FRP

Maurice Müller

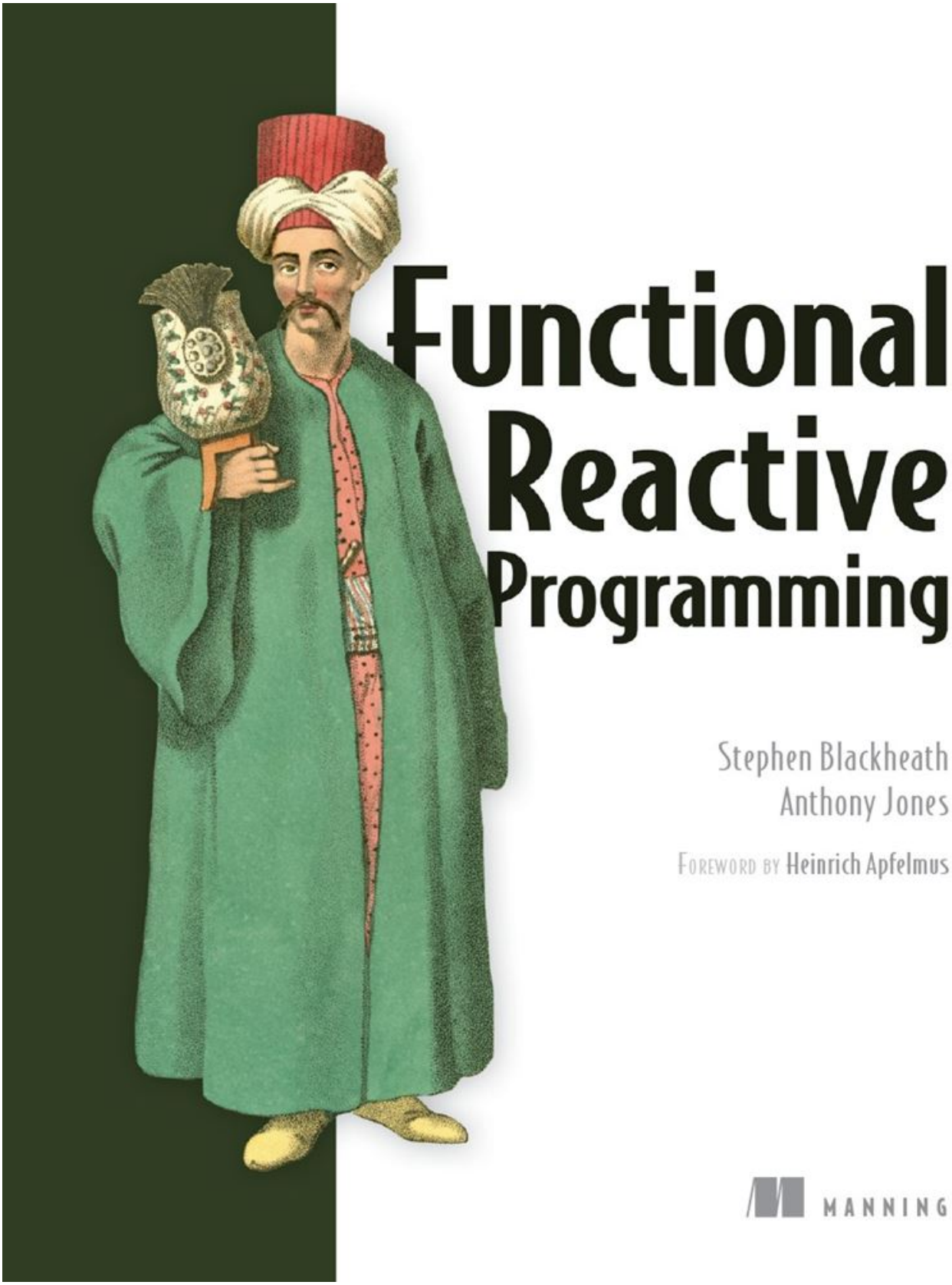
Version {project-version}, 2017-09-29

Was ist FRP?

- FRP
  - Funktionale Reaktive Programmierung
  - functional reactive programming

Definition(en)

Die nachfolgenden Definitionen sind aus *Functional Reactive Programming* von Stephen Blackheath und Anthony Jones:



FRP

"A specific method of reactive programming that enforces the rules of functional programming, particularly the property of compositionality."

Eine bestimmte Methode für die reaktive Programmierung, die die Regeln der funktionalen Programmierung erzwingt, insbesondere die des Kompositionalitätsprinzips.

- Was ist
  - reactive programming?
  - functional programming?
  - compositionality?

Reaktive Programmierung

"A broad term meaning that a programm is 1) event-based, 2) acts in response to input, and 3) is viewed as a flow of data, instead of the traditional flow of control. It doesn't dictate any specific method of achieving these aims. Reactive programmings gives looser coupling between program components, so the code is more modular."

Ein weit gefasster Begriff, der ein Programm beschreibt das 1) event-basiert ist, 2) auf Eingaben reagiert und 3) als ein Fluss von Daten gesehen wird (anstatt des traditionellen Kontrollflusses). Er gibt keine bestimmten Methoden vor diese Ziele zu erreichen. Reaktive Programmierung sorgt für losere Kopplung zwischen Programmteilen und macht somit den Code modularer.

Funktionale Programmierung (FP)

"A style or paradigm of programming based on functions, in the mathematical sense of the word. It deliberatly avoid shared mutable state, so it implies the use of immutable data structures, and it emphasizes compositionality."

Ein Stil oder Paradigma in der Programmierung, der auf Funktionen basiert im mathematischen Sinne. Es wird bewusst ein geteilter veränderbarer Zustand vermieden. Somit müssen implizit nicht veränderbare Datenstrukturen verwendet werden. Zusätzlich betont er Kompositionalität.

FP != !OOP

wenn überhaupt, dann **FP != imperative Programmierung**

- imperativer Stil = Elemente *machen* etwas
  - *machen* impliziert initialen Status, Übergänge und Endstatus
- FP = Komposition von Elementen, die etwas *sind*

Kompositionalität

"The property that the meaning of an expression is determined by the meanings of its parts and the rules used to combine them."

Die Eigenschaft, dass die Bedeutung eines Ausdrucks (eindeutig) bestimmt wird durch die Bedeutung seiner Teile und die Bedeutung der Regeln zur Verknüpfung dieser Teile.

Beispiel: Kompositionalität

- natürliche Sprachen
  - Gottlob Frege war Deutscher.
  - A war B.
  - Der Begründer des Kompositionalitätsprinzip war Deutscher.
- formale Sprachen
  - 1 + 3
  - 1 + (1 + 2)

Man beachte: `1 + 3` **wird *nicht*** zu `4` sondern `1 + 3` **ist** `4` ⇒ Kompositionalität!

Definition des Erfinders von FRP

- Vorhandensein einer formalen Semantik
  - zur Überprüfung der Korrektheit
- Zeitkontinuierlich

Funktionale Programmierung

FP vs Imperative Programmierung (IP)

Quelle: [FPJ-17]

```
if b == 0, return a
else increment a and decrement b
start again with the new a and b
```

klassische imperative Programmierung

- Bedingung wird getestet
- Variablen werden verändert
- es gibt Verzweigung(en)
- es gibt einen Rückgabewert

Obiger Pseudo-Code stellt eine imperative Addition dar.

Problem

- da etwas *gemacht* wird, muss es immer *gemacht* werden
- Nebeneffekte: Variablen (Zustand) wird geändert
- `1 + 3` kann durch `4` ersetzt werden → ist das mit IP auch möglich?
  - ja, falls keine (ungewollten oder gewollten) Nebeneffekte existieren → ansonsten verhält sich das Programm anders

Im obigen Pseudo-Code-Beispiel wird sich das Programm anders verhalten, da es eben den Nebeneffekt des In- bzw. Dekrementierens der Variablen gibt; falls diese an anderer Stelle wieder ausgelesen werden, würden sie bei einer Ersetzung ihren ursprünglichen Wert haben.

FP und Nebeneffekte

FP hat keine (wahrnehmbaren, d.h. äußeren) Nebeneffekte:

- keine Veränderung von Variablen
- keine Ausgabe auf der Konsole oder andere Geräte
- keine Veränderungen in Dateien, Datenbanken, Netzwerken, etc.
- keine Ausnahmen (Exceptions)

→ **funktionale Programme sind eine Komposition von Funktionen, die**

- *ein* Argument entgegen nehmen
- intern Variablen ändern, imperative Sachen machen, etc, aber nie wahrnehmbare Effekte nach außen haben
- *einen* Rückgabewert haben

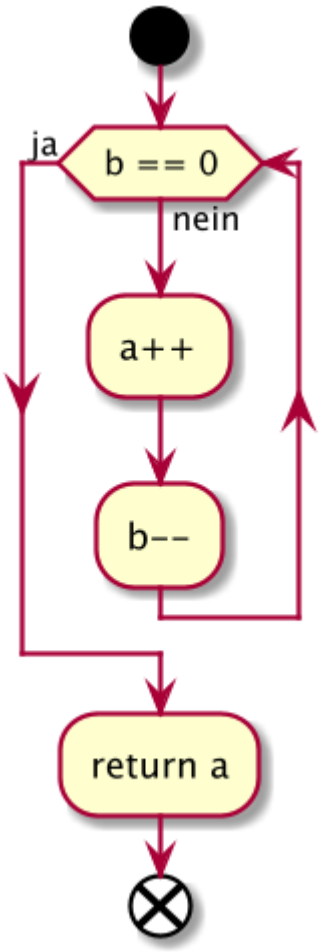
Theorie vs Praxis

- in der Praxis gibt es immer Nebeneffekte
  - Fehler: out-of-memory, stack-overflow, …
  - Beschreiben des Speichers, Thread Starten, …
  - unterschiedliche Dauer bis der Rückgabewert kommt

→ **FP hat keine *gewollten* Seiteneffekte** also keine Seiteneffekte, die Teil des Programms sind

Addition funktional

Aktivitätsdiagramm



Java

```
public static int add(int a, int b) {
  while (b > 0) {
    a++;
    b--;
  }
  return a;
}
```

- voll funktional -> keine äußeren Nebeneffekte
  - keine Exceptions, keine Mutation von externen Variablen, …

Das Java-Programm ist voll funktional, auch wenn es im Fall eines arithmetischen Überlaufs ein falsches Ergebnis liefert. Wichtig ist, dass keine Exception fliegt.

Die beiden Argumente zählen als *ein* Argument: nämlich als ein Paar von Integern.

Division funktional

```
public static int division(int a, int b) {
   return a / b;
}
```

Obiges Beispiel ist nicht funktional: es kann eine Exception werfen (falls b = 0).

Funktional:

```
public static int division(int a, int b) {
   return (int) (a / (float) b);
}
```

Achtung: auch *Logging* widerspricht der funktionalen Programmierung.

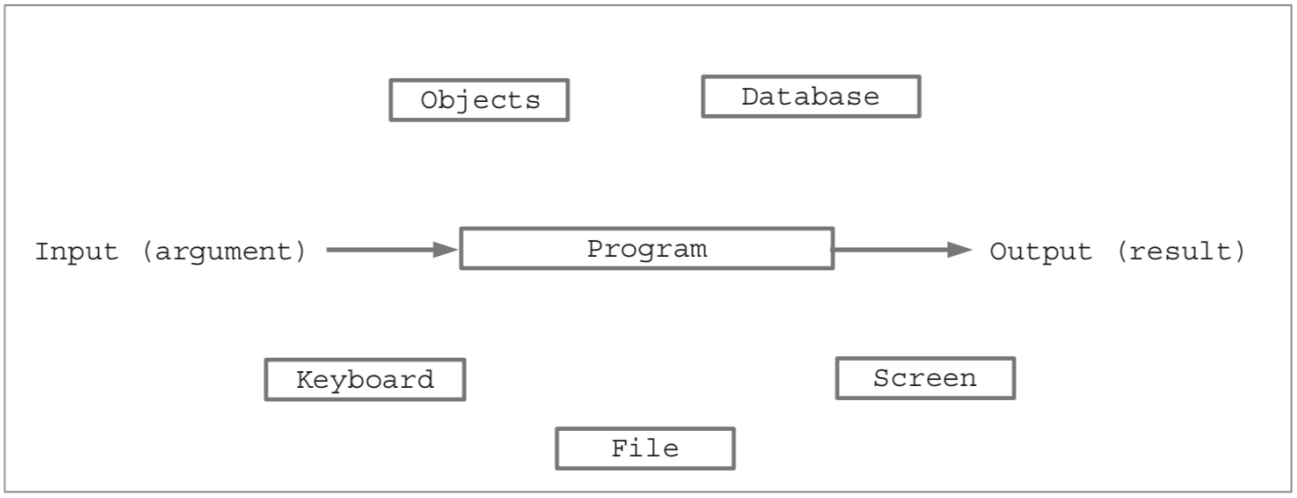
Quelle: [FPJ-17]

```
public static int add(int a, int b) {
  log(String.format("Adding %s and %s", a, b));
  while (b > 0) {
    a++;
    b--; }
  log(String.format("Returning %s", a));
  return a;
}
```

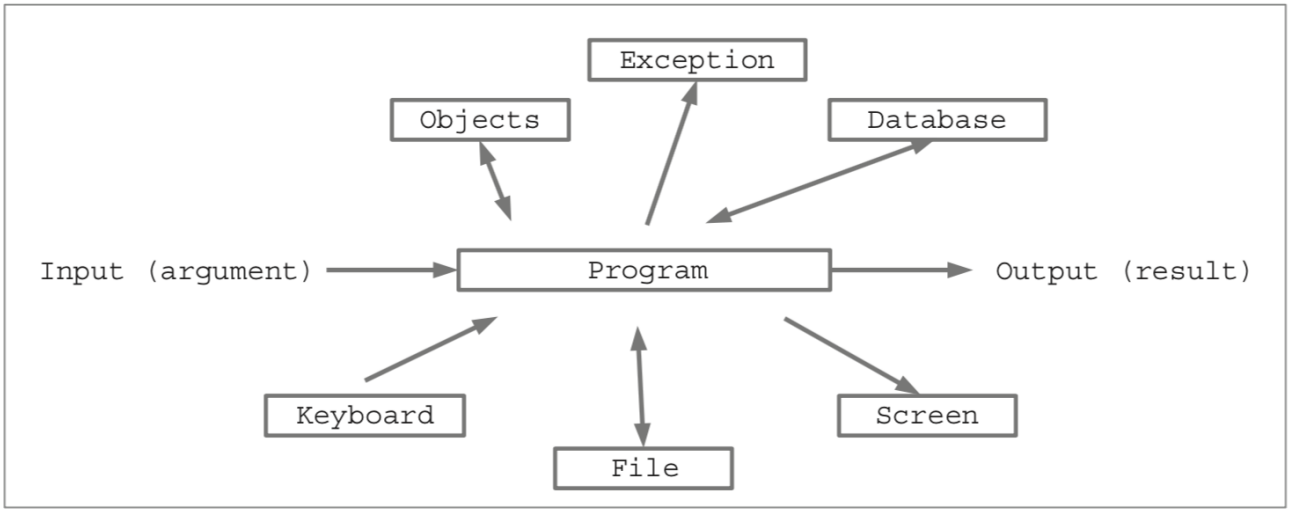
Referenzielle Transparenz

FP ist referentiell transparent

- keine Nebeneffekte
- ein Ausdruck hängt nur von *seiner* Umgebung ab (d.h., nicht von externer Umgebung)
  - kein Lesen von der Konsole / Dateien / ...
- Vorteile
  - wiederverwertbar
  - deterministisch (gleicher Rückgabewert bei gleichem Argument)
  - keine Excepetions (und damit keine Fehlerbehandlung)
  - läuft immer durch (da es von nichts Externem abhängt)



A referentially transparent program doesn't interfere with the outside world apart from taking an argument as input and outputting a result. Its result only depends on its argument.



A program that isn't referentially transparent may read data from or write it to elements in the outside world, log to file, mutate external objects, read from keyboard, print to screen, and so on. Its result is unpredictable.

Figure 1. Quelle: [FPJ-17]

Einfaches Beispiel

- entnommen aus [FPJ-17]
- imperative Programmierung zu funktionaler Programmierung
- Donut kaufen mit Kreditkarte

Imperativ

```
public class DonutShop {
    public static Donut buyDonut(CreditCard creditCard) {
        Donut donut = new Donut();
        creditCard.charge(Donut.price);
        return donut;
    }
}
```

- Kreditkartenzahlung ist ein Nebeneffekt
- schwierig zu testen (durch den Nebeneffekt)
  - Nebeneffekt muss weg
  - → Kartenzahlung als Datenrepräsentation zurückgeben

```
public class Payment {
    public final CreditCard creditCard;
    public final float amount;

    public Payment(CreditCard creditCard, float amount) {
        this.creditCard = creditCard;
        this.amount = amount;
    }
}
```

Problem: Donut muss ebenfalls zurückgegeben werden → extra Klasse oder Tupel zurückgeben

Purchase.java

```
public class Purchase {
    public final Donut donut;
    public final Payment payment;

    public Purchase(Donut donut, Payment
payment) {
        this.donut = donut;
        this.payment = payment;
    }
}
```

Tuple

```
public class Tuple<T, U> {
    public final T _1;
    public final U _2;

    public Tuple(T t, U u) {
        this._1 = t;
        this._2 = u;
    }
}
```

(+) Tuple wird häufig gebraucht und kann an vielen Stellen wiederverwendet werden

(+) Purchase-Klasse ist sprechender

DonutShop mit Tuple

```
public class DonutShop {
    public static Tuple<Donut, Payment> buyDonut(CreditCard creditCard) {
        Donut donut = new Donut();
        Payment payment = new Payment(creditCard, Donut.price);
        return new Tuple<>(donut, payment);
    }
}
```

Vorteile:

- Zahlung ist jetzt entkoppelt
- einfacher zu testen
- Zahlungen können gruppiert werden (da nicht mehr direkt abgebucht wird)

Zahlungen gruppieren

```
public class Payment {
    public final CreditCard creditCard;
    public final float amount;

    public Payment(CreditCard creditCard, float amount) {
        this.creditCard = creditCard;
        this.amount = amount;
    }

    public Payment combine(Payment payment) {
        return new Payment(creditCard, amount + payment.amount);
    }
}
```

DonutShop mit Mehrfach-Zahlung

```
public class DonutShop {
    public static Tuple<List<Donut>, Payment> buyDonuts(final int quantity, final
CreditCard cCard) {
        return new Tuple<>(Collections.nCopies(quantity, new Donut()),
            new Payment(cCard, Donut.price * quantity));
    }
}
```

Test des DonutShops

```
@Test
public void testBuyDonuts() {
    CreditCard creditCard = new CreditCard();
    Tuple<List<Donut>, Payment> purchase = DonutShop.buyDonuts(5, creditCard);
    assertEquals(Donut.price * 5, purchase._2.amount);
    assertEquals(creditCard, purchase._2.creditCard);
}
```

Zustandsmaschine

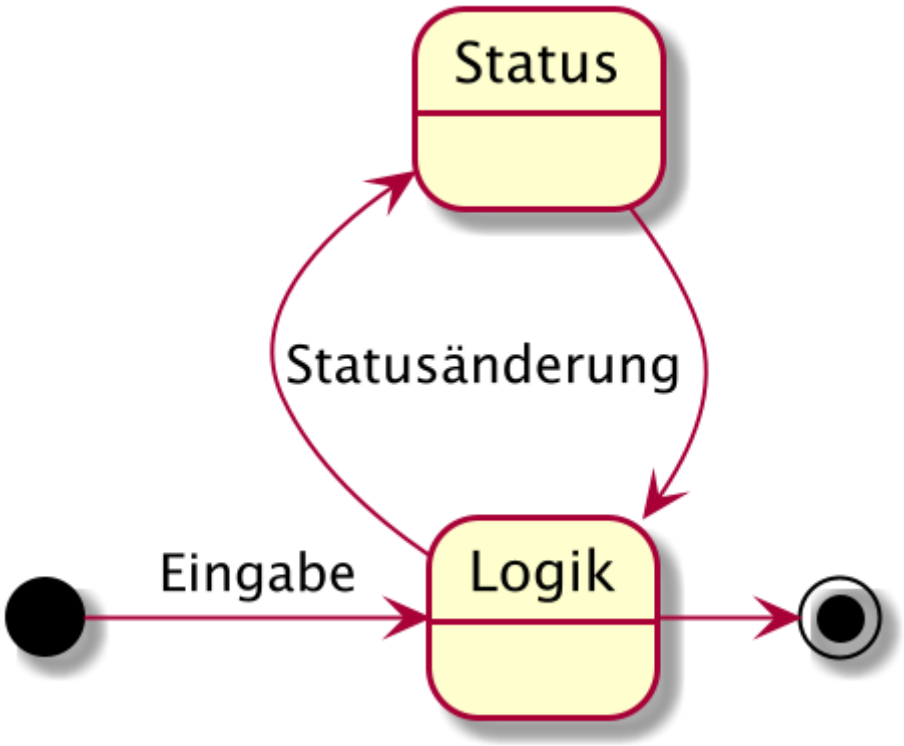
englisch: State Machine

- eigtl. *Endlicher Automat* bzw. *finite state machine*

Definition

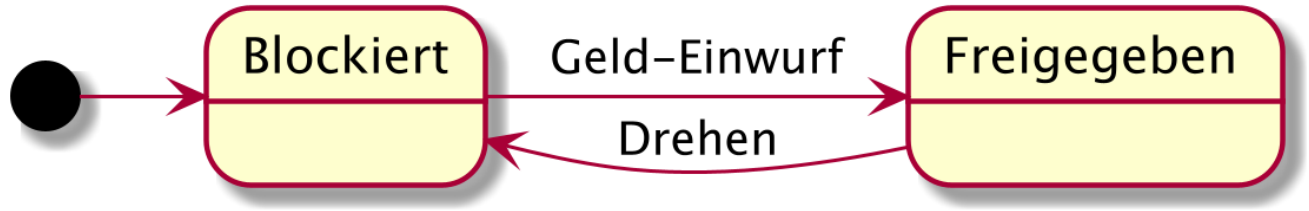
1. Eingabe / Aktion erfolgt
2. Programmlogik fällt Entscheidungen aufgrund des aktuellen Status und der Eingabe
3. ggf. erfolgt eine Statusänderung
4. ggf. erfolgt eine Ausgabe

Jedes Programm lässt sich als Zustandsmaschine beschreiben.



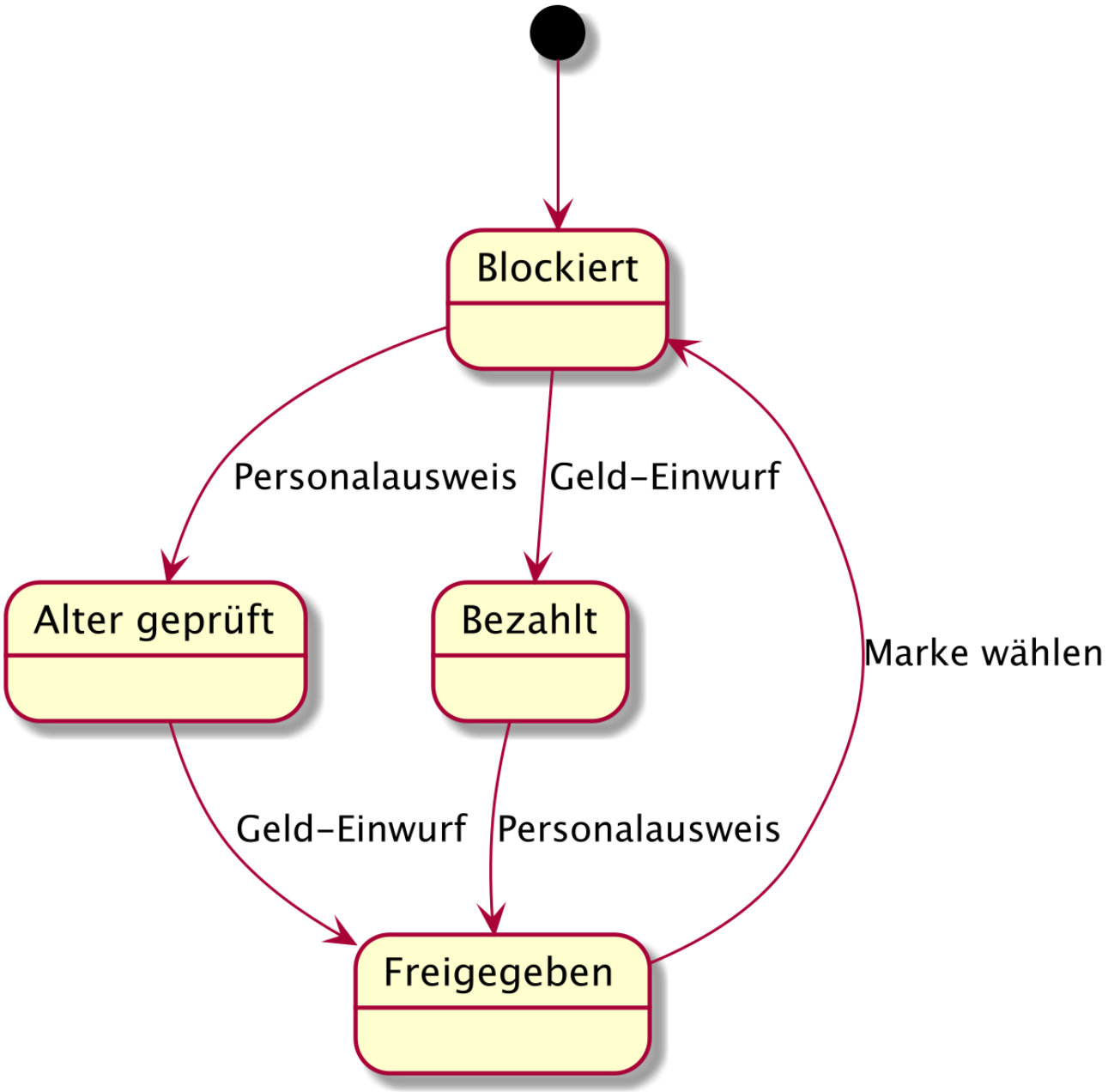
Beispiele

Kaugummi Automat



Status	Eingabe	Folgestatus	Ausgabe
Blockiert	Geld	Freigegeben	Dreh-Schalter freigegeben
Blockiert	Drehen	Blockiert	-
Freigegeben	Drehen	Blockiert	Dreh-Schalter blockiert
Freigegeben	Geld	Freigegeben	-

Zigaretten-Automat



Status	Eingabe	Folgestatus	Ausgabe
Blockiert	Geld	Bezahlt	Betrag wird angezeigt
Blockiert	Personal- ausweis	Alter geprüft	Bestätigung anzeigen
Bezahlt	Personal- ausweis	Freigegeben	Aufforderung zur Auswahl
Alter geprüft	Geld	Freigegeben	Aufforderung zur Auswahl
Freigegeben	Marke wählen	Blockiert	Zigaretten

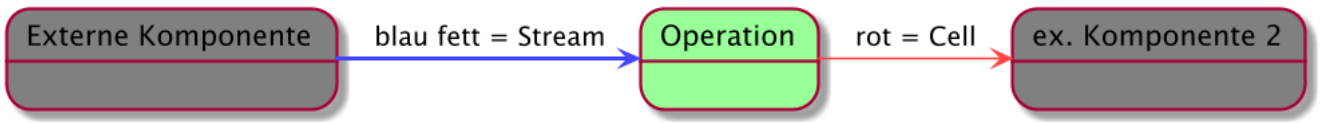
Grundlagen

alle folgenden Code-Beispiele in diesem Abschnitt von https://github.com/SodiumFRP/sodium

- Stream
  - ein Ereignisfluss

    - *auch* Observable, EventStream oder Event
- Cell (Zelle)
    - repräsentiert einen sich ändernden Wert
    - *auch* BehaviorSubject, Behavior oder Property
- Operation
    - eine Funktion / Code, der einen Stream oder eine Zelle in einen anderen Stream oder eine andere Zelle konvertiert
- Primitive
    - eine nicht weiter zerlegbare Operation
    - alle Operationen sind Primitive oder setzen sich aus diesen zusammen

Darstellung



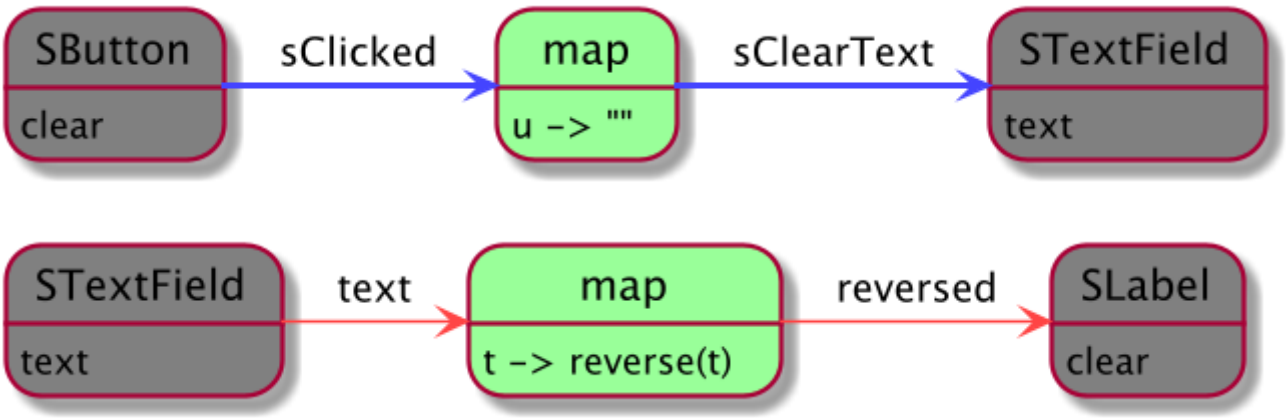
Datentyp Unit

- steht für *Nichts* / *kein Wert*
- nicht identisch mit *null*
- eine Funktion, die nichts zurück gibt, ist per Definiton konstant und damit keine richtige Funktion
    - ggf. interessiert der Rückgabewert aber nicht (z.B. bei der Ausgabe von Text)
        - `putStrLn :: String → IO ()`
    - ggf. interessiert der Eingabewert aber nicht (z.B. bei Auslöseschaltern)
        - `Event clicked(IO ())`

Primitiv Map

- konvertiert einen Stream / Zelle in einen anderen Stream / Zelle
- `map :: (a → b) → Stream a → Stream b`
- `map :: (a → b) → Cell a → Cell b`
- `<B> Stream<B> map(final Lambda1<A,B> f)`

Beispiel: Map



Map Stream

```
SButton clear = new SButton("Clear");
Stream<String> sClearIt = clear.sClicked.map(u -> "");
STextField text = new STextField(sClearIt, "Hello");
```

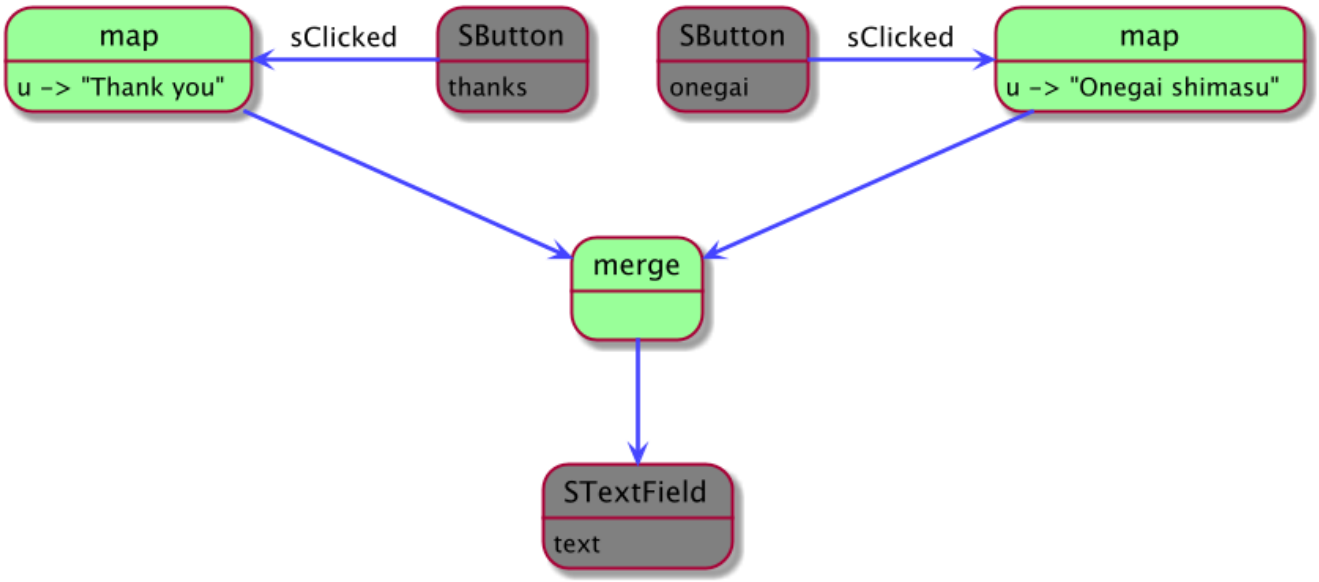
Map Cell

```
STextField msg = new STextField("Hello");
Cell<String> reversed = msg.text.map(t ->
    new StringBuilder(t).reverse().toString());
SLabel lbl = new SLabel(reversed);
```

Primitiv Merge

- führt zwei Streams vom gleichen Typ zusammen
- wenn einer der Streams feuert, wird dessen Wert weitergeleitet
- feuern beide gleichzeitig, entscheidet eine übergebene Funktion, was passiert
- `merge :: Stream a → Stream a → (a → a → a) → Stream a`
- `Stream<A> merge(final Stream<A> s, final Lambda2<A,A,A> f)`

Beispiel Merge

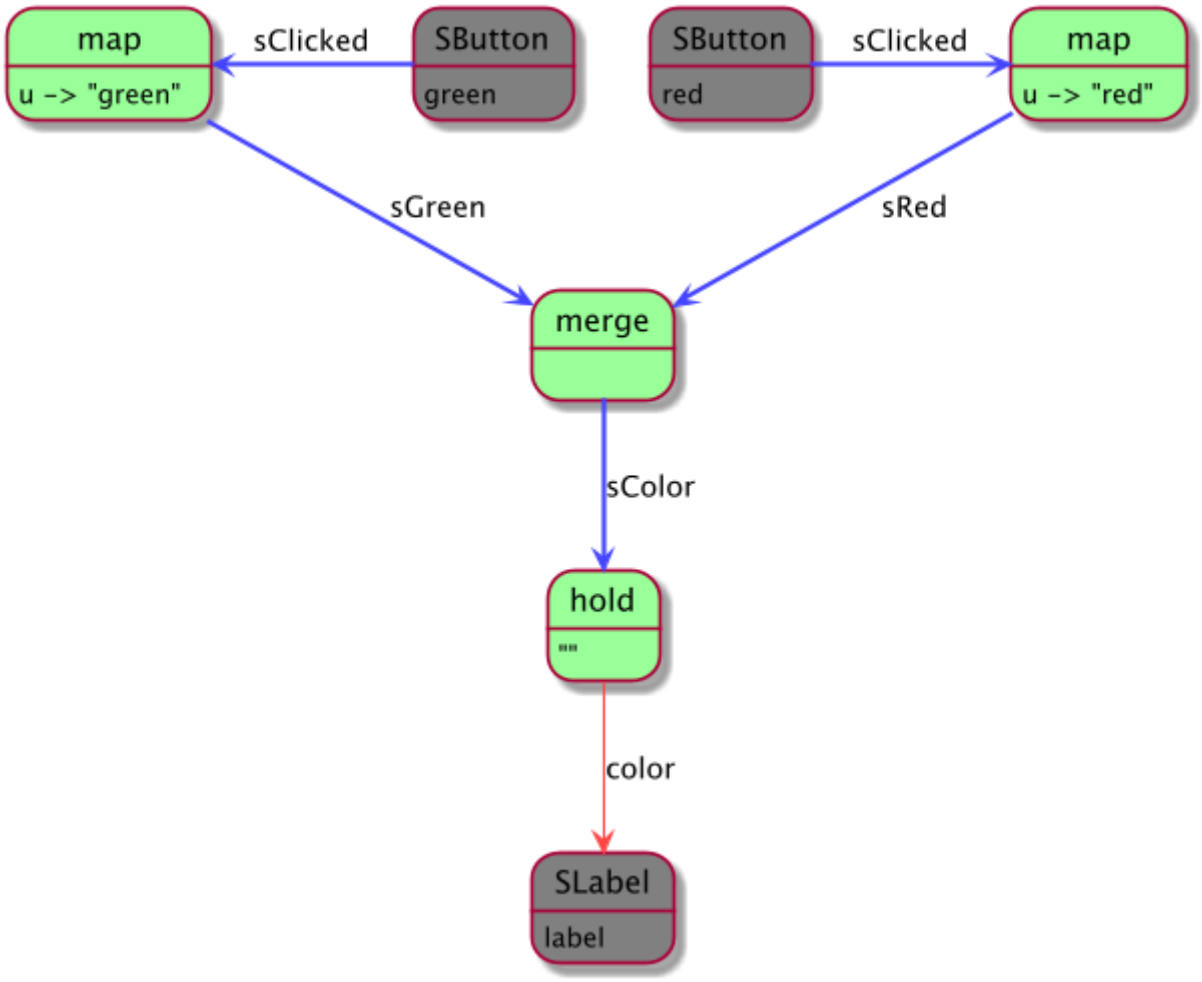



```
SButton onegai = new SButton("Onegai shimasu");
SButton thanks = new SButton("Thank you");
Stream<String> sOnegai = onegai.sClicked.map(u ->
    "Onegai shimasu");
Stream<String> sThanks = thanks.sClicked.map(u -> "Thank you");
Stream<String> sCanned = sOnegai.merge(sThanks, (vOnegai, vThanks) -> vOnegai);
// Kurzform:
// Stream<String> sCanned = sOnegai.orElse(sThanks);
STextField text = new STextField(sCanned, "");
```

Primitiv Hold

- aus einem Stream wird eine Zelle erzeugt, die immer den letzten Wert des Streams enthält
- `hold :: a → Stream a → T → Cell a`
- `public final Cell<A> hold(final A initValue)`

Beispiel Hold

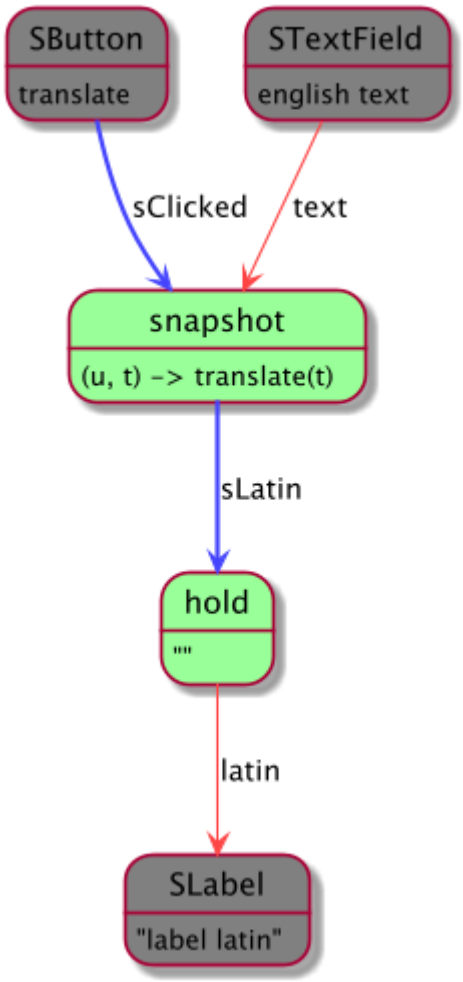



```
SButton red = new SButton("red");
SButton green = new SButton("green");
Stream<String> sRed = red.sClicked.map(u -> "red");
Stream<String> sGreen = green.sClicked.map(u -> "green");
Stream<String> sColor = sRed.orElse(sGreen);
Cell<String> color = sColor.hold("");
SLabel lbl = new SLabel(color);
```

Operation Snapshot

- holt den aktuellen Wert einer Zelle, sobald ein bestimmter Stream a feuert
- aus Stream a und dem Wert b wird Stream c befüllt
- `snapshot :: (a → b → c) → Stream a → Cell b → Stream c`
- `<B,C> Stream<C> snapshot(final Cell<B> c, final Lambda2<A,B,C> f)`

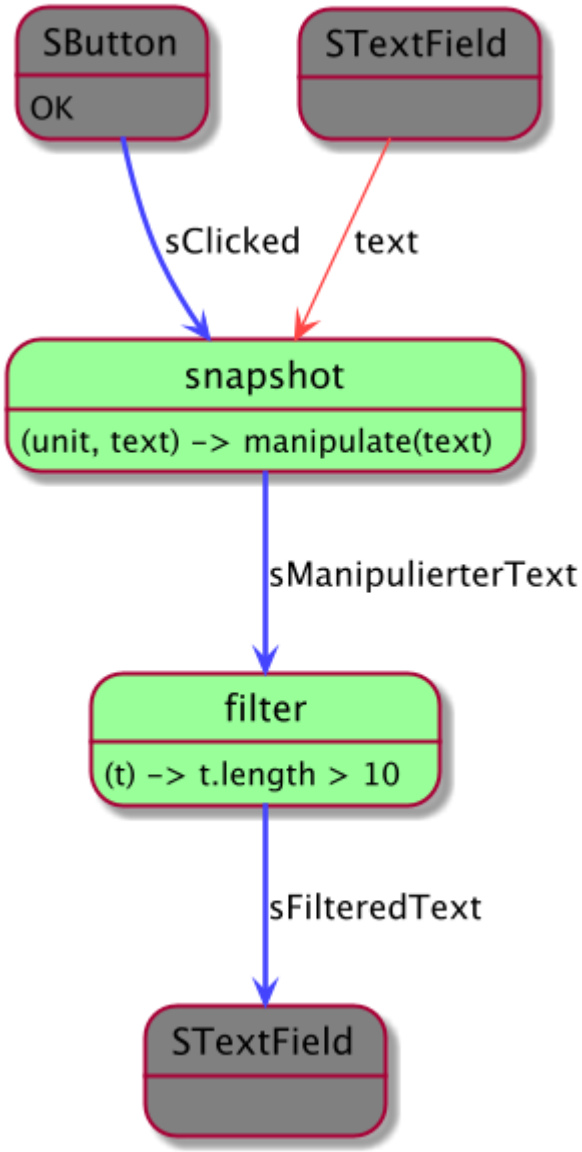
Beispiel Snapshot




```
STextField english = new STextField("I like FRP");
SButton translate = new SButton("Translate");
Stream<String> sLatin =
    translate.sClicked.snapshot(english.text, (u, txt) ->
    txt.trim().replaceAll(" |$", "us ").trim());
Cell<String> latin = sLatin.hold("");
SLabel lblLatin = new SLabel(latin);
```

Primitiv Filter

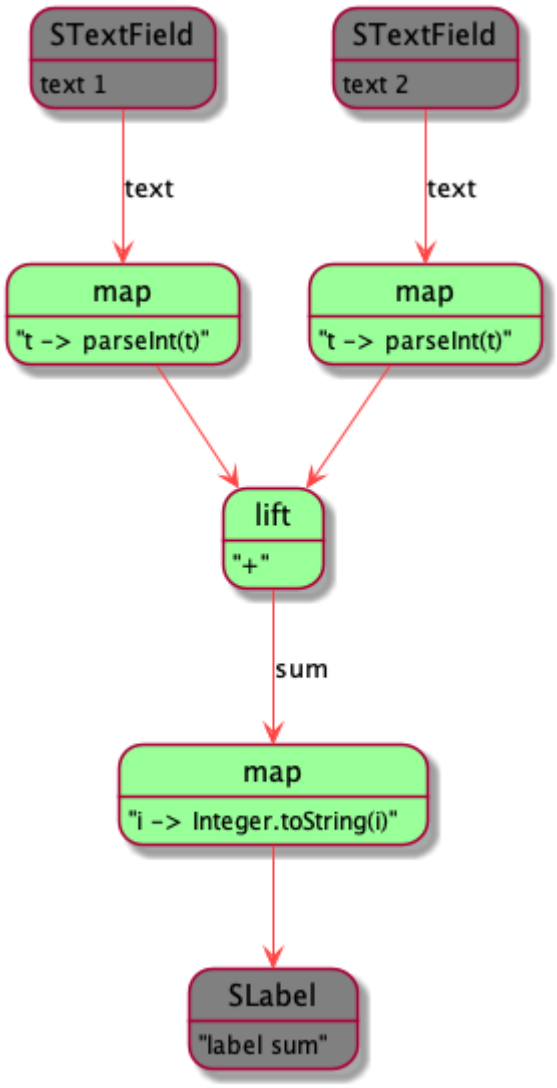
- wenn eine Eigenschaft wahr ist, wird der Wert des Streams in einen anderen Stream gefeuert
- `filter :: (a → Bool) → Stream a → Stream a`
- `Stream<A> filter(final Lambda1<A,Boolean> predicate)`



Primitiv Apply

- zwei Zellen werden verknüpft
- Fortführung: *lift* → eine beliebige Anzahl an Zellen wird verknüpft
- `Apply :: Cell (a → b) → Cell a → Cell b`
- `public <B,C> Cell<C> apply(Cell<B> b, Lambda2<A,B,C> fn)`

Beispiel Apply / Lift




```
STextField txtA = new STextField("5");
STextField txtB = new STextField("10");
Cell<Integer> a = txtA.text.map(t -> parseInt(t));
Cell<Integer> b = txtB.text.map(t -> parseInt(t));
Cell<Integer> sum = a.lift(b, (a_, b_) -> a_ + b_);
SLabel lblSum = new SLabel(sum.map(i -> Integer.toString(i)));
```

Primitiv Never Stream

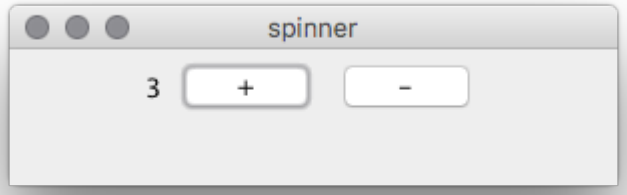
- ein Stream, der nicht feuern *kann*
- benutzt, um Funktionalität "auszuschalten"
- `Never :: Stream a`
- Umsetzung in Java: `new Stream<>()`

Akkumulator

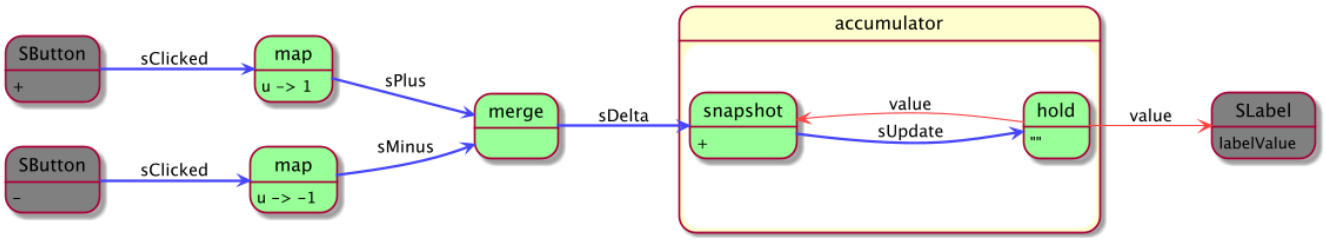
Ansammlen von Zustandsänderungen

Beispiel von der gleichen Quelle wie zuvor.

Spinner



in FRP



Java Code

```
Stream<Integer> sPlus = plus.sClicked.map(u -> 1);
Stream<Integer> sMinus = minus.sClicked.map(u -> -1);
Stream<Integer> sDelta = sPlus.merge(sMinus, (i1, i2) -> 0);

Stream<Integer> sUpdate = sDelta
                  .snapshot(valueCell, (delta, value) -> value + delta);
Cell<Integer> valueCell = sUpdate.hold(0);
```

Wo ist der Fehler?

- `sDelta.snapshot(valueCell, (delta, value) → value + delta);`
  - `valueCell` wird benutzt, bevor es deklariert wurde
  - `valueCell` und `sUpdate` hängen voneinander ab

Value Loop

```
CellLoop<Integer> loop = new CellLoop<>();
Stream<Integer> sUpdate = sDelta
                  .snapshot(loop, (delta, value) -> delta + value);
loop.loop(sUpdate.hold(0));
```

`StreamLoop` als Equivalent zu `CellLoop`.

- Java ist zu sequentiell -> CellLoop ist nur ein Hilfsmittel.
- Das funktionale Denken ist nicht sequentiell sondern abhängigkeitsbezogen.
- Funktionale Sprachen kennen die zeitliche Abhängigkeit was die Initialisierung angeht nicht.
  - Vergleich mit einem Schaltkreis: da gibt es auch voneinander abhängige Module -> spielt aber bei der Reihenfolge vom Zusammenbau keine Rolle, so lange am *Ende* alles richtig verdrahtet ist.

Java Code

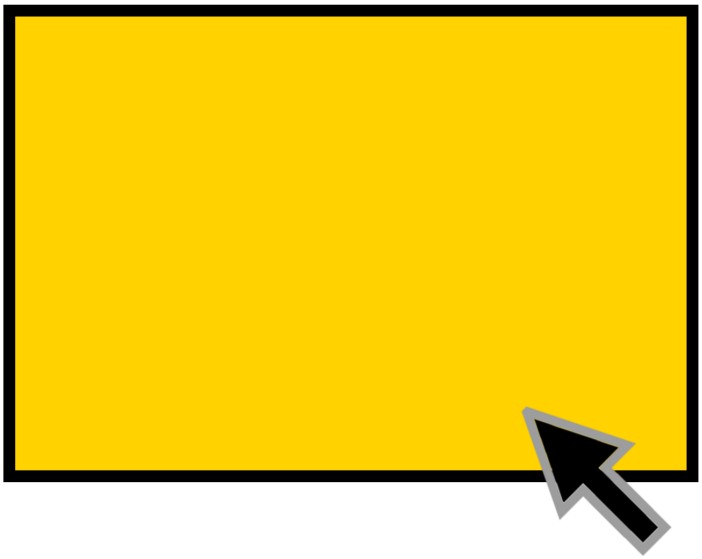
```
CellLoop<Integer> value = new CellLoop<>();
SLabel lblValue = new SLabel(
                     value.map(i -> Integer.toString(i)));
SButton plus = new SButton("+");
SButton minus = new SButton("-");
Stream<Integer> sPlusDelta = plus.sClicked.map(u -> 1);
Stream<Integer> sMinusDelta = minus.sClicked.map(u -> -1);
Stream<Integer> sDelta = sPlusDelta.orElse(sMinusDelta);
Stream<Integer> sUpdate = sDelta.snapshot(value,
      (delta, value_) -> delta + value_);
value.loop(sUpdate.hold(0));
```

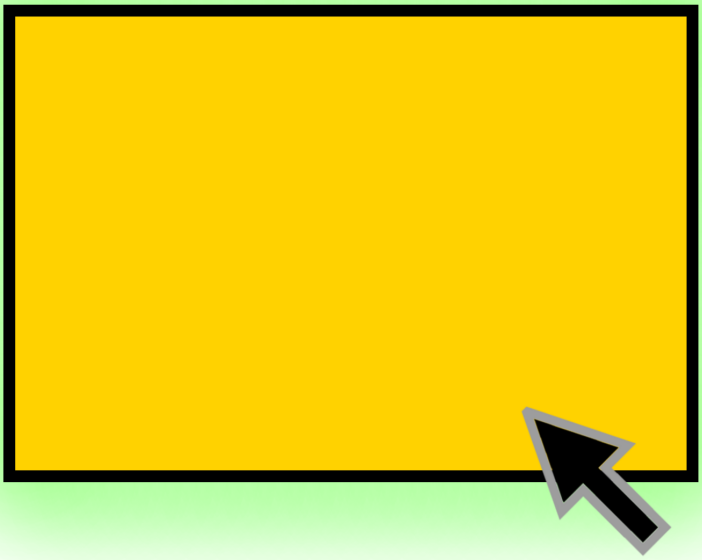
- Manche Frameworks erlauben kein Value Loop, sondern bieten eine Primitive *accumulate* oder ähnliches an.
- Zum Thema Transaktionen später mehr.
  - Transaktionen werden normal vom Framework automatisch gesteuert, aber hier schwierig.

Beispiel: Zeichenprogramm

Selektierung / Deselektierung

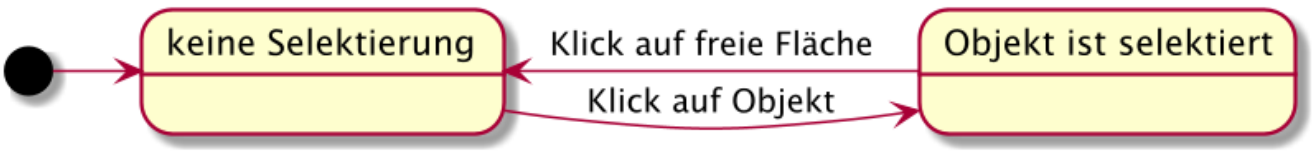
1. nichts ist selektiert
2. Objekt wurde selektiert durch Klick auf das Objekt
3. Selektierung wird aufgehoben durch Klick auf freie Fläche





Zustandsmaschine an der Tafel herleiten

Zustandsmaschine



Was passiert bei erneutem Klick auf das Objekt?

- Diagram gibt darauf keine Antwort
  - genauso keine Antwort, was passiert, wenn auf eine freie Fläche geklickt wird, ohne das ein Objekt selektiert war

Status	Eingabe	Folgestatus	Ausgabe
Keine Selektierung	Klick auf Objekt	Objekt selektiert	Objekt hervorheben
Keine Selektierung	Klick auf freie Fläche	Keine Selektierung	-
Objekt selektiert	Klick auf Objekt	Objekt selektiert	-

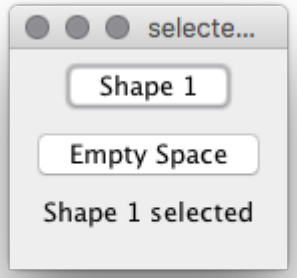
Status	Eingabe	Folgestatus	Ausgabe
Objekt selektiert	Klick auf freie Fläche	Keine Selektierung	Objekt nicht mehr hervorheben

An der Tafel die FRP Darstellung herleiten:

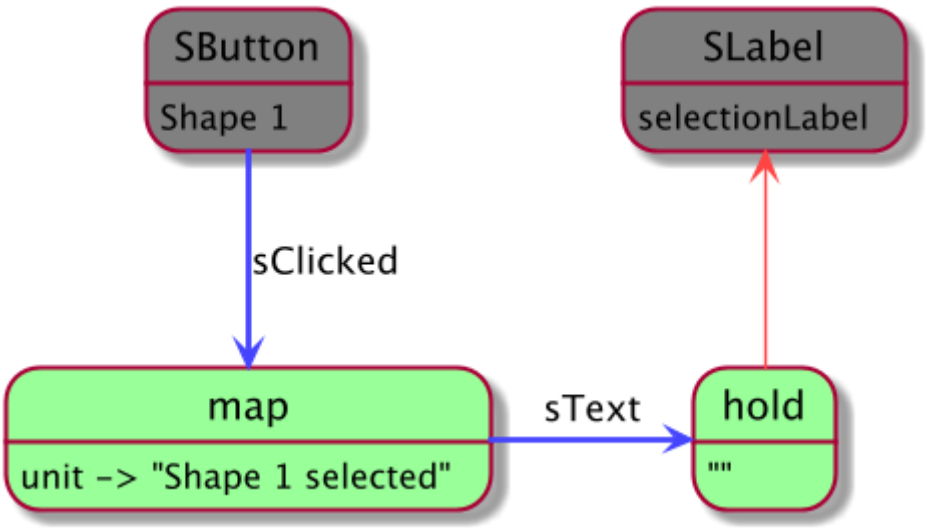
1. Selektierung
2. Deselektierung
3. Gesamt (Merge)

und stehen lassen, damit der Code nachher verständlicher erklärt werden kann.

Vereinfachtes Beispiel



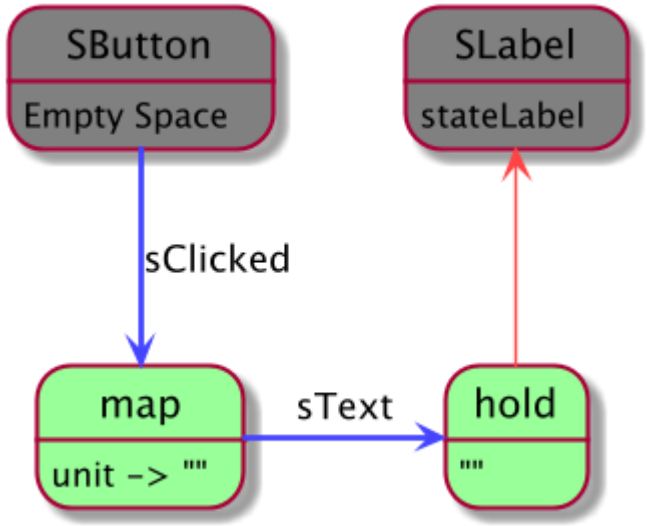
FRP: Selektierung



Java: Selektierung

```
SButton shape_1 = new SButton("Shape 1");
Stream<String> shape1Selected = shape_1.sClicked
                                    .map(u -> "Shape 1 selected");
SLabel lblSelection = new SLabel(shape1Selected.hold(""));
```

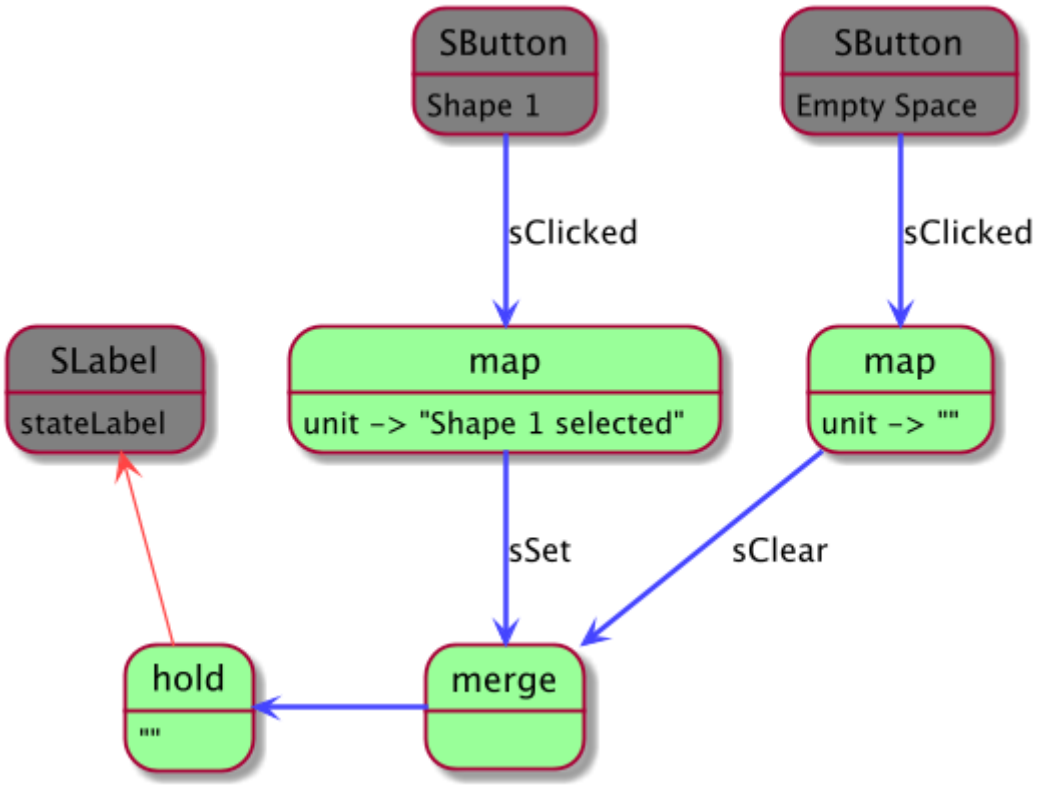
FRP: Deselektierung



Java: Deselektierung

```
SButton emptySpace = new SButton("Empty Space");
Stream<String> sClearSelection = emptySpace.sClicked.map(u -> "");
SLabel lblSelection = new SLabel(sClearSelection.hold(""));
```

FRP: Gesamt

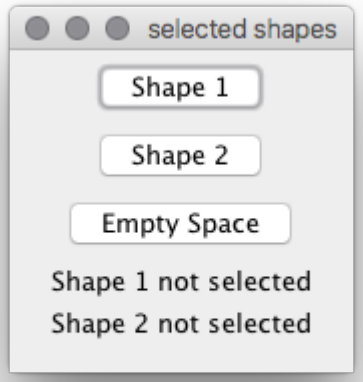


Java: Gesamt

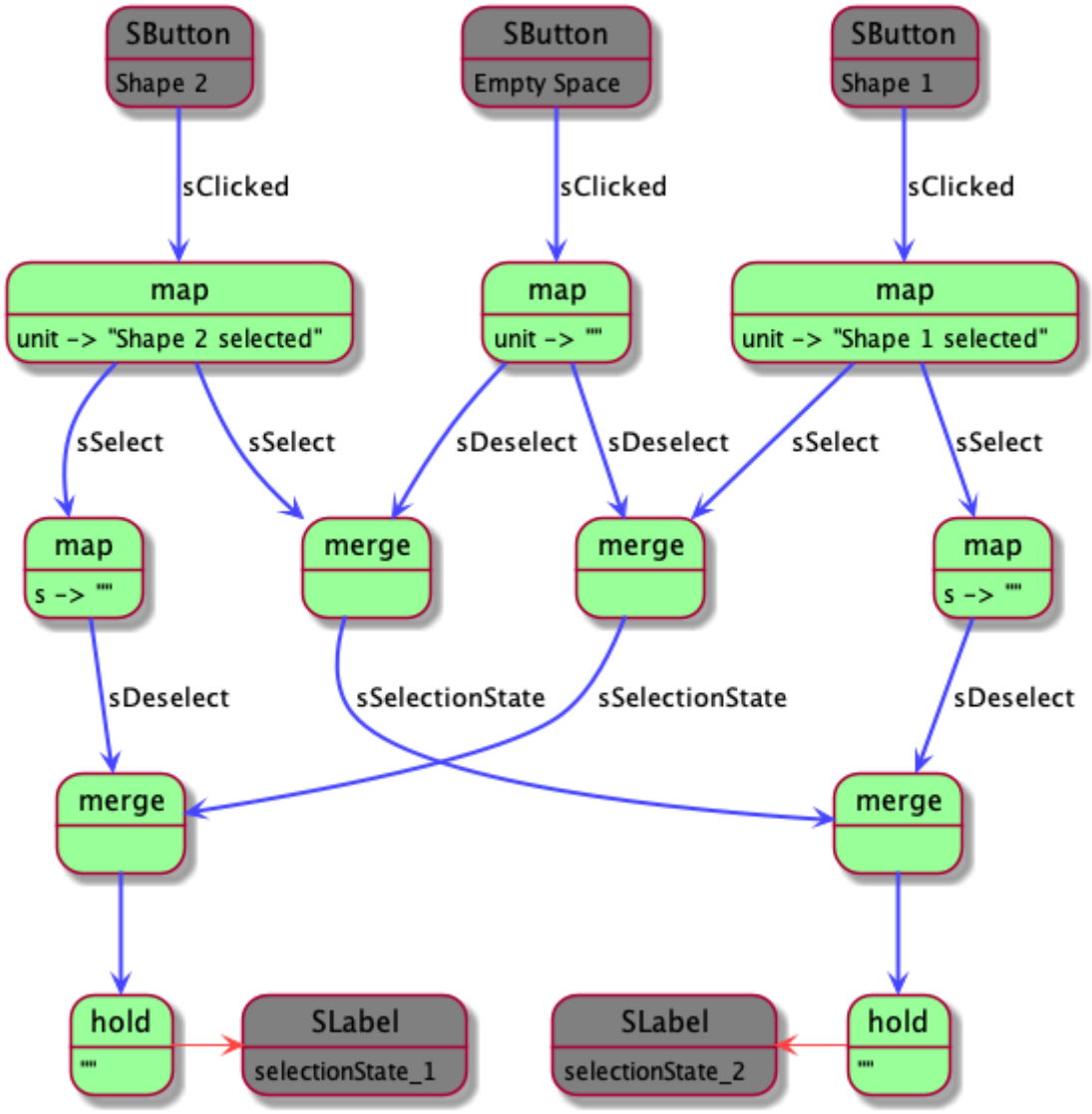
```
SButton shape_1 = new SButton("Shape 1");
SButton emptySpace = new SButton("Empty Space");

Stream<String> shape1Selected = shape_1.sClicked
                                    .map(u -> "Shape 1 selected");
Stream<String> shape1AndSpace = shape1Selected
                  .merge(emptySpace.sClicked.map(u -> ""), (s1, s2) -> "never happens
");

SLabel lblSelection = new SLabel(shape1AndSpace.hold(""));
```

Merge Funktion kann theoretisch leer sein, da es hier nicht zu gleichzeitigen Events kommen kann.

Vereinfacht: 2 Objekte



FRP: 2 Objekte



Java: 2 Objekte

```
SButton shape_1 = new SButton("Shape 1");
SButton shape_2 = new SButton("Shape 2");
SButton emptySpace = new SButton("Empty Space");

Stream<String> shape1Selected = shape_1.sClicked
                                       .map(u -> "Shape 1 selected");
Stream<String> shape2Selected = shape_2.sClicked
                                       .map(u -> "Shape 2 selected");
Stream<String> emptySpaceSelected = emptySpace.sClicked.map(u -> "");

Stream<String> unselectShape1 = emptySpaceSelected
                        .merge(shape2Selected, (s1, s2) -> "never happens")
                        .map(s -> "Shape 1 not selected");
Stream<String> unselectShape2 = emptySpaceSelected
                        .merge(shape1Selected, (s1, s2) -> "never happens")
                        .map(s -> "Shape 2 not selected");

Stream<String> shape1 = shape1Selected.merge(unselectShape1, (s1, s2) -> "never
happens");
Stream<String> shape2 = shape2Selected.merge(unselectShape2, (s1, s2) -> "never
happens");


SLabel lblSelection_1 = new SLabel(shape1.hold("Shape 1 not selected"));
SLabel lblSelection_2 = new SLabel(shape2.hold("Shape 2 not selected"));
```

Sample und Switch

Primitiv Sample

- gibt den Wert einer Zelle zu einem bestimmten Zeitpunkt
- `Sample :: Cell a → T → a`

Implementierung in Java

```
A sample() {
  return value;
}
```

Wie kann man daraus *Snapshot* nachbauen?

- Snapshot (original)
  - `Stream<A> sSnap = stream1.snapshot(cell, (valueStream1, valueCell) → doStuff(valueStream1, valueCell))`

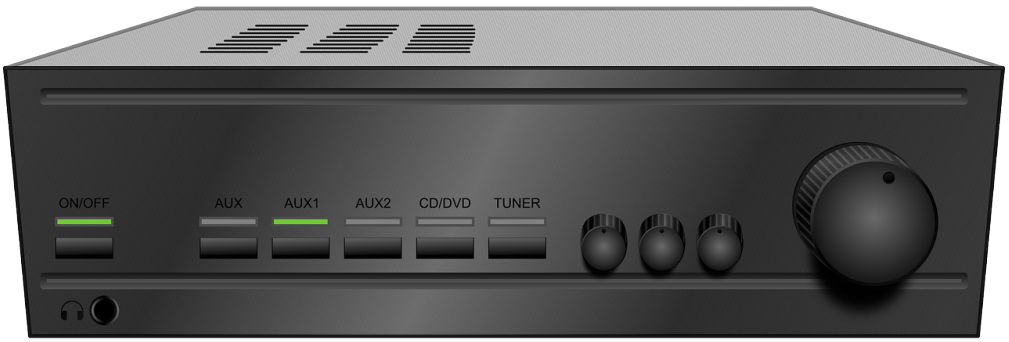
- Nachbau mit Map & Sample
  - `Stream<A> sSnap = stream1.map(value → doStuff(value, cell.sample()))`

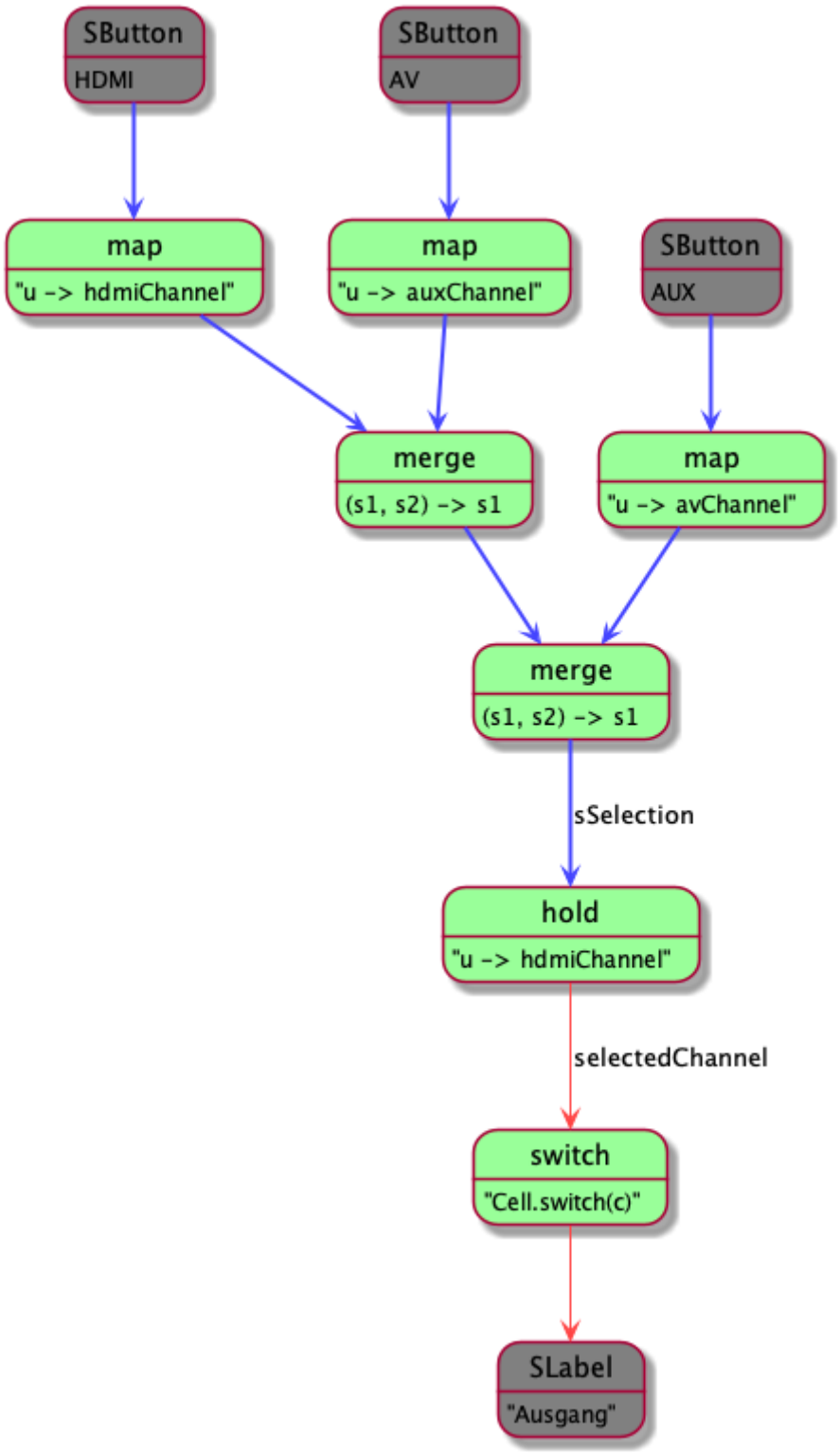
Primitiv Switch

- verknüpft immer den aktuellen Stream der Zelle
  - `Switch :: Cell (Stream a) → Stream a`
  - `Stream<A> switchS()`
- verknüpft immer die aktuelle Zelle der Zelle
  - `Switch :: Cell (Cell a) → T → Cell a`
  - `Cell<A> switchC()`
- auch *join* genannt

Beispiel: AV Receiver

- ein Receiver mit 3 Eingängen
  - HDMI, AV und AUX
- eine Fernbedienung, die zwischen den 3 Kanälen umschaltet





Der Knackpunkt ist: Wie kommt der SelectedChannel zustande?

Java

```
Cell<String> hdmiCell = new Cell<>("HDMI stream");
Cell<String> avCell = new Cell<>("AV stream");
Cell<String> auxCell = new Cell<>("AUX stream");

SButton hdmi = new SButton("HDMI");
SButton av = new SButton("AV");
SButton aux = new SButton("AUX");

Stream<Cell<String>> hdmiStream = hdmi.sClicked.map(u -> hdmiCell);
Stream<Cell<String>> avStream = av.sClicked.map(u -> avCell);
Stream<Cell<String>> auxStream = aux.sClicked.map(u -> auxCell);

Cell<Cell<String>> currentSelection = hdmiStream
                            .merge(avStream, (c1, c2) -> c1)
                            .merge(auxStream, (c1, c2) -> c1)
                            .hold(hdmiCell);

SLabel label = new SLabel(Cell.switchC(currentSelection));
```

- Warum nicht HDMI, AV und AUX als Stream?
  - Das sind in der realen Welt Kanäle; der Inhalt der Kanäle ist ein Stream im weiteren Sinne.
  - Man schaltet also Kanäle um -> Zellen. So ist man dynamisch.
  - Außerdem: hätten man einen Stream im FRP Sinne: er würde jedes Mal feuern, wenn sich sein Wert ändert -> also dauerhaft. Man möchte aber eigentlich nur das Signal *umbiegen* und keinen Dauerfeuer-FRP-Stream.

SwitchS

```
SButton func1Btn = new SButton("Func_1");
SButton func2Btn = new SButton("Func_2");
SButton fireBtn = new SButton("Fire");

Stream<String> func1 = fireBtn.sClicked.map(u -> "Func 1");
Stream<String> func2 = fireBtn.sClicked.map(u -> "Func 2");

Cell<Stream<String>> functionality = func1Btn.sClicked.map(u -> func1)
                                        .orElse(func2Btn.sClicked.map(u -> func2))
                                        .hold(new Stream<String>());

SLabel lblTest = new SLabel(Cell.switchS(functionality).hold("Start value."));
```

Switch Anwendungsfall

- dynamisches Verändern von Funktionalität (Stream) oder Status (Zellen)

- der *Graph* wird geändert

Transaktionen

Transaktion = logische Einheit mehrere Programmschritte

Transaktionen kommen u.a. in Datenbanken häufig vor.

Alternative Namen: * moment (der Moment, der Augenblick): betont das Konzept anstatt den Ablauf ** wird so auch in manchen Frameworks genannt * instant (gleiche Übersetzung)

Eigenschaften: ACID

- A
  - Atomarität (*Atomicity*): Alles-Oder-Nichts-Prinzip → von außen sieht es aus wie ein Schritt
- C
  - Konsistenz (*Consistency*): Transaktionen hinterlassen den Datenbestand / Zustand konsistenz (falls er voher konsistenz war)
- I
  - Isolation (*Isolation*): Transaktionen laufen isoliert ab → unabhängig von ihrer Umgegebung und anderen Transaktionen

- D
  - Dauerhaftigkeit (*Durability*): Auswirkungen bleiben bestehen

Dauerhaftigkeit: schließt Transaktion in Transaktion aus → denn wenn die äußere Transaktion fehlschlägt, muss die innere zurück genommen werden

ACID und Funktionen

Reine Funktionen (pure functions)

- gleiche Eingabe liefert immer gleiches Ergebnis
- hängen von nichts ab, dass sich während der Ausführung verändern kann (z.B. Variablen)
- haben keine Nebeneffekte
  - keine externe Zustandsänderung (außer durch Rückgabe)
  - keine Exceptions
- keine I/O Operationen

Erfüllte Eigenschaften

- Isolation: reine Funktionen sind unabhängig von ihrer Umgebung
- Atomarität: reine Funktionen laufen durch oder nicht → es gibt keinen Zustand dazwischen
- Konsistenz: nicht inhärent
- Dauerhaftigkeit: nicht inhärent

Funktionen in FRP

- hängen vom Zustand ab → Zustand kann sich ändern
  - Ereignisverarbeitung hängt (fast) immer vom Zustand ab
- Zustand wird über Zellen abgebildet
  - Zellen isolieren veränderbare Werte
  - Kompositionalität wird dadurch sicher gestellt

Transaktionen und FRP

- das Framework kümmert sich um Transaktionen automatisch
- explizite Transaktionen sind möglich
  - z.B. hilfreich beim Initialisieren
- Threadsicherheit / Nebenläufigkeit leicht umzusetzen
- Aktionen können *gleichzeitig* (Stichwort: Atomarität) stattfinden
- Achtung: manche Frameworks unterstützen Transaktionen nicht (z.B. die Reactive Extensions

(Rx))

Zeitkontinuierlich

Conal Elliott

- Erfinder von FRP

- FRP: formale Semantik und zeitkontinuierlich

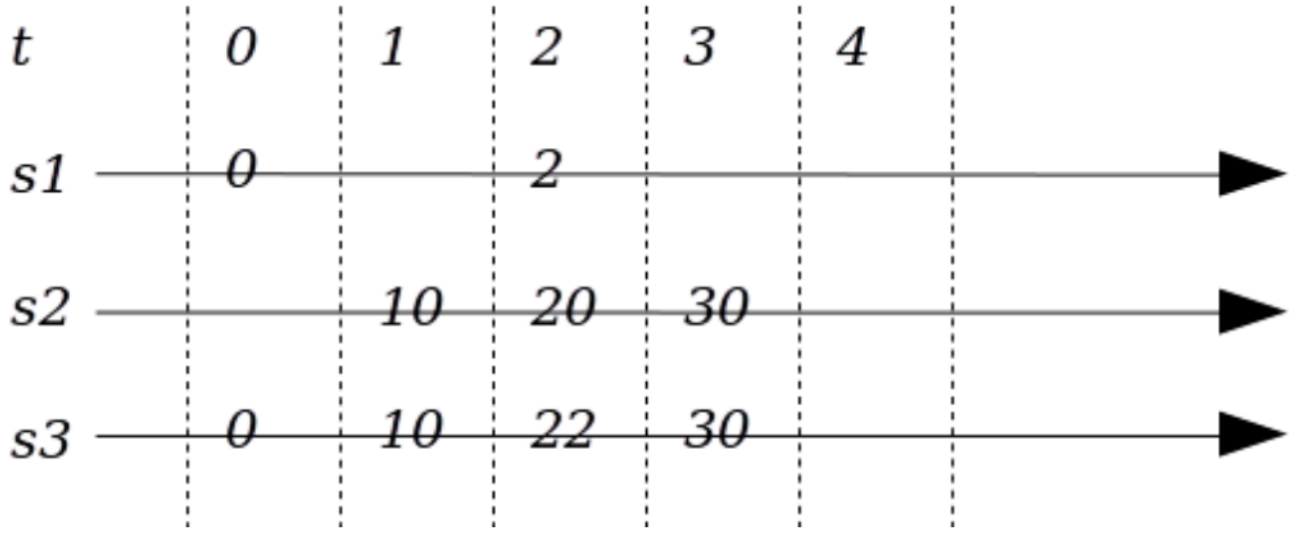
Formale Semantik

Beispiel von Sodium `merge`

```
Merge :: Stream a → Stream a → (a → a → a) → Stream a

occs (Merge sa sb) = coalesce f (knit (occs sa) (occs sb))
  where knit ((ta, a):as) bs@((tb, _):_) | ta <= tb = (ta, a) : knit as bs
        knit as@((ta, _):_) ((tb, b):bs) = (tb, b) : knit as bs
        knit as bs = as ++ bs
coalesce :: (a → a → a) → S a → S a
coalesce f ((t1, a1):(t2, a2):as) | t1 == t2 = coalesce f ((t1, f a1 a2):as)
coalesce f (ta:as) = ta : coalesce f as coalesce f [] = []
```

Formale Semantik: Testfall

```
let s1 = MkStream [([0], 0), ([2], 2)]
let s2 = MkStream [([1], 10), ([2], 20), ([3], 30)]
let s3 = Merge s1 s2 (+)
```



Sinks

- FRP Framework muss in den Rest integriert werden
  - Streams befeuern und direkt in Zellen schreiben
  - auf Ereignisse von Streams und Zellen reagieren

```
public class StreamSink<A> extends StreamWithSend<A> {
    public void send(final A a) { /* [...] */ }
}
```

```java
public final class CellSink<A> extends Cell<A> {
    //[...]
    public void send(A a) { /*[...]*/ }
}
```

- zusätzlich je: `Listener listen(Handler<A> action)`

Functional Data Structures

- Funktionale Datenstrukturen
  - Nicht-veränderbare Datenstrukturen
- bei externer Anbindung ein Muss

Gegenbeispiel

```java
Cell<List<String>> cell =  new Cell<>(new ArrayList<>());
Cell<List<Integer>> mappedCell = cell
       .map(value ->
                value.stream().map(Integer::parseInt)
                        .collect(Collectors.toList()));

// anderes Modul
Cell<List<String>> cell = getCellFromExample();
cell.sample().add("Hello");
```

- `ConcurrentModificationException` kann auftreten

Vorteile von Immutability

- Thread-sicher
- konsistente Zustände
- geringere Kopplung
- einfacher zu verstehen
- kürzerer Code

Analogie: Bilddarstellung

- Rastergrafik (auch Pixelgrafik)
  - Pixel sind rasterförmig angeordnet
  - ein Pixel repräsentiert eine Maßeinheit
  - jedes Pixel hat einen eigenen Farbwert

    - Auflösung hängt von der Anzahl der Pixel ab
- Vektorgrafik
    - aus verschiedenen Primitive zusammengesetzt
        - z.B. Linien, Kreise, Polygone, Kurven
    - skaliert beliebig (berechnet Rastergrafik)

TimerSystem.java

- Sodium spezifisch
- jedes echte FRP System braucht etwas ähnliches

```
class TimerSystem {
    // ...
    public final Cell<T> time;
    public Stream<T> at(Cell<Optional<T>> tAlarm) {
        // ...
    }
}
```

Zeitkontinuierlich

- deklarativ wird der Zustand anhand der Zeit definiert
- Zeit ist kontinuierlich → Maschinen *rastern* sie

Rasterung

```
private static void loop() throws InterruptedException {
    long systemSampleRate = 1000L;
    StreamSink<Unit> sMain = new StreamSink<>();
    while(true) {
        sMain.send(Unit.UNIT);
        Thread.sleep(systemSampleRate);
    }
}
```

Beispiel: Zeitanzeige (linear)

```
TimerSystem timerSystem = new SecondsTimerSystem();
Cell<Double> time = timerSystem.time;

SLabel lblValue = new SLabel(time.map(value -> Double.toString(value)));
```

Beispiel: Freier Fall

```
TimerSystem timerSystem = new SecondsTimerSystem();
Cell<Double> time = timerSystem.time;

//v(t) = g*t
Cell<Double> velocity = time.map(seconds -> 9.81 * seconds);
//s(t) = 1/2 * g * t^2
Cell<Double> distance = time.map(seconds -> 0.5 * 9.81 * seconds * seconds);

SLabel lblSeconds = new SLabel(time.map(value -> Double.toString(value) + " s"));
SLabel lblSpeed = new SLabel(velocity.map(value -> Double.toString(value) + " m/s"));
SLabel lblDistance = new SLabel(distance.map(value -> Double.toString(value) + " m"));
```

Beispiel: Ball

- startet mit bestimmter Höhe
- fällt nach unten
- weniger als 0m geht nicht

Live Coding

siehe GitHub Repository → beispiele/frp/ball.java

FRP vs. Observer

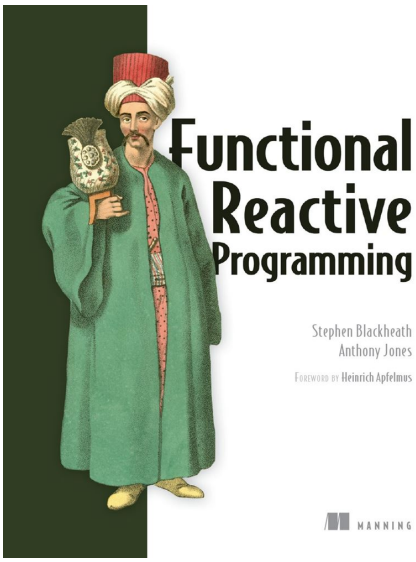
- viele der Probleme des Observers sind implizit gelöst
  - Transaktionen sind vorhanden
  - die Ablaufreihenfolge ist immer gleich (keine unvorhersehbare Reihenfolge bzw. die resultierenden Probleme)
  - Threadsicherheit ist gegeben, da keine Abhängigkeit nach außen existiert
  - es ändern sich nur die Sachen, die es betrifft (keine ungewollten Benachrichtigungen)
  - *Observer* können nicht vergessen werden
  - Ereignisse können nicht verpasst werden

Vertiefung

Beispiel *Drag and Drop* mit Listener und FRP

Literatur

Functional Reactive Programming *von Stephen Blackheath und Anthony Jones*



Functional Programming in Java *von Pierre-Yves Saumont* [FPJ-17]