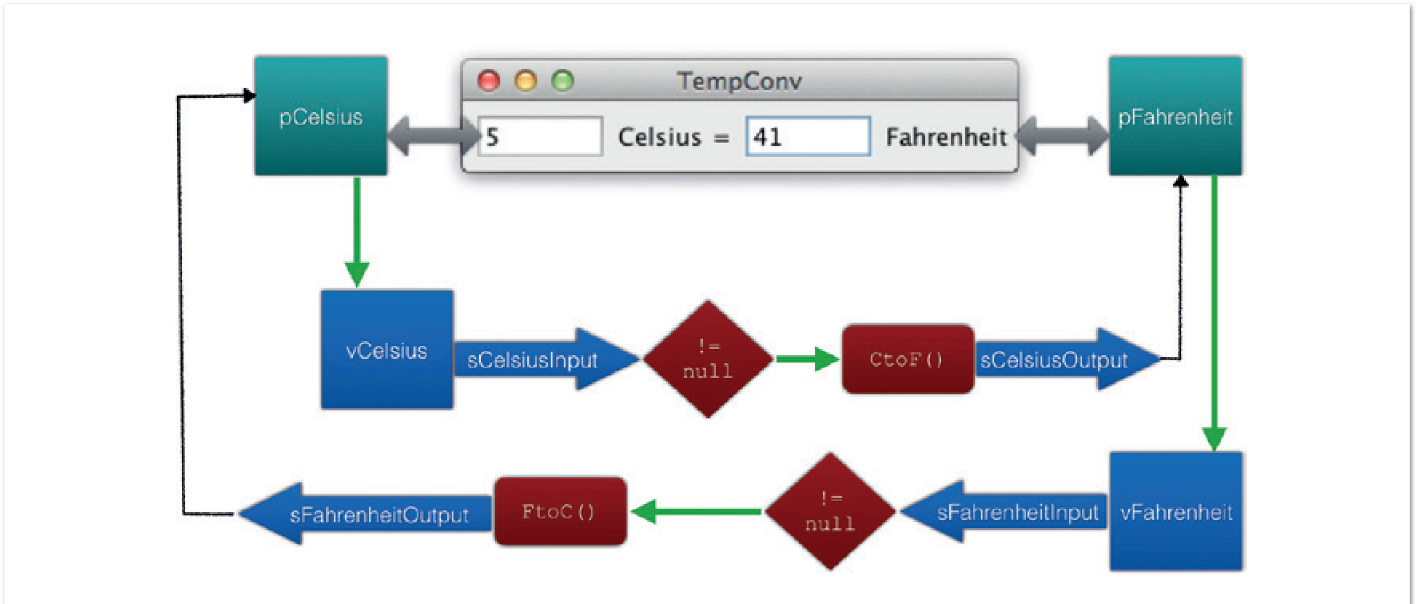

Abbildung 4: Darstellung des Graphen für die Temperatur-Umrechnung

jekts zurück, sodass die Loop später geschlossen werden kann. Die Anfangsbuchstaben „v“ und „s“ sind entscheidend, weil sonst nicht klar ist, ob eine „EventStreamLoop“ oder eine „ValueLoop“ erzeugt werden soll.

```
sClick = streamOf(btCountUp, ActionEvent.ACTION)
// -----
sNextCount = sClick.snapshot(vCount).map { count ->
count + 1 }
vCount = sNextCount.hold(0)
// -----
tfCount.textProperty() << vCount.
map(Integer.&toString)
```

Listing 1

Zweite GUI: Temperatur-Umrechnung

Diese Aufgabe hat bestimmt jeder schon einmal in einer Lehrveranstaltung programmieren müssen. Es soll eine Temperatur von Celsius in Fahrenheit oder umgekehrt berechnet werden *(siehe Abbildung 4)*. Das Besondere an dieser Version ist, dass die Berechnung sofort beim Eintippen passiert, was einen vor eine Herausforderung stellt, wenn das programmseitige Schreiben in ein Textfeld ebenfalls als Eintippen interpretiert wird, wie es bei JavaFX der Fall ist.

Auch hier lässt einen Sodium nicht im Stich. Diese Art von Schleife wird nämlich erkannt und erzeugt eine „Exception“, für die der Autor in seinem Groovy-Wrapper Fenja eine Methode gebaut hat, um sie zu ignorieren. Das Verhalten ist dann nämlich das Gewünschte, dass sofort beim Entdecken der Schleife die Änderungen nicht weiter propagiert werden. Es entsteht somit genau ein Rundgang und es wird nicht versucht, den umgerechneten Wert nochmal zurückzurechnen. *Listing 2* zeigt den Code dazu in Groovy.

```
sCelsiusInput     =  valueOfSilent(pCelsius).values();
sFahrenheitInput  =  valueOfSilent(pFahrenheit).values();
// -----
sFahrenheitOutput = sCelsiusInput
        .filter { it != null }
        .map { cToF(it.doubleValue()) };
sCelsiusOutput = sFahrenheitInput
        .filter { it != null }
        .map { fToC(it.doubleValue()) };
// -----
sFahrenheitOutput.listen(pFahrenheit.&setValue)
sCelsiusOutput.listen(pCelsius.&setValue)
```

Listing 2

Bei der Eingabe kommt die erwähnte Methode „valueOfSilent()“ vor, die im Fall einer ungewollten Schleife den Fehler unterdrückt und die Schleife einfach nur durchbricht. Bei der Ausgabe wird kein Binding verwendet, da man sonst nichts mehr eingeben könnte. Stattdessen kommt die Methode „listen()“ zum Einsatz, die einfach im Falle eines Events aufgerufen wird und somit den Setter für das Textfeld aufruft.

Dritte GUI: Flight Booker

Der Flight Booker ist von den sieben die Lieblingsaufgabe des Autors. Sie demonstriert nämlich am besten, welches Problem „Functional Reactive Programming“ löst und wie man damit typische Fehler der GUI-Programmierung von vornherein unterbindet. Er hat sie deshalb auch schon in vielen seiner Trainings eingesetzt.

Die Aufgabe ist leicht erklärt. Der Dialog besteht aus einer ComboBox für die Auswahl, ob man nur Hinflug oder Hin- und Rückflug buchen möchte. Darunter sind zwei Textfelder, in die das Hin- beziehungsweise Rückreisedatum eingetragen wird. Das Textfeld für die Rückreise soll natürlich deaktiviert sein, wenn nur ein Hinflug gebucht wird. Das ist sicherlich noch leicht zu lösen, doch etwas schwieriger ist der „Buchen“-Button. Dieser soll genau dann aktiviert sein, wenn die Buchung so gültig wäre, die notwendigen Daten also gültig sind und bei der Buchung eines Rückflugs dieser zeitlich nicht vor dem Hinflug liegt *(siehe Abbildung 5)*.

Dieser Sachverhalt ist zwar auch mit herkömmlichen Mitteln recht einfach zu programmieren, doch tritt hier häufig der Fehler des „First Update“ auf. Wer hat nicht schon mal einen Bug-Report auf den Tisch bekommen, der etwa so klang: „Wenn ich den Dialog das

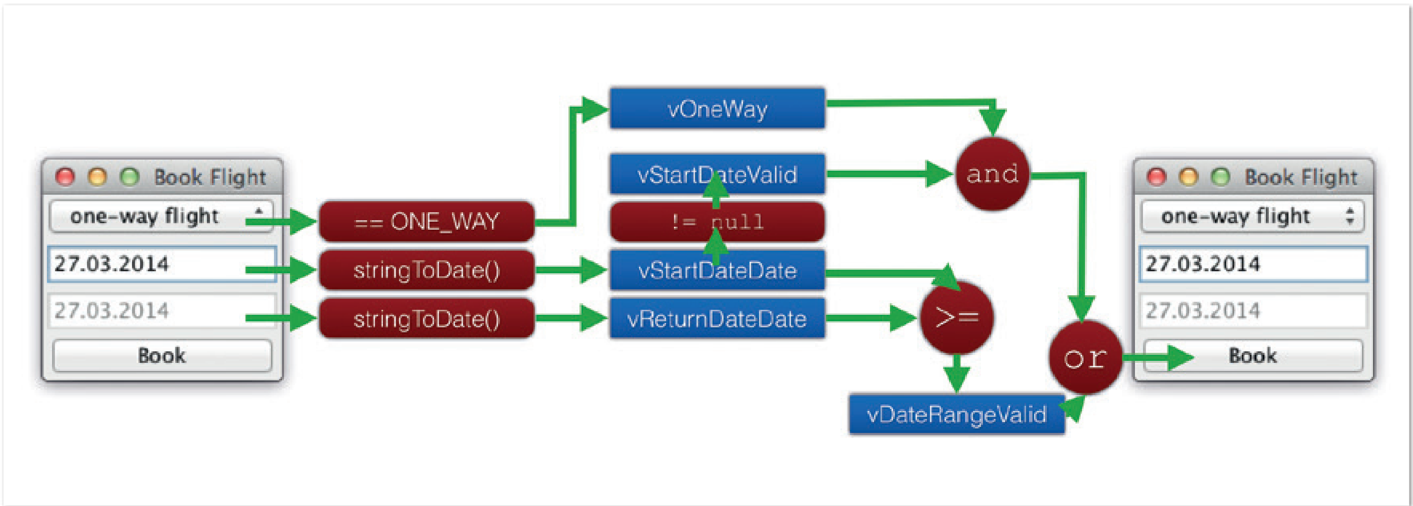

Abbildung 5: Darstellung des Graphen für den Flight Booker

```
vOneWay = vFlightType.map { v -> v == ONE_WAY_FLIGHT }

vStartDateAsDate = vStartDate.map { txt -> isDateString(txt)
                                        ? stringToDate(txt)
                                        : null }
vReturnDateAsDate = vReturnDate.map { txt -> isDateString(txt)
                                          ? stringToDate(txt)
                                          : null }

vStartDateIsValid = vStartDateAsDate.map { v -> v != null }
vReturnDateIsValid = vReturnDateAsDate.map { v -> v != null }

vDateRangeIsValid = (vStartDateAsDate**vReturnDateAsDate)
      { s, r -> s != null && r != null && s.compareTo(r) <= 0 }

vDatesValid = (vOneWay**vStartDateIsValid**vDateRangeIsValid)
              { o, s, r -> (o && s) || r }
```

Listing 3

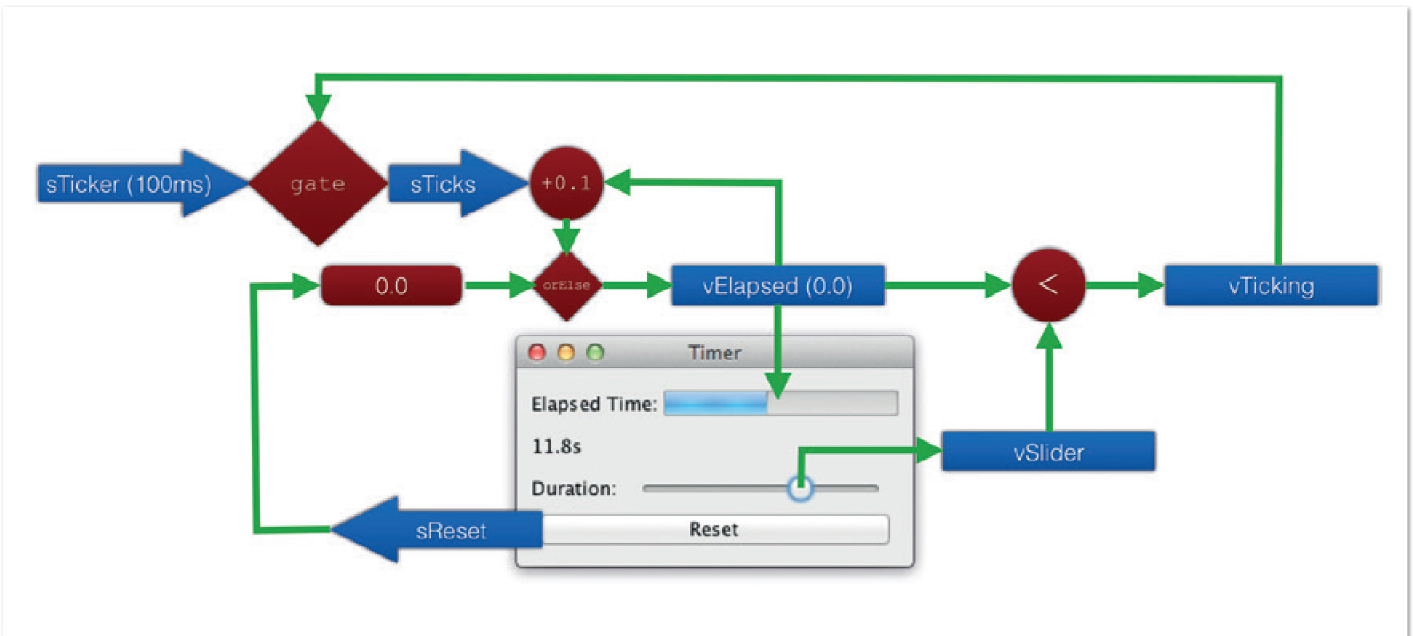

Abbildung 6: Darstellung des Graphen für die Timer-Aufgabe

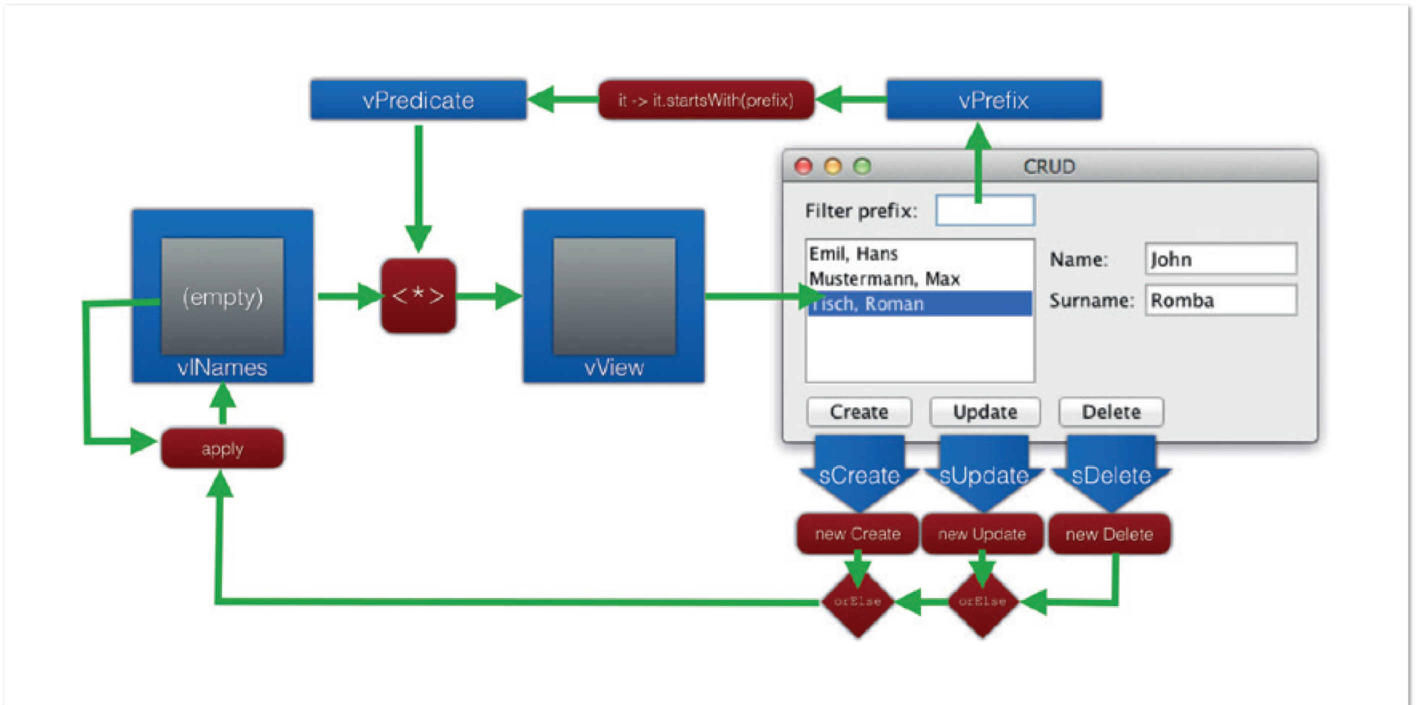

Abbildung 7: Darstellung des Graphen für die CRUD-Aufgabe

erste Mal starte, ist der „Buchen"-Button deaktiviert, obwohl alle Daten richtig sind. Ich muss erst das Datum verändern, damit der Button aktiv wird." Das liegt daran, dass die Berechnung, ob der Button aktiv ist, bei der herkömmlichen Methode oft nur bei Änderungen stattfindet und der Button am Anfang einfach fest aktiviert oder deaktiviert wird. Spätestens wenn sich am Initialisierungsprozess etwas ändert, könnte man sich für die falsche Option entschieden haben und es kommt zum beschriebenen Bug-Report.

Bei FRP kann dieser Fehler nicht passieren, da einmal dauerhaft definiert ist, unter welchen Bedingungen der Button aktiviert ist. Je nachdem, ob diese am Anfang vorherrschen, wird der Button aktiviert oder eben nicht. Für diese GUI reicht der Verarbeitungsteil als Code *(siehe Listing 3)*. Hier kommt wieder die Möglichkeit vor, Groovy-Operatoren zu überladen, um mithilfe des „*"-Operators mehrere „Values" zu kombinieren.

Vierte GUI: Timer

In der vierten Aufgabe soll getestet werden, wie gut zeitabhängige Events funktionieren. Hier hat der Autor mithilfe von JavaFX einen „EventStream" erzeugt, der alle hundert Millisekunden feuert. Dieser läuft durch einen „filter()", der nur dann die Events durchlässt, wenn die „ProgressBar" ihren durch den Slider vorgesehenen Maximalwert noch nicht erreicht hat *(siehe Abbildung 6)*.

Schließlich ist da noch der Reset-Button, der einfach ein „EventStream" erzeugt, das mit dem gefilterten Timer-„EventStream" kombiniert wird. Hier kommt die Funktion „orElse()" zum Einsatz, die zwei „EventStreams" auf eine Weise kombiniert, dass ihr Verhalten immer deterministisch ist. Hier tut sich Sodium wieder mit seinen Transaktionen hervor. Treten nämlich innerhalb der gleichen Transaktion zwei Events auf, die in den gleichen „EventStream" fließen sollen, ist genau festgelegt, welches Event durchkommt und welches nicht. Bei anderen reaktiven Frameworks kann das Verhalten in so einer Situation durchaus zufällig sein.

Fünfte GUI: CRUD

Bei der CRUD-Aufgabe wird es schon etwas komplizierter. Es soll ein übliches Create-Read-Update-Delete-Interface aufgebaut werden. Hierfür musste man eine spezielle Datenstruktur bauen, die eine reaktive Liste verwalten kann. Diese Listen-Datenstruktur wird mit den Events aus den drei Buttons am unteren Rand gefüttert und befindet sich jeweils in dem grauen Kasten in der Grafik *(siehe Abbildung 7)*.

Diese Datenstruktur ist notwendig, weil es in dieser Aufgabe möglich sein sollte, die Liste live zu filtern, Listen in Sodium jedoch bisher noch nicht vorgesehen sind. Listen sind überhaupt noch ein schwieriges Thema in FRP. Deshalb arbeitet der Autor gerade an einer neuen Version von Fenja, die ohne Sodium auskommt und auch reaktive Listen beinhaltet.

Sven Reinck
sreinck@fluxparticle.de

Sven Reinck hat von 2003 bis 2007 Informatik an der Fachhochschule in Wedel nahe Hamburg studiert und mit Diplom und Master of Computer Science abgeschlossen. Anschließend hat er in verschiedenen Firmen gearbeitet und dabei vor allem in der Spielebranche wertvolle Praxiserfahrung gesammelt. Seit dem Jahr 2012 unterrichtet Sven Reinck als freiberuflicher IT-Trainer und bietet seit dem Jahr 2016 auch eigene Seminare an.