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))

PLAGE: UNVORHERSEHBARE REIHENFOLGE

BEISPIEL: ZEICHENPROGRAMM

- Klick auf ein Element selektiert dieses
- Klick nicht auf das Element deselektiert dieses
- wenn nichts selektiert ist, ist der Mauszeiger ein Pfeil
- wenn ein Element selektiert ist, ist der Mauszeiger ein *X*

IMPLEMENTIERUNG MIT BEOBACHTER-MUSTER

Beaobachter Interface

```
interface SelectionListener {
  void update(Element e, boolean selected);
}
```

Interface Implementierung

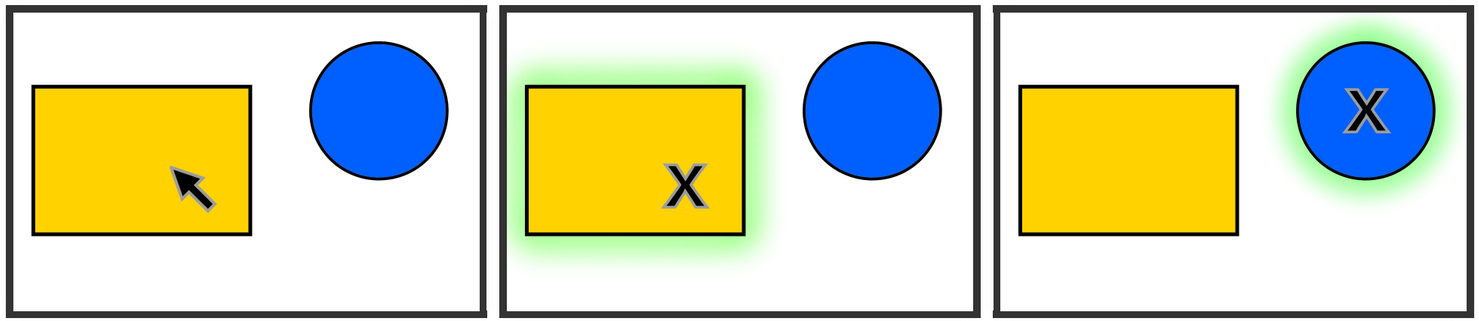
```
class CursorMonitor implements SelectionListener {
  private List<Element> selectedElements = new ArrayList<>();

  public void update(Element e, boolean selected) {
    if(selected) {
      selectedElements.add(e);
    } else {
      selectedElements.remove(e);
    }
    updateCursor();
  }

  private void updateCursor() {
    if(selectedElements.isEmpty()) {
      cursorArrow();
    } else {
      cursorX();
    }
```

TESTFALL

1. nichts ist selektiert
2. Klick auf Rechteck
3. Klick auf Kreis



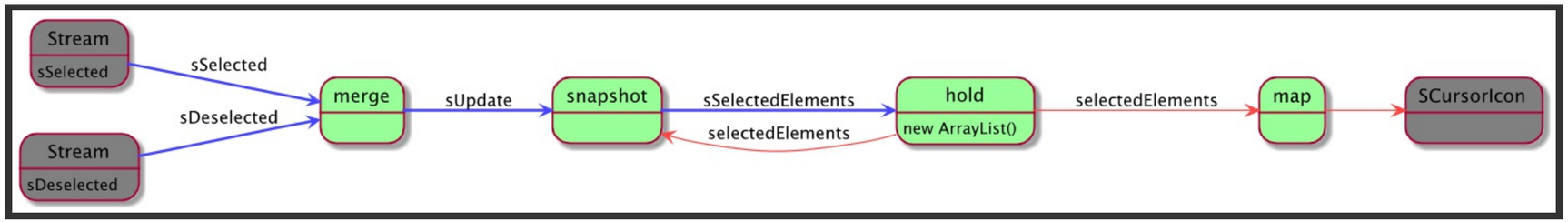
AUSWERTUNG

1. Alternative 1
    1. Rechteck wird selektiert: Cursor wird zum *X*
    2. Kreis wird selektiert: Cursor bleibt *X*
    3. Rechteck wird deselektiert: Cursor bleibt *X*
2. Alternative 2
    1. Rechteck wird selektiert: Cursor wird zum *X*
    2. Rechteck wird deselektiert: Cursor wird zum Pfeil
    3. Kreis wird selektiert: Cursor wird zum *X*

VERBESSERUNGSVORSCHLÄGE

- Timer einbauen, der bei Deselektierung das Update um ein paar ms verzögert
  - (-) Deselektierung ist verzögert
- Transaktion händisch einbauen
  - (-) fehleranfällig
  - (-) sehr aufwändig
- die Reihenfolge der Events garantieren
  - (-) sehr schwierig und umständlich
- Listener mit Prioritäten versehen (sehr umständlich)

IN FRP


Stream
sSelected
Stream
sDeselected
sSelected
sDeselected
merge
sUpdate
snapshot
sSelectedElements
selectedElements
hold
new ArrayList()
selectedElements
map
SCursorIcon

PLAGE: VERPASSTE EREIGNISSE

BEISPIEL: VERBINDUNGSAUFBAU

```
public class Connection {
  void addListener(Listener listener);
  void removeListener(Listener listener);
  void requestConnection();
  void requestDisconnection();

  [...]

  interface Listener {
    void online(Session session);
    void offline(Session session);
  }
}
```

```
Connection connection = new Connection();
//[...]
// andere Clients bekommen auch diese Connection
//[...]
Client client = new Client(connection);
```

AUSWERTUNG

- Alternative 1
  - *Client* registriert sich als Listener
  - Verbindung wird aufgebaut
  - *Client* wird informiert
- Alternative 2
  - Verbindung wird aufgebaut
  - *Client* registriert sich als Listener
  - *Client* wird **nicht** informiert (Verbindung schon da)

VERBESSERUNGSVORSCHLÄGE

- bei Listener-Registrierung den aktuellen Zustand schicken
  - (-) ungewollte Seiteneffekte möglich

IN FRP

- erst erfolgt die Initialisierung (als Transaktion), dann werden Events gefeuert

PLAGE: ZUSTANDSCHAOS

BEISPIEL: VERBINDUNGSABBAU

Erweiterung des Connector.Listener

```
interface Listener {
  void online(Session session);
  void offline(Session session);
  void tearDown(Session session, TearDownCallback callback);

  interface TearDownCallback {
    void tornDown();
  }
}
```

ZUSTÄNDE DES CONNECTORS

- ONLINE
- OFFLINE
- CONNECTING
- TEARING_DOWN

EVENTS DES CONNECTORS

- requestConnection
- requestDisconnection
- Verbindung hergestellt
- Verbindung fehlgeschlagen
- TearDown Bestätigung der Clients

PROBLEM

- viele Events passen nicht zu allen Zuständen
- Lösung
  - Zustand merken
  - 20 Möglichkeiten abbilden (und dabei nichts vergessen)
- Randfälle
  - Verbindung ist schneller aufgebaut als die Methode beendet, die dies gestartet hat
  - TEAR_DOWN der Clients schneller als die Methode, die dies ausgelöst hat

IN FRP

- Randfälle sind automatisch gelöst
- bringt Ordnung in die 20 unterschiedlichen Möglichkeiten

PLAGE: THREAD-PROBLEME

BEISPIEL: VERBINDUNGSHANDHABUNG

- alle Events (requestConnection, requestDisconnection, Verbindung hergestellt, Verbindung fehlgeschlagen, TearDown Bestätigung) und *addListener* und *removeListener* können jetzt parallel kommen
- Race Conditions werden das Programm unbrauchbar machen

LÖSUNG

- in Java z.B. **synchronized** benutzen

```
synchronized void notifyOnline(Session s) {
  for(Listener listener : listeners) {
    listener.online(s);
  }
}
```

VERBESSERTE LÖSUNG

```
void notifyOnline(Session s) {
  List<Listener> copy;
  synchronized (this) {
    copy = listeners.clone();
  }
  for(Listener listener : copy) {
    listener.online(s);
  }
}
```

URSACHE THREAD-PROBLEME

- geteilter veränderbarer Zustand
  - englisch: *shared mutable state*
- in FRP gibt es kein *shared mutable state*

PLAGE: VERGESSENE LISTENER

- *removeListener* wird vergessen
- schwierig den Zeitpunkt für *removeListener* zu finden
- in FRP gibt es keine Listener
  - in der Sprache des Beobachter-Muster: der Lebenszyklus des Beobachters hängt auch vom Subjekt ab

ZEITKONTINUIERLICH

CONAL ELLIOTT

- Erfinder von FRP

- FRP: formale Semantik und zeitkontinuierlich

FORMALE SEMANTIK

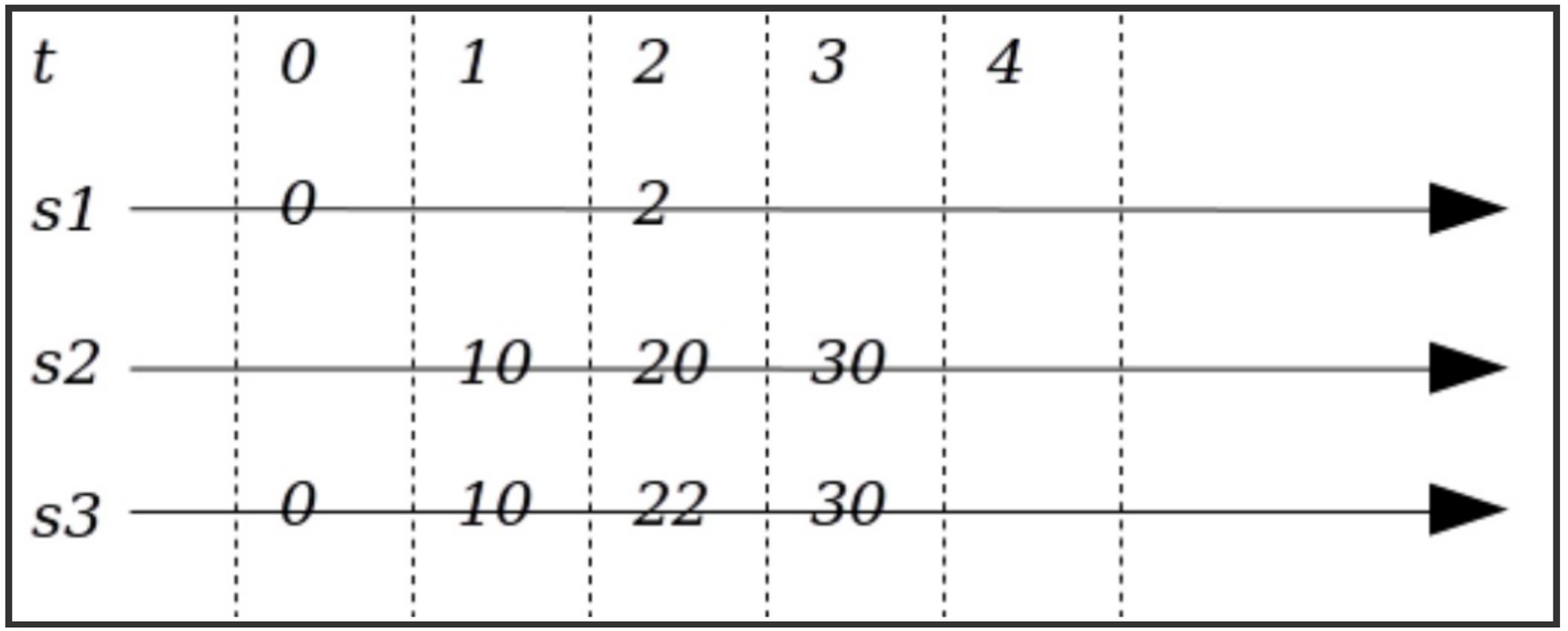
Beispiel von Sodium merge

```
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) { /* [...] */ }
}
```

```
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

```
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

VERTIEFUNG

Beispiel *Drag and Drop* mit Listener und FRP

LITERATUR

Functional Reactive Programming von Stephen Blackheath und Anthony Jones