# Functional Reactiv Programming

# Agenda

1. Was ist funktionales Programmieren
    1. Unterschied zum objektorientierten Programmieren
    2. Schlüsselkonzepte
2. Reaktives Programmieren
    1. Vergleich zum Observerpattern
    2. Verwendung der RxJS Bibliothek
    3. Hot und Cold Observables
3. CycleJS
    1. Konzept
    2. Sources und Sinks
    3. Komponenten
    4. Demonstration

# Funktionales Programmieren

# Vergleich OOP zu FP

```
public class Arbeiter{
    private String name;
    private int gehalt;

    public Arbeiter(String _name, int _gehalt){
```

```
for (angestelltenListe Angestellter: tempAngestellter){
    tempAngestellter.gehaltsErhoehung(200);
}
```

```
        return this.gehalt;
    }

    public String toString(){
        return this.name + " verdient " + this.gehalt;
    }
}
```

# Vergleich OOP zu FP

```
arbeiter = [
  [ "Alice",  10000.0 ],
  [ "Bob", 12500.0 ]
]
```

```
var function gehaelter_erhoehen(arbeiterListe, erhoehung){
  let neueArbeiterListe = arbeiterListe.map(arbeiter =>{
    gehalt_erhoehen(erhoehung);
  })
  return neueArbeiterListe;
}
```

```
var function gehalt_erhoehen(arbeiter, erhoehung){
  let neuerArbeiter =  [arbeiter[0], arbeiter[1] += erhoehung];
  return neuerArbeiter;
}
```

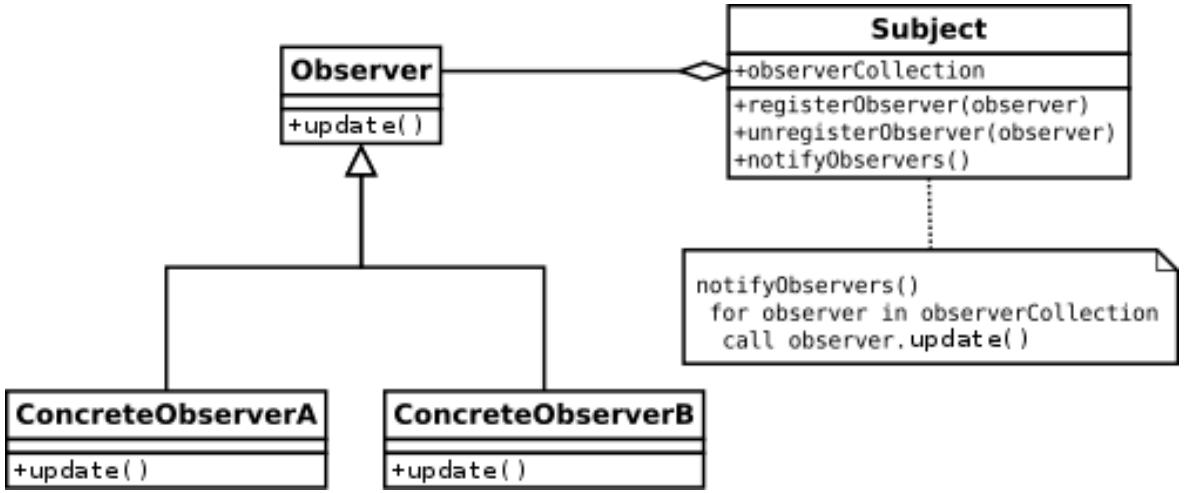
# Schlüsselkonzepte

- Funktionen haben eine einzige Aufgabe
- Daten sind Immutable
- Funktionen sind rein
- Funktionen sind First-Class

# Reaktives Programmieren

# Was ist reaktives Programmieren?

- Unter reaktiven Programmieren versteht man das weitergeben einer Änderung
  - Komponenten ändern Aufgrund von neuen Informationen Zustand
  - Neue Werte müssen nicht gepollt werden
  - Observer-Pattern

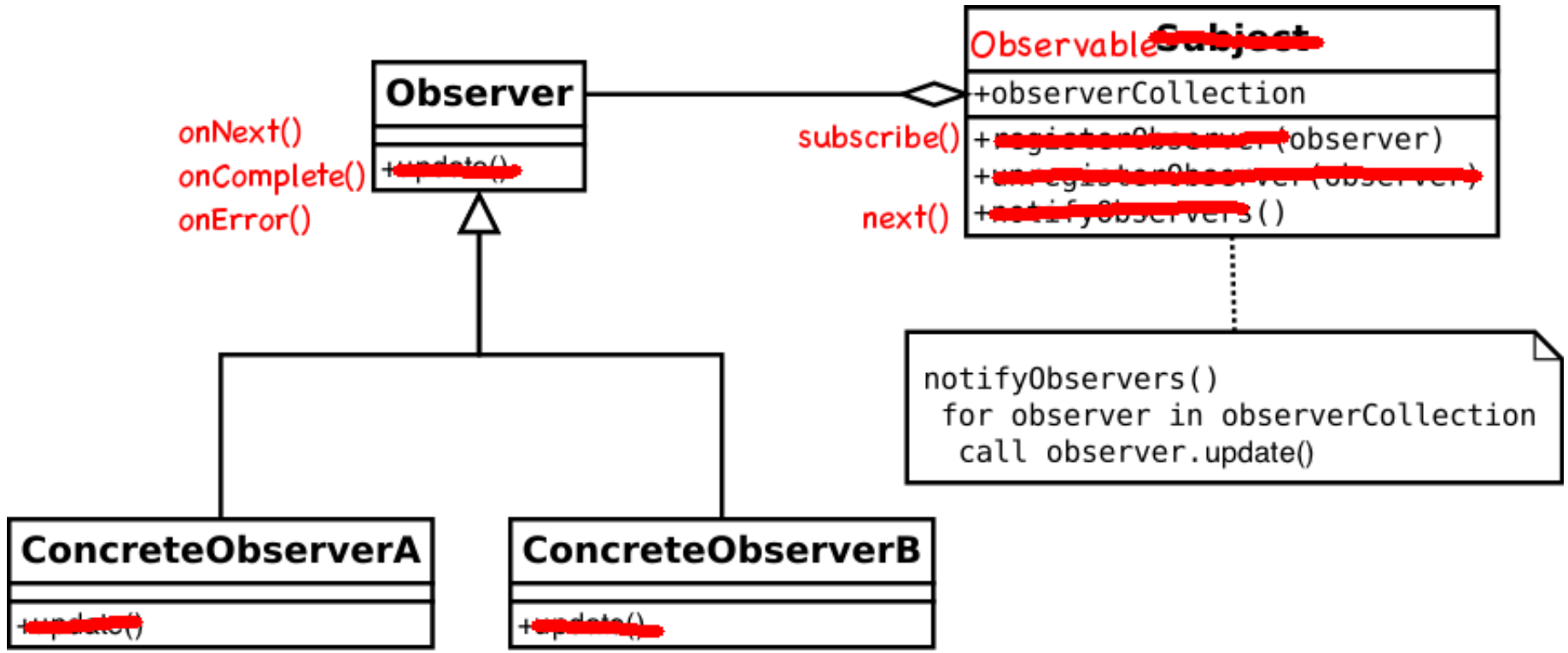


# Reaktives Programmieren mit RxJS

- RxJS greift das Observer-Pattern auf
- Mittelpunkt sind Observables
  - Observables sind Funktionen, welchen ein Producer enthalten
  - Producer können Klickevents, Websockets, Promises, Arrays oder ganz normale Objekte/Primitives sein
- Über subscribe() können neue Observer hinzugefügt werden

```
//create observable that emits click events
const source = Rx.Observable.fromEvent(document, 'click');
//map to string with given random number
const example = source.map(event => `Event time: ${Math.random()}`)
```

# Reaktives Programmieren mit RxJS



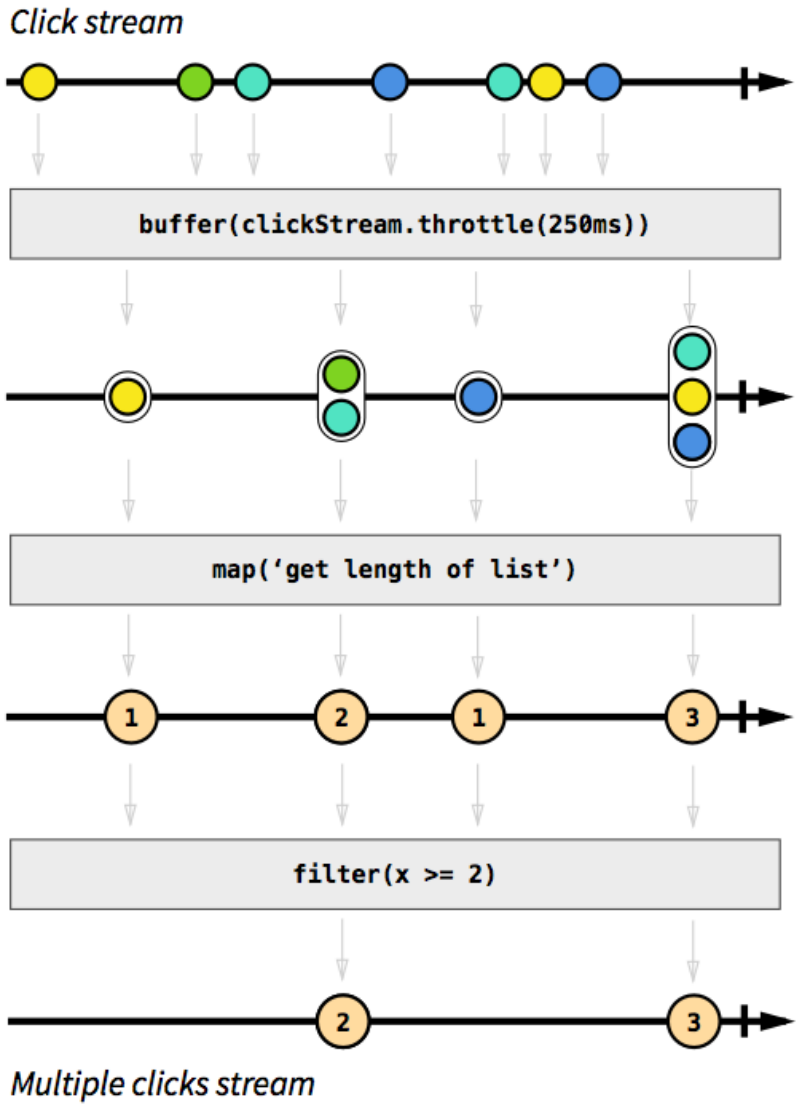
# Nur die halbe Wahrheit

- Über subscribe() wird intern für jeden Observer ein neues Observables erzeugt
- Die Ausgegebenen Werte sind voneinander unabhänig
- Dieses Verhalten wird „Cold Observable“ genannt
- „Hot Observable“ entspricht eher dem normalen Observer-Pattern

# Warum RxJS?

- Stellt Methoden zur Erstellung von Observables breit
- Hauptgrund: Operatoren
- Ein Observable kann je nach Producer viele Werte über einen unbestimmten Zeitpunkt ausgeben
- Man spricht von einem Stream von Werten
- Operatoren können diesen Stream manipulieren

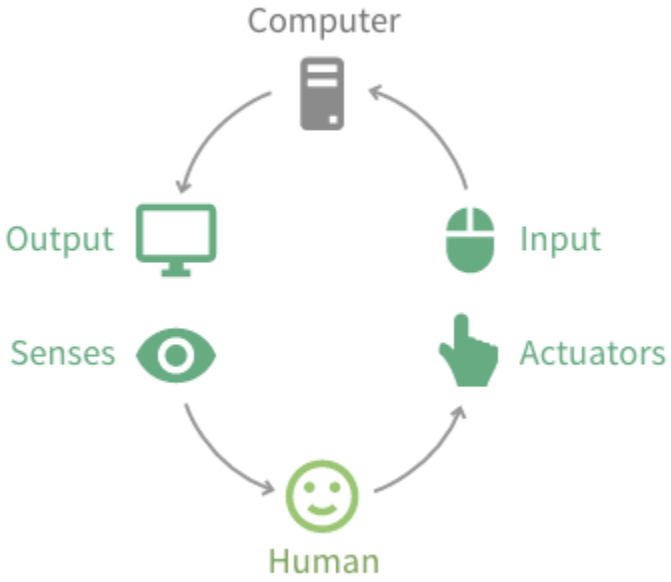
# Operatoren


Click stream
buffer(clickStream.throttle(250ms))
map('get length of list')
1
2
1
3
filter(x >= 2)
2
3
Multiple clicks stream

# CycleJS

# Konzept von CycleJS

- CycleJS ist ein funktionales reaktives Framework, geschrieben in Javascript
- Hauptgedanke: “What if the User were a function?“

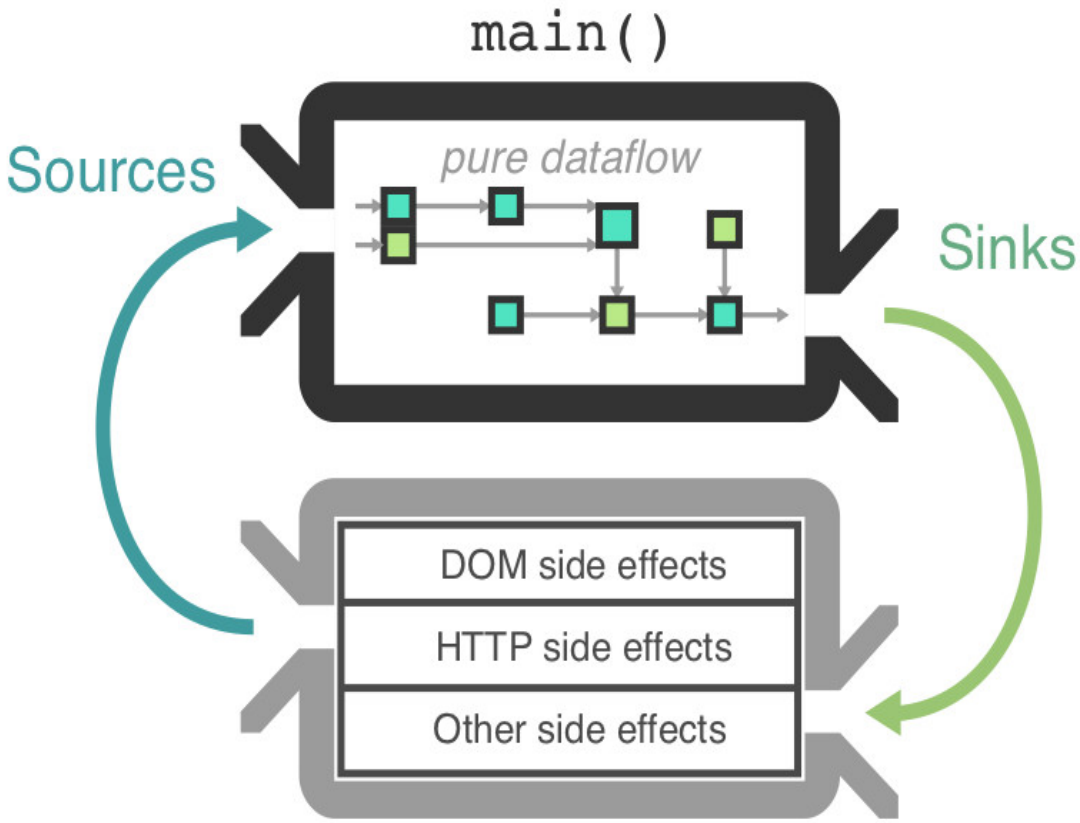


# Konzept von CycleJS

- CycleJS löst circuläre Abhängigkeit
- Aus UserFunction() wird main()
- Aus DisplayFunction() wird Driver Objekt
- Beide werden durch eine run() Funktion verbunden

```
var UserOutPut = UserFunction(DisplayFunction(UserOutput));
```

# Sources and Sinks



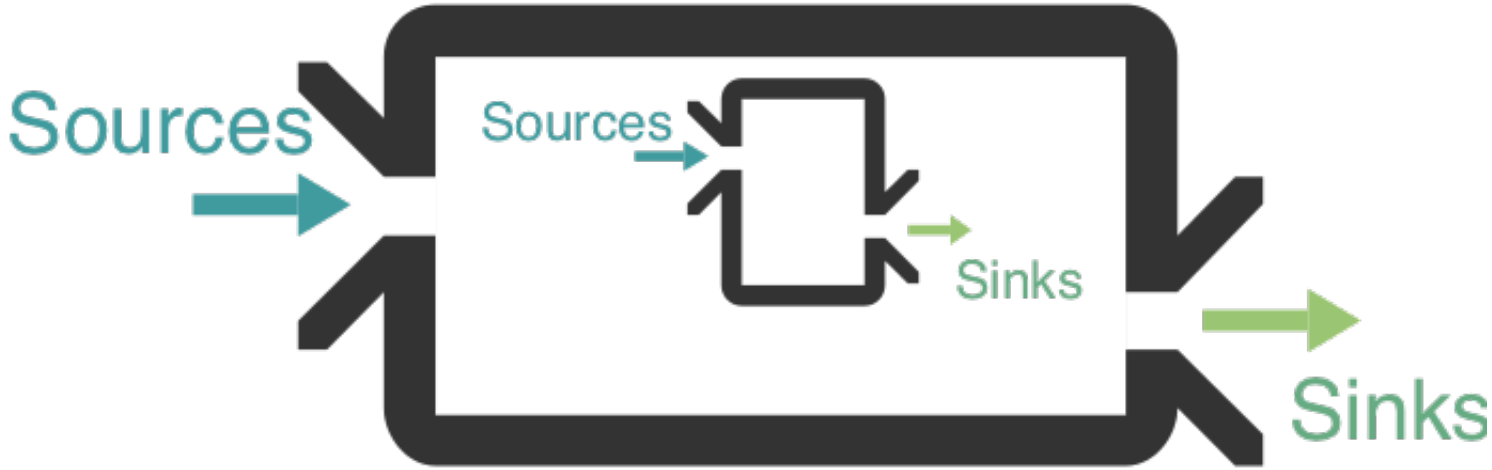

```
import xs from 'xstream';
import {run} from '@cycle/run';
import {div, input, p, makeDOMDriver} from '@cycle/dom';

function main(sources) {
  const sinks$ = {
   DOM: sources.DOM.select('input').events('change')
      .map(ev => ev.target.checked)
      .startWith(false)
      .map(toggled =>
        div([
          input({attrs: {type: 'checkbox'}}), 'Toggle me',
          p(toggled ? 'ON' : 'off')
        ])
      )
  };
  return sinks$;
}

run(main, {
  DOM: makeDOMDriver('#app')
});
```

# Driver

- DOM-Driver, HTTP-Driver, Websocket-Driver, GraphQL-Driver
- Alle Kapselt Seiteneffekte von der main() Funktion ab
- Die main() transformiert somit nur Daten
- Somit kann auch User nur mit Daten interagieren

# Erstellung Komponenten

- Jede CycleJS main() Funktion ist gleichzeitig auch eine Komponente
- Komponenten erhalten ein Sourceobjekt und geben ein Sinksobjekt herraus => selbe Struktur wie eigentliche main() Funktion
- Komponenten können zusätzlich „props“ bekommen, um Funktion und aussehen zu definieren



# Software demonstration