FRP

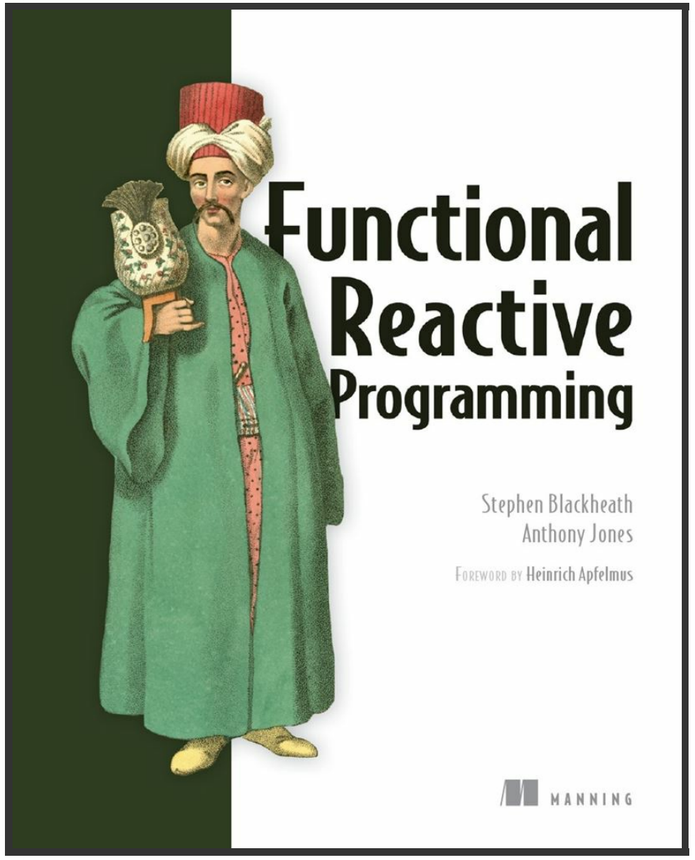
Maurice Müller

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.

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

"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.

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

DEFINITION DES ERFINDERS VON FRP

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

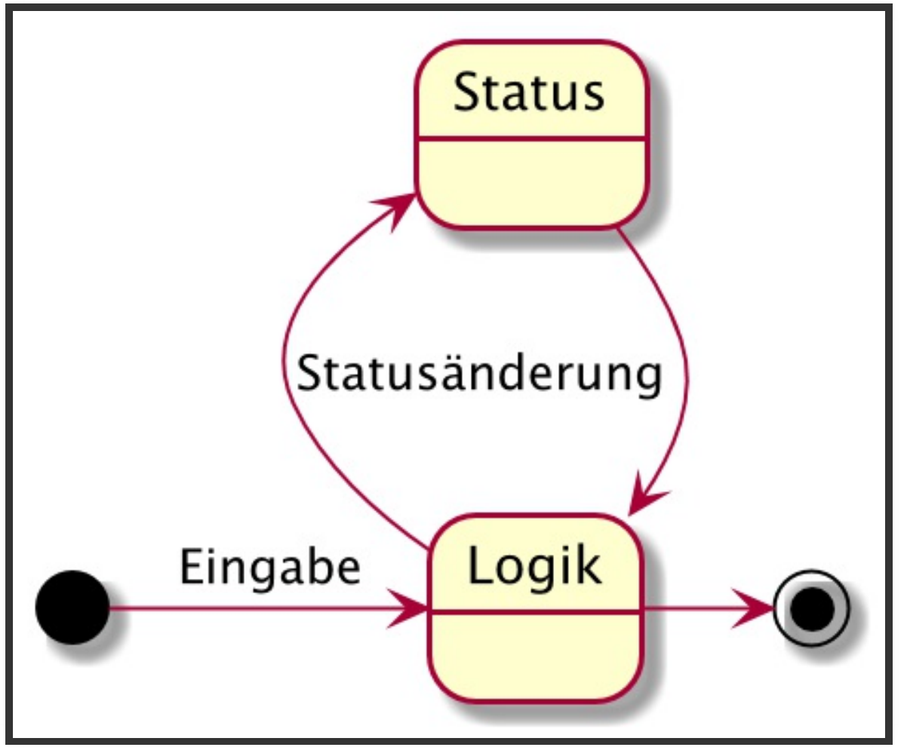
ZUSTANDSMASCHINE

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

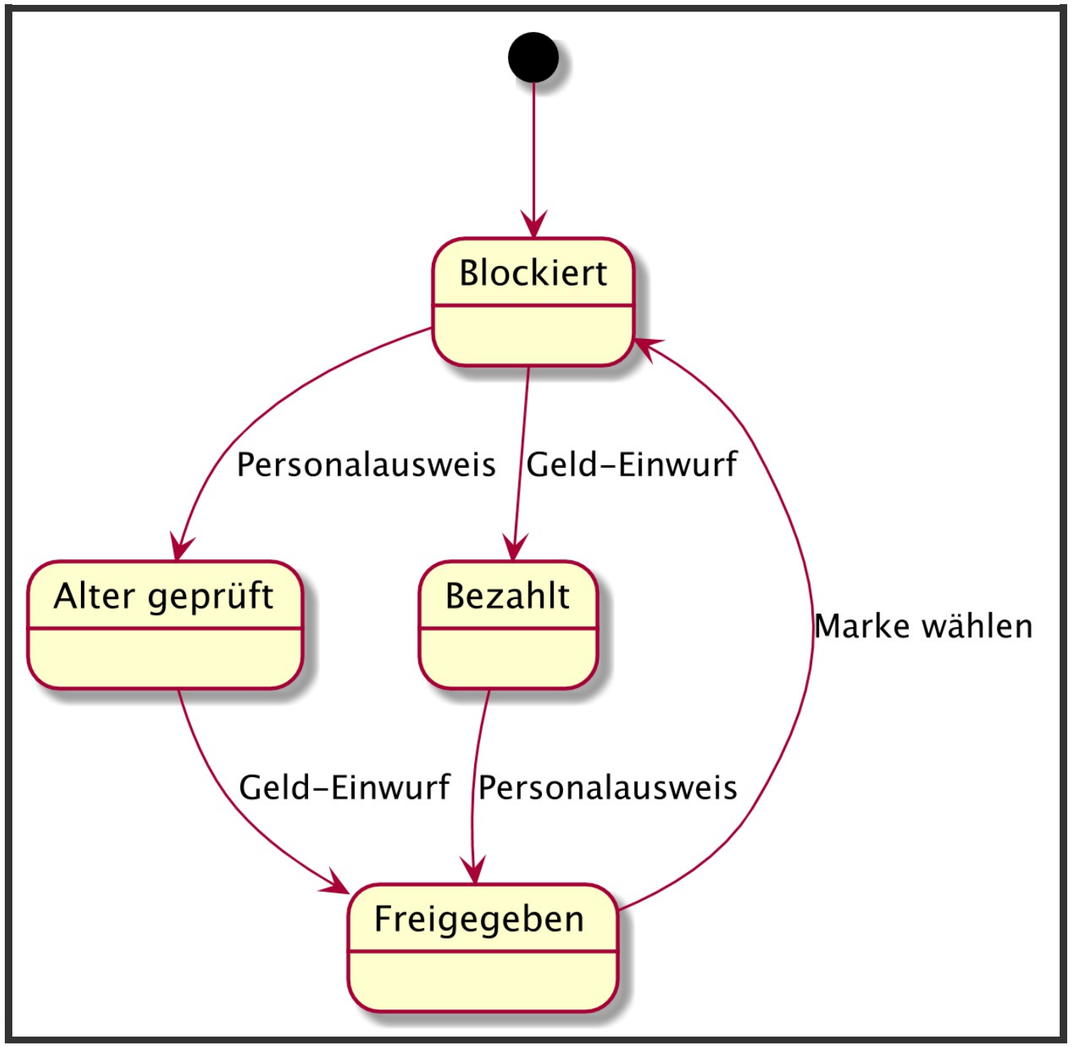

Status
Statusänderung
Eingabe
Logik

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	Blockiert	Zigaretten

EXKURS: ENTWURFSMUSTER

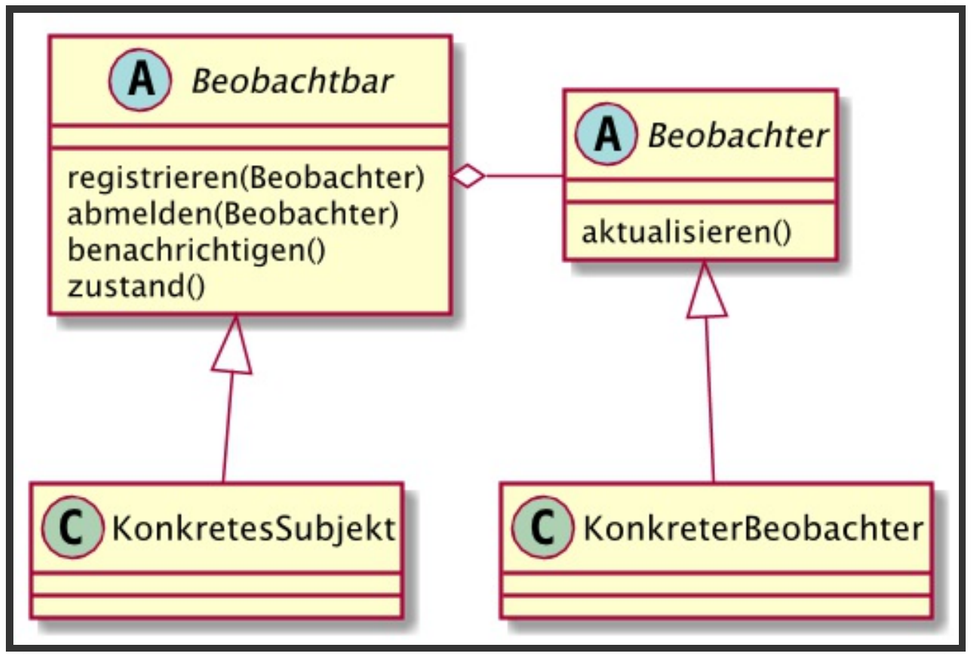
Wie reagiert man auf Statusänderung?

- Beobachter-Muster
  - englisch: *listener* oder *observer pattern*

WAS ES LÖST

- 1:n Abhängigkeit zwischen Objekten ohne hohe Kopplung
- eine beliebig große Anzahl an Objekten sollen sich automatisch ändern, wenn sich ein bestimmtes Objekt ändert
- ein bestimmtes Objekt muss abhängige Objekte informieren

UML


Beobachtbar
registrieren(Beobachter)
abmelden(Beobachter)
benachrichtigen()
zustand()
Beobachter
aktualisieren()
KonkretesSubjekt
KonkreterBeobachter

- Wie kommt der Beobachter an den aktuellen Zustand?
  - Beobachter::aktualisieren(konkretesSubjekt)
  - Beobachter::aktualisieren(spezifischerWert)
  - new KonkreterBeobachter(konkretesSubjekt)
  - abhängig vom Anwendungsfall

- Wer löst die Methode Beobachtbar::benachrichtigen aus?
  - KonkretesSubjekt selbst
    - (+) keine Änderung wird übersehen
    - (-) jede Änderung löst aus
  - Benutzer von konkretesSubjekt
    - (+) Transaktionen möglich
    - (-) *benachrichtigen* kann leicht übersehen werden

VORTEILE

- Entkopplung von Subjekt und Beobachter
- ein konkreter Beobachter kann mehrere Subjekte beobachten
- der Beobachter hängt vom Subjekt ab
  - und nicht das Subjekt von spezifischen Beobachtern
- dynamisches Registrieren und Abmelden von Beobachtern zur Laufzeit

NACHTEILE

- Beobachter bekommen immer die Benachrichtigung
  - selbst wenn manche davon nicht benötigt werden
- ggf. umständliches (fehleranfälliges) Aufräumen nach Löschung eines KonkretesSubjekt
- komplizierte oder fehleranfällige Transaktionsimplementierung
- nicht implizit Thread-safe

IMPLEMENTIERUNG SUBJEKT

```
class Meinung implements Beobachtbar {

  private List<Beobachter> beobachterListe = new ArrayList<>();
  private boolean ja = true;

  public void registrieren(Beobachter beobachter) {
    beobachterListe.add(beobachter);
  }

  public void abmelden(Beobachter beobachter) {
    beobachterListe.remove(beobachter);
  }

  public void toggleState() {
    ja = !ja;
    beobachterListe.forEach(beobachter ->
                beobachter.aktualisieren(this));
  }
```

IMPLEMENTIERUNG BEOBACHTER

```
class Meinungsausgabe implements Beobachter<Meinung> {

  public void aktualisieren(Meinung meinung) {
    System.out.println("Die Meinung hat sich geändert!");
    System.out.println("Ihrem Vorhaben wird nun " + meinung.status()
                    + " entgegen gebracht.")
  }

}
```

PLAGE: UNGEWOLLTE REKURSION

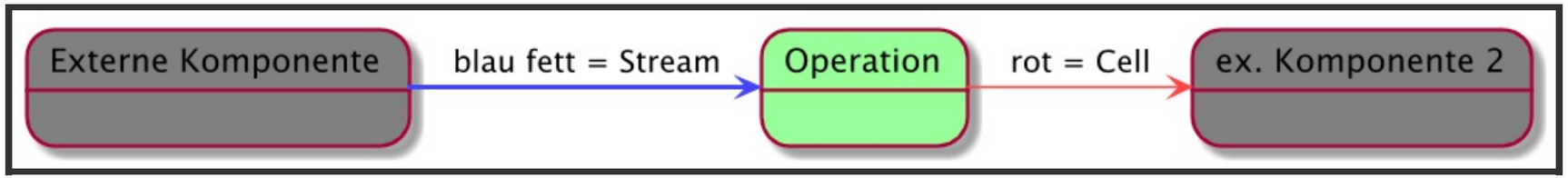
1. ein Beobachter ändert den Zustand des Subjekts, nachdem er informiert wurde
2. ein anderer Beobachter empfängt dieses Änderungen und ändert auch das Subjekts
3. der erste Beobachter wird informiert und es geht von vorne los

GRUNDLAGEN

- 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



QUELLE

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

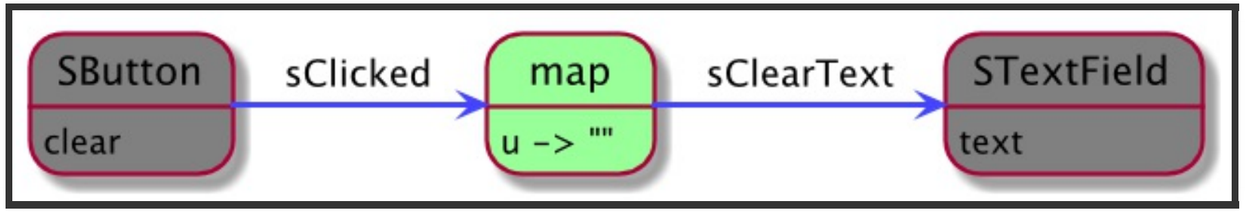
DATENTYP UNIT

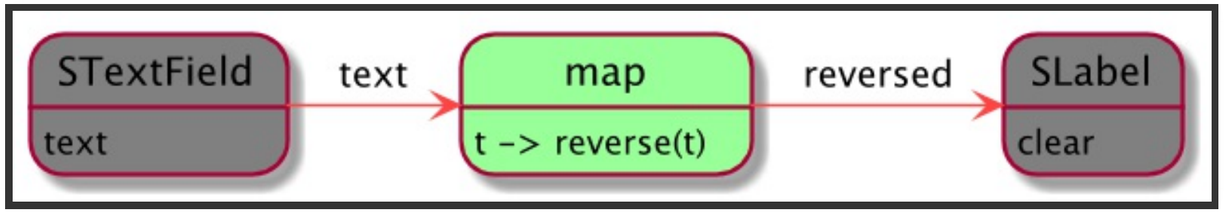
- steht für *Nichts / kein Wert*
- nicht identisch mit *null*
- eine Funktion, die nichts zurück gibt, ist per Definiton konstant und damit keine richtige Funktion
  - ggf. interessiert der Rückgabewert aber nicht (z.B. bei der Ausgabe von Text)
    - putStrLn :: String → IO ()
  - ggf. interessiert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

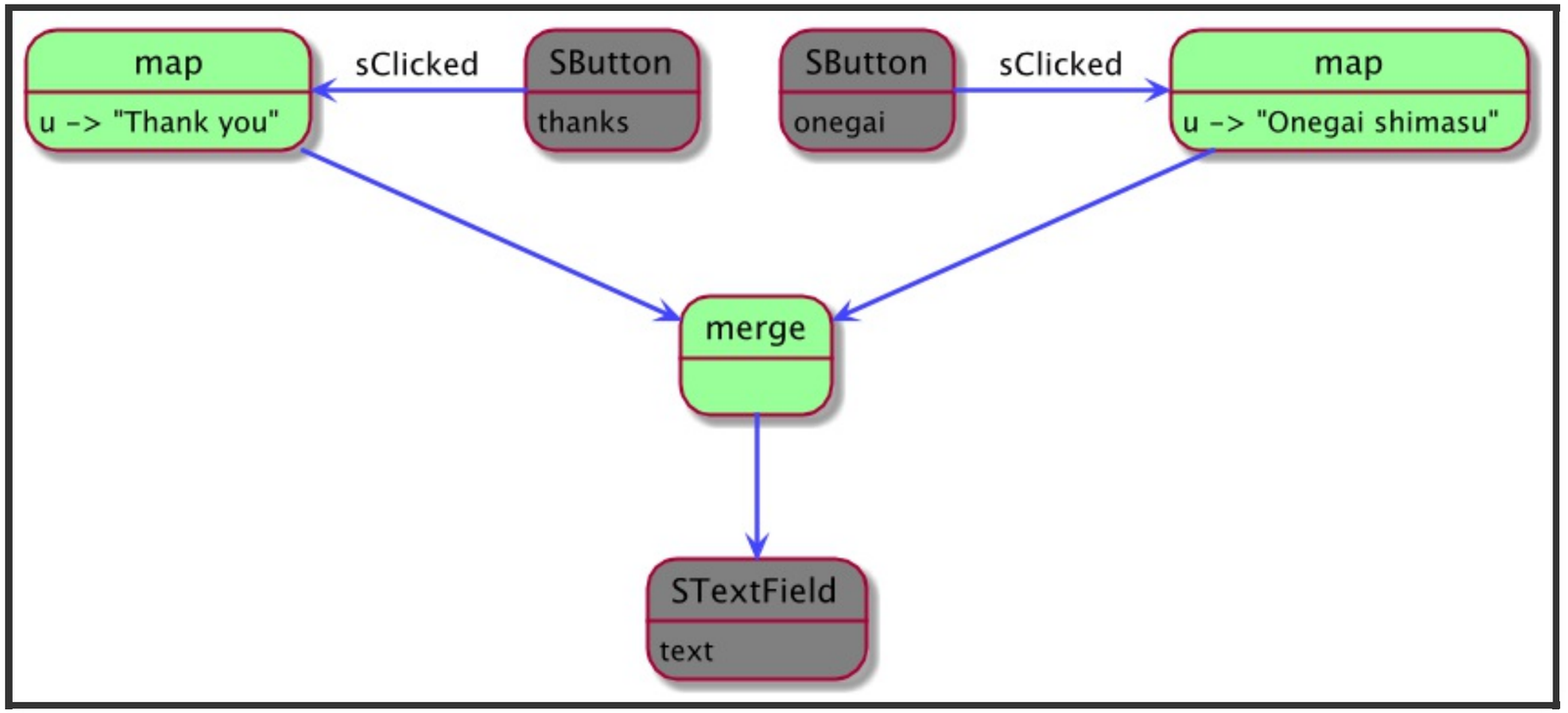
Map Cell

```
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

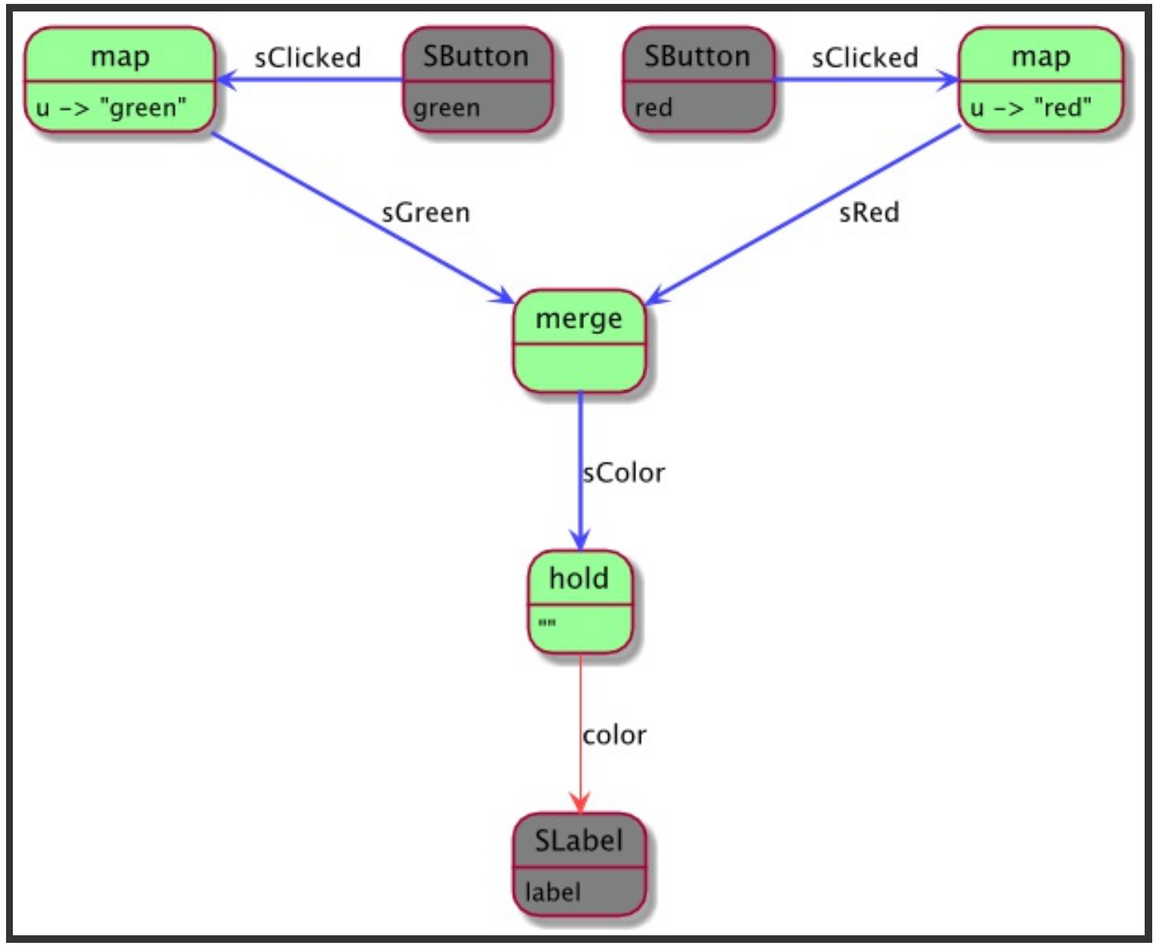


```
SButton onegai = new SButton("Onegai shimasu");
SButton thanks = new SButton("Thank you");
Stream<String> sOnegai = onegai.sClicked.map(u ->
    "Onegai shimasu");
Stream<String> sThanks = thanks.sClicked.map(u -> "Thank you");
Stream<String> sCanned = sOnegai.orElse(sThanks);
//sCanned = sOnegai.merge(sThanks,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

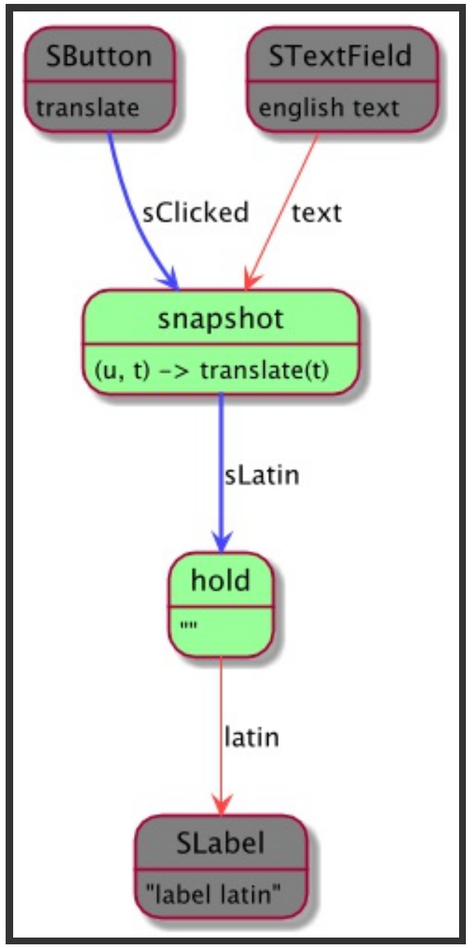


```
SButton red = new SButton("red");
SButton green = new SButton("green");
Stream<String> sRed = red.sClicked.map(u -> "red");
Stream<String> sGreen = green.sClicked.map(u -> "green");
Stream<String> sColor = sRed.orElse(sGreen);
Cell<String> color = sColor.hold("");
SLabel lbl = new SLabel(color);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

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)

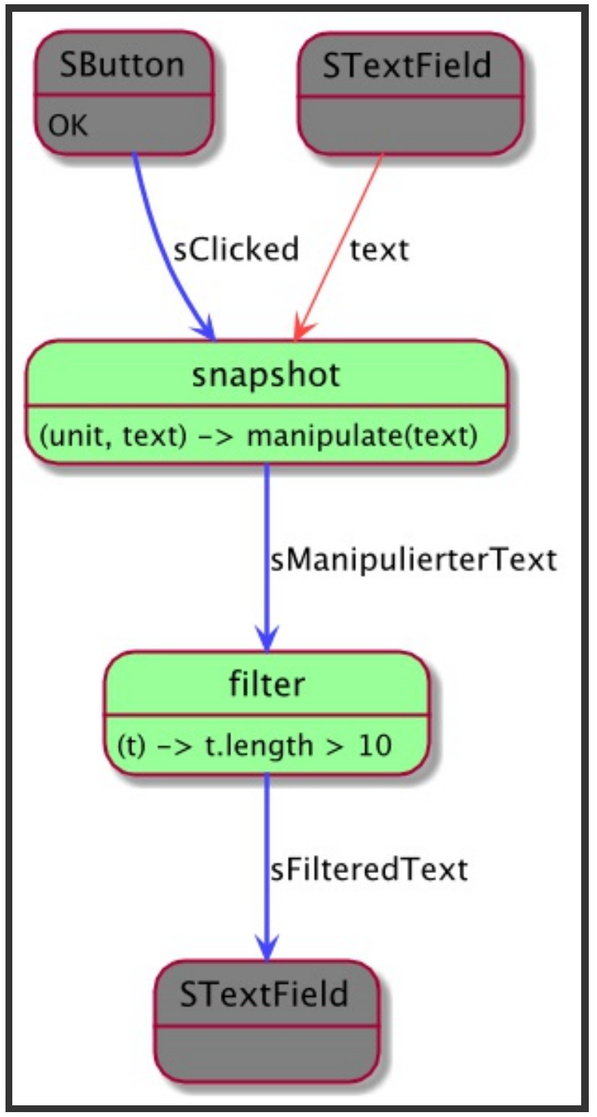
BEISPIEL SNAPSHOT



```
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(" I$", "us ").trim());
Cell<String> latin = sLatin.hold("");
SLabel lblLatin = new SLabel(latin);
```

PRIMITIV FILTER

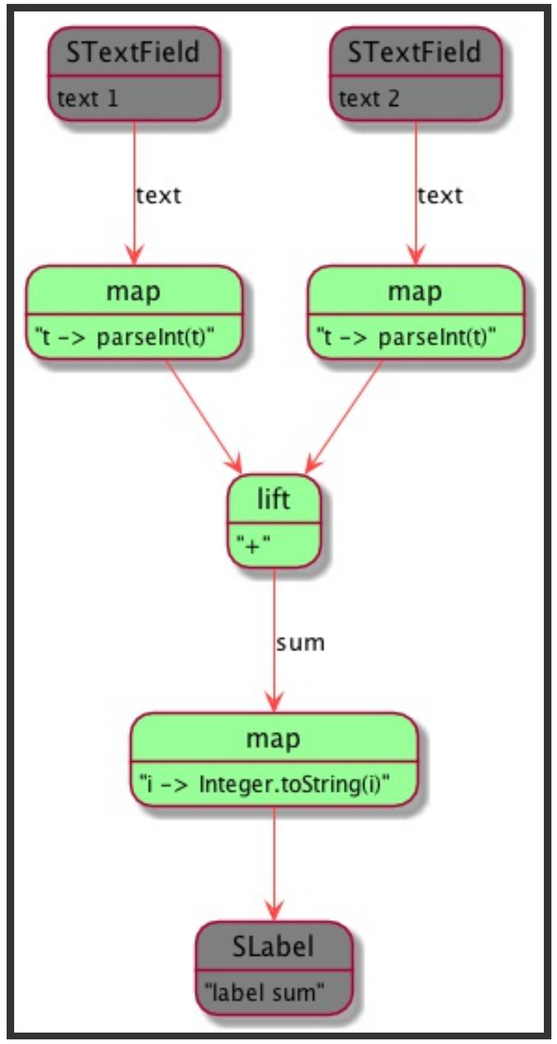
- wenn eine Eigenschaft wahr ist, wird der Wert des Streams in einen anderen Stream gefeuert
- filter :: (a → Bool) → Stream a → Stream a
- Stream<A> filter(final Lambda1<A,Boolean> predicate)


SButton
OK
STextField
sClicked
text
snapshot
(unit, text) -> manipulate(text)
sManipulierterText
filter
(t) -> t.length > 10
sFilteredText
STextField

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)));
```

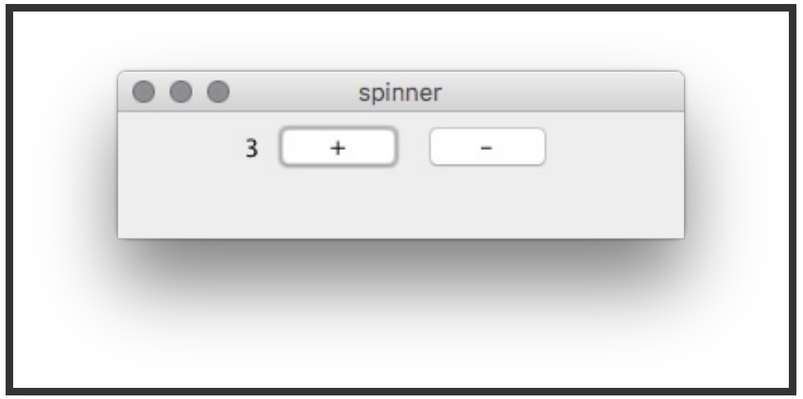
PRIMITIV NEVER STREAM

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

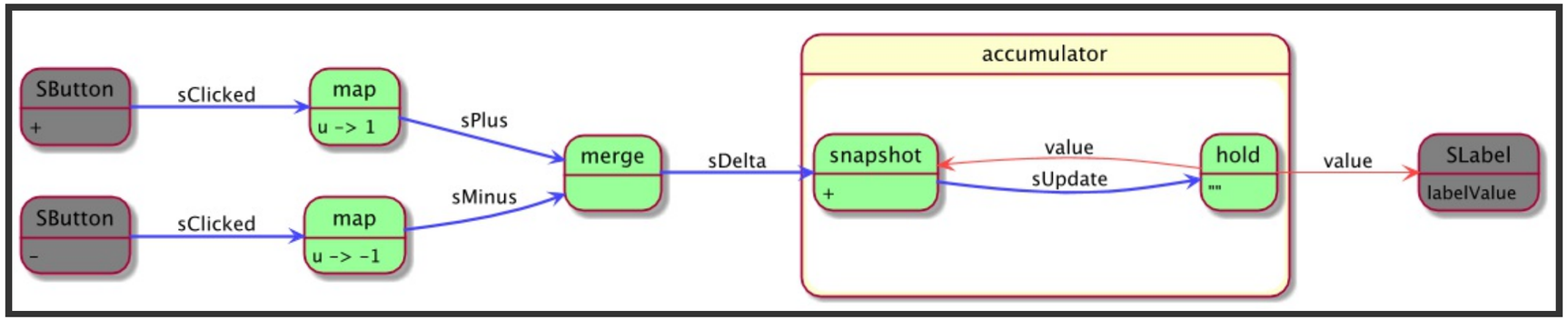
AKKUMULATOR

Ansammlen von Zustandsänderungen

SPINNER



IN FRP


SButton
+
sClicked
map
u -> 1
sPlus
SButton
-
sClicked
map
u -> -1
sMinus
merge
sDelta
accumulator
snapshot
+
value
sUpdate
hold
""
value
SLabel
labelValue

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.

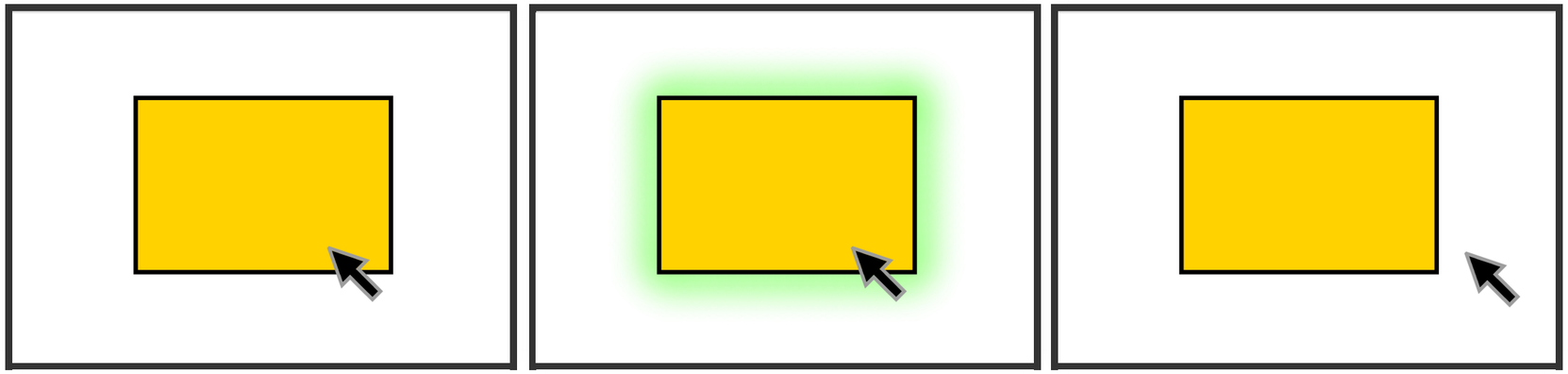
JAVA CODE

```
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));
```

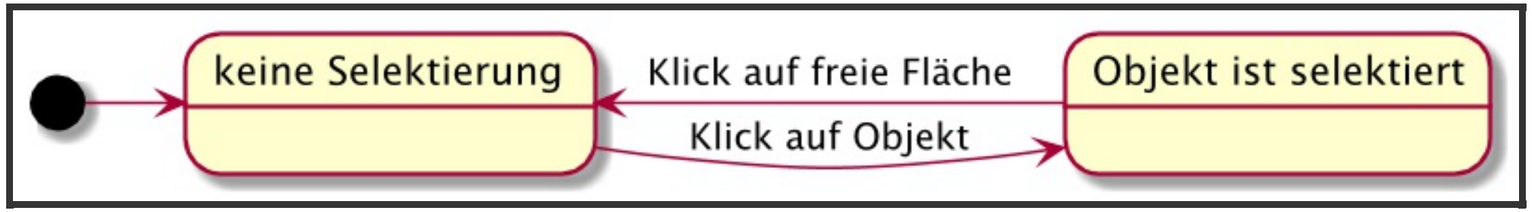
BEISPIEL: ZEICHENPROGRAMM

Selektierung / Deselektierung

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

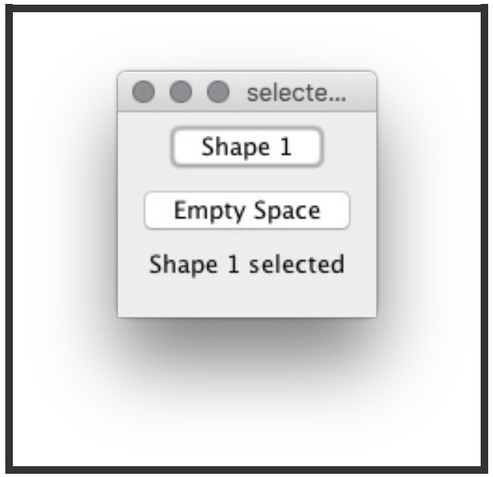


ZUSTANDSMASCHINE

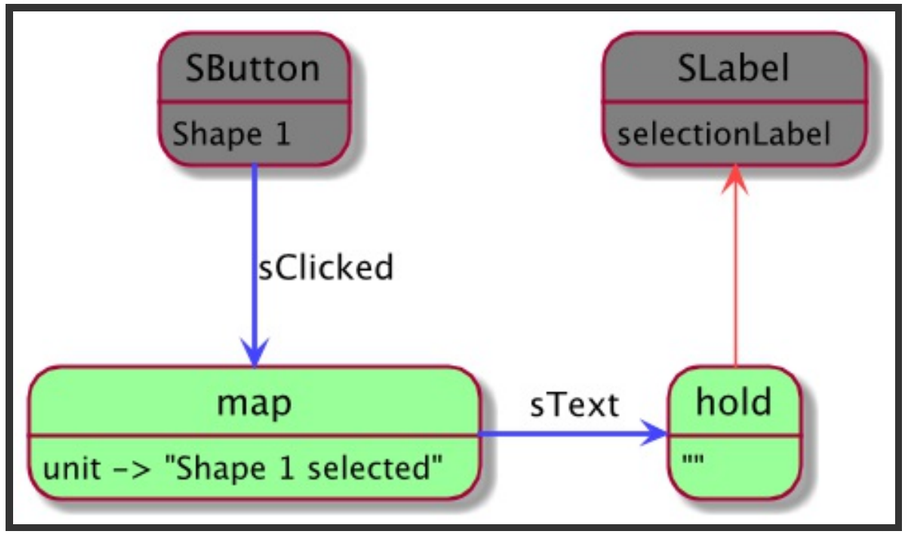


Status	Eingabe	Folgestatus	Ausgabe
Keine Selektierung	Klick auf Objekt	Objekt selektiert	Objekt hervorheben
Keine Selektierung	Klick auf freie Fläche	Keine Selektierung	-
Objekt selektiert	Klick auf Objekt	Objekt selektiert	-
Objekt selektiert	Klick auf freie Fläche	Keine Selektierung	Objekt nicht mehr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

VEREINFACHTES BEISPIEL



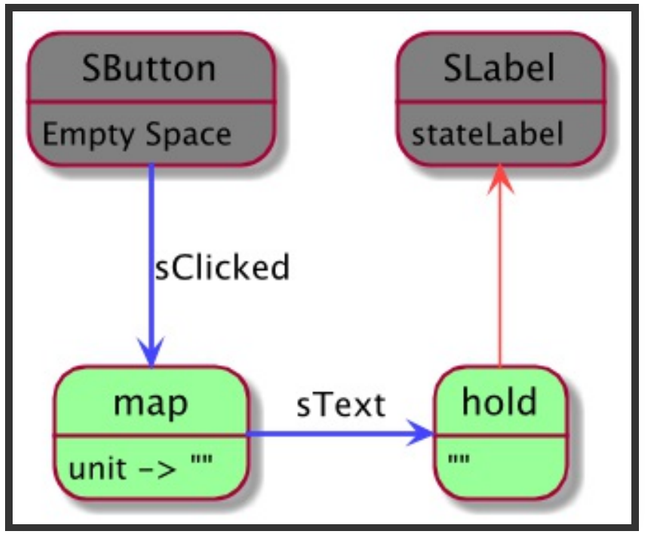
FRP: SELEKTIERUNG



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(""));
```

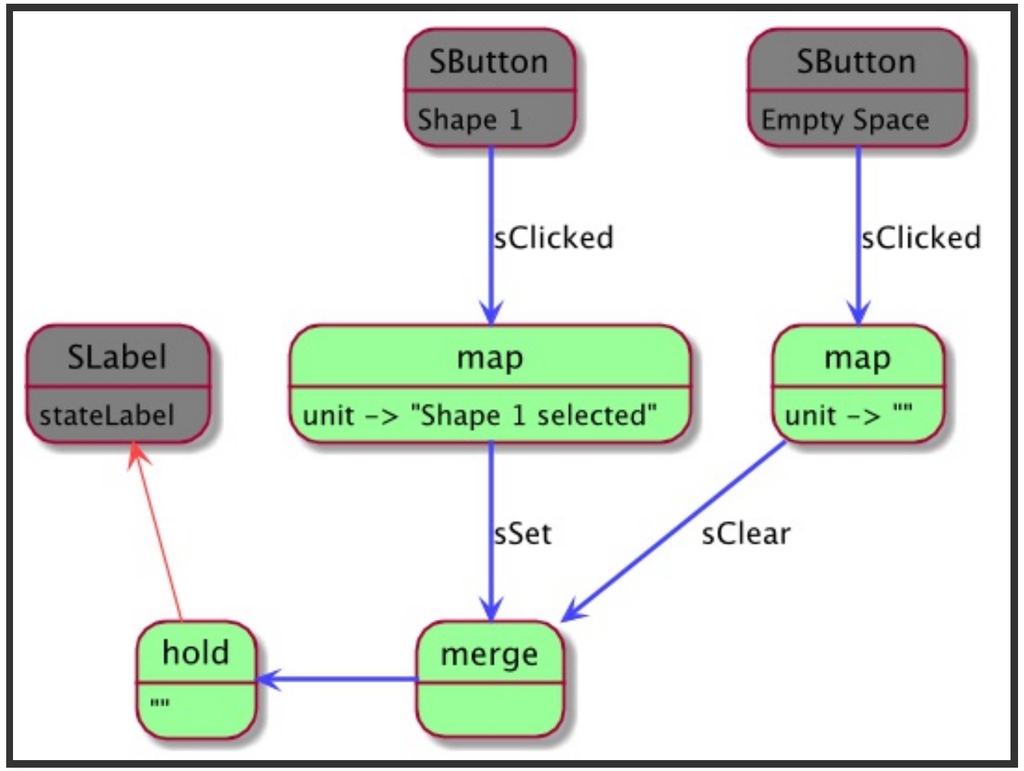
FRP: DESELEKTIERUNG



JAVA: DESELEKTIERUNG

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

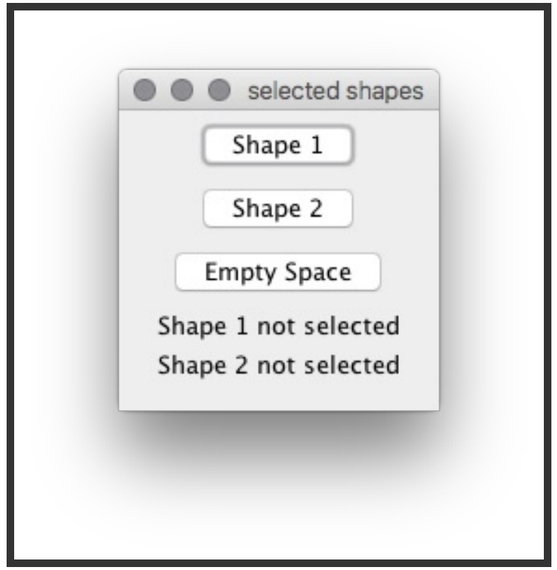
FRP: GESAMT


SButton
Shape 1
SButton
Empty Space
sClicked
sClicked
SLabel
stateLabel
map
unit -> "Shape 1 selected"
map
unit -> ""
sSet
sClear
hold
""
merge

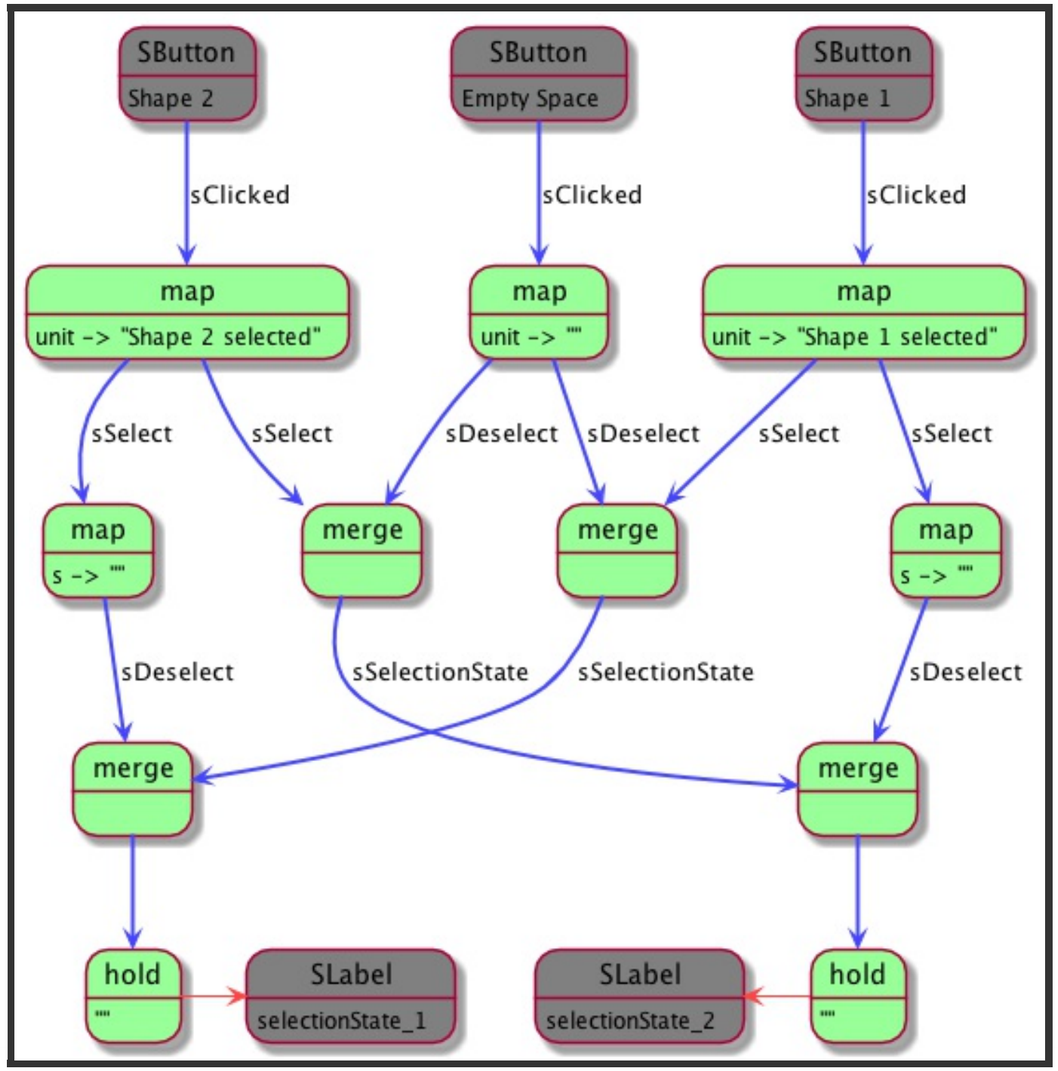
JAVA: GESAMT

```
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(""));
```

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");
```

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?

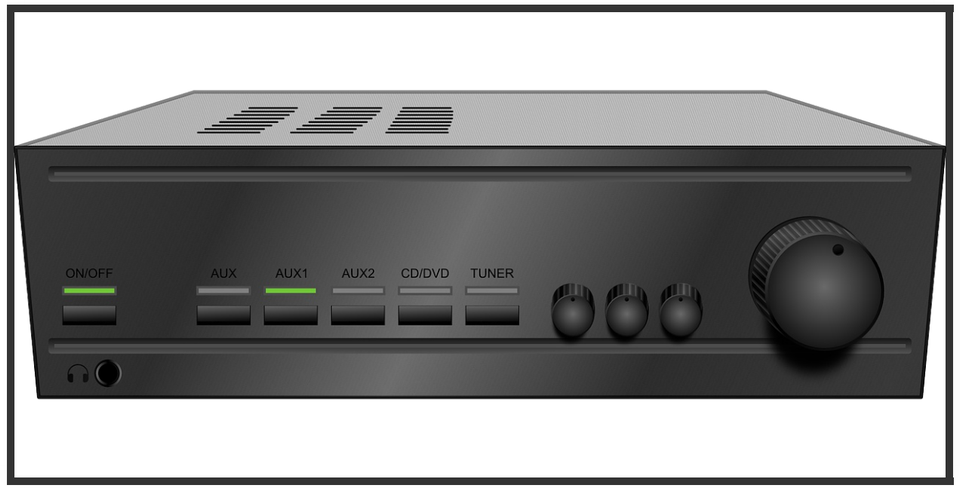
- Snapshot (original)
  - Stream<A> sSnap = stream1.snapshot(cell, (valueStream1, valueCell) → doStuff(valueStream1, valueCell))
- Nachbau mit Map & Sample
  - Stream<A> sSnap = stream1.map(value → doStuff(value, cell.sample()))

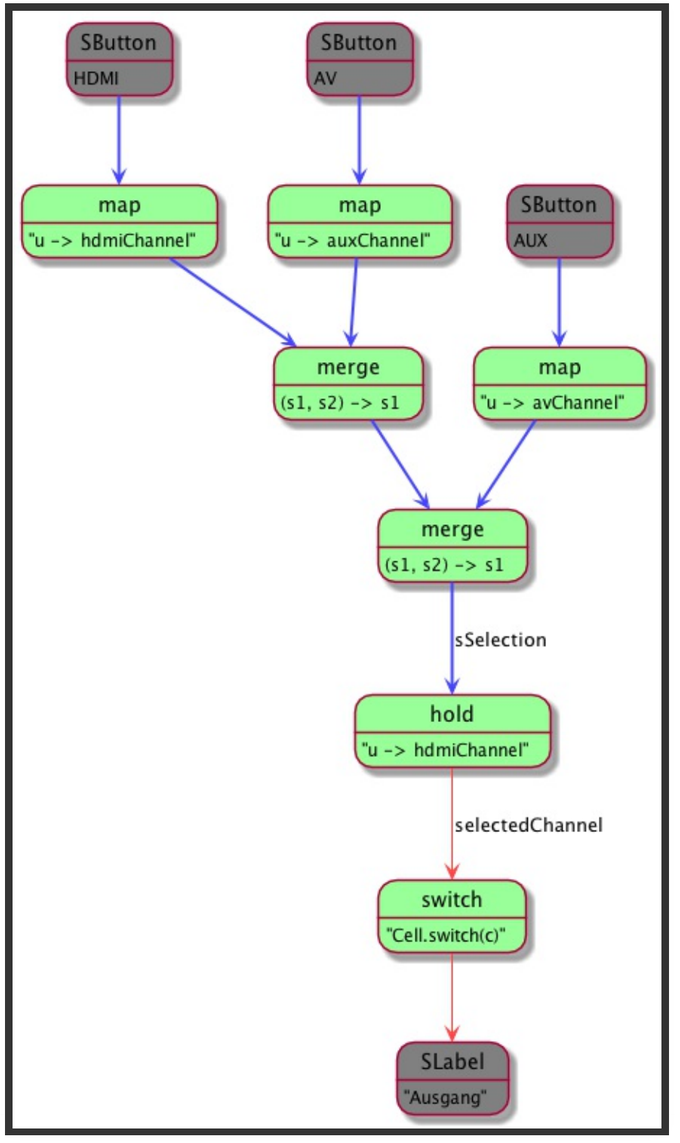
PRIMITIV SWITCH

- 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




SButton
HDMI
SButton
AV
map
"u -> hdmiChannel"
map
"u -> auxChannel"
SButton
AUX
merge
(s1, s2) -> s1
map
"u -> avChannel"
merge
(s1, s2) -> s1
sSelection
hold
"u -> hdmiChannel"
selectedChannel
switch
"Cell.switch(c)"
SLabel
"Ausgang"

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));
```

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


Boggle
WTF

TRANSAKTIONEN

Transaktion = logische Einheit mehrere Programmschritte

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

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))