Cycle.js

Seminararbeit WS 16/17

Maximilian Vogel
Fakultät für Informatik und Wirtschaftsinformatik
Hochschule Karlsruhe
Betreuer: Prof. Dr. Thomas Fuchß

10. Dezember 2016

Diese Seminararbeit befasst sich mit dem JavaScript-Framework Cycle.js. Es wird die Funktionsweise von Cycle.js anhand einiger Beispiele aufgearbeitet. Dabei wird der Schwerpunkt auf die Architektur und den clientseitigen Datenfluss gelegt. Es wird die Behandlung von Seiteneffekten sowie der Modularisierung von komplexeren Anwendungen beschrieben. Zudem wird Cycle.js anderen Framworks gegenüber gestellt.

1 Motivation

Wir bewegen uns heute mehr denn je im Internet und interagieren mit vielen unterschiedlichen Webseiten. Ob bei einer Bestellung im Onlineshop oder einer Suche über die Suchmaschine, wir möchten, dass Webseiten direkt auf unsere Eingaben reagieren und uns Ergebnisse schnell bereitgestellt werden. Wir achten bei der Auswahl von Dienstleistungen im Internet nicht nur auf den reinen Nutzen, den wir von einem Angebot erhalten können, sondern auch wie gut diese Webseiten aussehen oder ob sie schnell reagieren. Nach einer Studie der ARD/ZDF gehen 49% der deutschen Bevölkerung täglich über das Smartphone ins Internet [BF16]. Das stellt neue Herausforderungen an die Entwicklung von Webseiten. Gerade bei Webseiten, die auf den Zugriff von mobilen Endgeräten optimiert werden, muss ein Augenmerk auf eine gute Performanz gelegt werden. Webseiten bei denen der Nutzer nach einer Aktion immer wieder auf eine neue Seite geleitet wird und lange warten muss sie geladen ist, entsprechen nicht mehr unserer Vorstellung von einer gut bedienbaren und zeitgemäßen Anwendung. Sich reaktiv verhaltende Programme sind für uns inzwischen selbstverständlich und die meisten großen Webseiten sind dementsprechend umgesetzt. Diese Seminararbeit stellt das JavaScript-Framework Cycle.js vor,

welches es möglich machen soll, reaktive und funktionale Benutzeroberflächen zu erstellen.

1.1 Programmbeispiele

Auf alle in dieser Ausarbeitung vorgestellten Programmbeispiele, sowie auf weitere mit Cycle.js entwickelte Anwendungen, kann über das folgenden Github Repository zugegriffen werden:

`https://github.com/muetzerich/explore-cyclejs`

2 Grundlagen

In den nachfolgenden Abschnitten werden einige Grundlagen erläutert, die zum Verständnis des Frameworks Cycle.js notwendig sind. Es wird zunächst eine Einführung in die Funktionale Programmierung mit ECMAScript 6 gegeben. Anschließend wird die Mensch-Maschine Kommunikation, sowie einige Grundlagen der reaktiven Programmierung behandelt.

2.1 ECMAScript 6

ECMAScript 6[1] wurde im Juni 2016 von der privaten non-Profit Organisation Ecma International, als Nachfolger von ECMAScript 5, welches auch von JavaScript implementiert wird, vorgestellt. Ecma International entwickelte unter anderem Standards und Normen diverser Programmiersprachen und technische Prinzipien wie zum Beispiel JSON, C# oder den FAT12/16 Standard.[2] Da ES6 aktuell noch nicht bzw. nicht vollständig von allen modernen Browsern unterstützt wird, muss die Sprache mithilfe eines Transpilers wie zum Beispiel Babel oder Traceur von ES6 zu ES5 übersetzt werden (siehe auch Abschnitt 3.2).

Die folgenden Abschnitte sollen keinen vollständigen Überblick über alle neuen Funktionen von ES6 geben, sondern es werden ausgewählte Sprachbestandteile vorgestellt, die gerade im Hinblick auf die funktionale Programmierung und zum Verständnis der Cycle.js Programmbeispiele notwendig sind.

2.1.1 Block-Scoped Variablen und Konstanten

Bisher konnten Variablen in JavaScript nur mithilfe des `var` Schlüsselworts deklariert werden. Die Sichtbarkeit der Variablen wurde dadurch immer durch die nächsthöhere Funktion eingegrenzt. Mithilfe des `let` Schlüsselworts lassen sich nun auch Variablen deklarieren, die nur im nächsthöheren Block sichtbar sind.

[1] Der ECMA6 Standard ist unter `http://www.ecma-international.org/ecma-262/6.0/` zu finden
[2] Weitere Ecma Standards: `http://www.ecma-international.org/publications/standards/Standard.htm`

```
1 for (let i = 0; i < a.length; i++) {
2     let scoped = a[i] //scoped und i sind nur innerhalb des for-Blocks
          sichtbar
3 }
```

Listing 1: `let` Schlüsselwort in ES6

Mithilfe des `const` Schlüsselworts lassen sich immutable Variablen erstellen, denen zu einem späteren Zeitpunkt kein neuer Wert zugewiesen werden kann. Das gilt aber nur für die Variable selbst. Ist der Inhalt zum Beispiel ein Objekt, so ist dieses trotzdem veränderlich.

```
1 const SPEED_OF_LIGHT = 299792,458
```

Listing 2: Konstanten in ES6

2.1.2 Arrow Functions

Mithilfe von Arrow Functions lassen sich nun Funktionen in kürzerer Schreibweise darstellen. Zudem wird das `this` der Arrow Funktion auf den Wert vom äußeren Kontext gesetzt. Besteht die Funktion aus einem Ausdruck, ist kein `return` Statement notwendig, da die Rückgabe des Ergebnisses implizit durchgeführt wird.

```
1 //ES6
2 result  = data.map(t => t + 42)
3 //ES5
4 result  = data.map(function (t) { return t + 42; })
```

Listing 3: Arrow-Functions in ES6

2.2 Funktionale Programmierung

JavaScript unterstützt das Programmieren in verschiedenen Paradigmen wie zum Beispiel der Imperativen Programmierung, Objektorientierte Programmierung und Funktionale Programmierung. Nach Hudson [Hud89, 3] spricht man von Deklarativer Programmierung wenn der Zustand eines Programms durch eine Sequenz von Anweisungen verändert wird. Diese Veränderungen des Programmzustands werden als Seiteneffekte (Side Effects) bezeichnet. Im Gegensatz dazu steht der Ansatz der Deklarativen Programmierung, bei dem das Programm keinen expliziten Zustand besitzt, sondern aus einer Reihe von Ausdrücken besteht[Hud89, 3]. In der Funktionale Programmierung wird das Programm durch Funktionen dargestellt[Tor96, 1]. Somit sind funktionale Sprachen wie die deklarativen Sprachen, nur dass die Ausdrücke durch

Funktionen ersetzt werden. Sie ist somit eine Untergruppe der Deklarativen Programmiersprachen. Erzeugt eine Funktion keine Seiteneffekte und gibt sie bei einem bestimmten Eingabewert immer das gleiche Ergebnis zurück, so kann man sie als *pure Function* bezeichnen.

```
1 //pure function ohne Seiteneffekte
2 (x,y) => {x+y}
3
4 //impure function mit Seiteneffekt
5 let array = [1,2,3]
6 (x) => {
7   x.pop()
8   return input[0] + 1
9 }
```

Listing 4: Pure vs. impure Funktionen in Javascript

Neben Pure Functions spielen Funktionen Höherer Ordnung eine große Rolle in der Welt der Funktionalen Programmierung. Diese Funktionen nehmen eine andere Funktion als Parameter entgegen oder geben sie als Rückgabewert zurück. Ein oft verwendetes Beispiel für eine Funktion höherer Ordnung ist die `map()` Methode, welche auf jedes Element des Arrays eine bereitgestellte Funktion anwendet und anschießend ein neues Array als Ergebnis zurückliefert [Con16].

```
1 //Array mit verschiedenen Zahlen
2 const numbers = [0,1,2,3]
3
4 //Funktion die auf den Input +1 addiert
5 const add = x => x + 1
6
7 //Aufruf der map-Methode mit der add() Funktion als Parameter
8 let result = numbers.map(add)
9
10 //result = [1,2,3,4]
```

Listing 5: Anwendung der `map()` Methode

2.3 Mensch-Maschine Kommunikation

André Staltz, der Entwickler von Cycle.js, beschreibt die Mensch-Maschine Kommunikation als einen Dialog, welcher ein ständiger Austausch zwischen zwei Seiten ist[Sta16d]. Die eine Seite ist dabei der Mensch und die andere Seite der Computer. Der Computer stellt Informationen dar, welche der Mensch wahrnimmt. Darauf reagiert der Nutzer und führt eine Aktion auf dem

Computer aus. Der Computer kann als eine Abstraktion einer Cycle.js Anwendung gesehen werden.

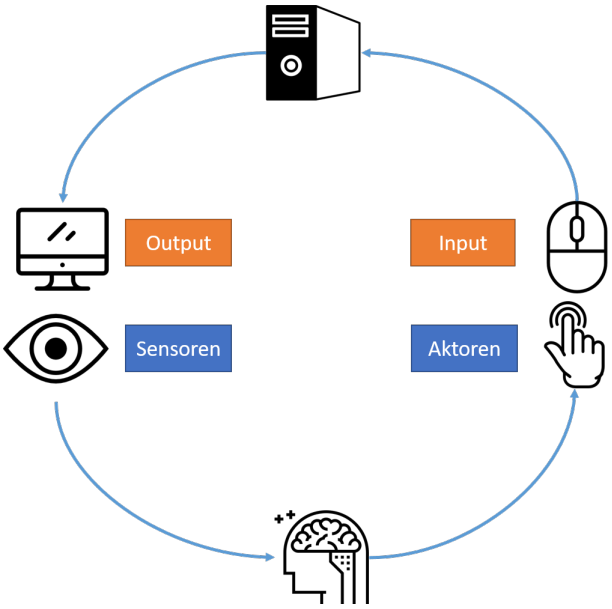

Abbildung 1: Mensch-Maschine Kommuikation als Zyklus nach [Sta16d]

Diese zyklische Kommunikation lässt sich mit den zwei nachfolgenden Funktionen beschreiben. Einer Funktion für den Mensch:

$$y = human(x)$$

und einer Funktion für den Computer:

$$x = computer(y)$$

Die Hauptaufgabe des Cycle.js Frameworks besteht darin, diesen zyklischen Ablauf technisch umzusetzen. Der ständige Fluss von Informationen wird mithilfe von Streams (mehr dazu im nächsten Abschnitt) umgesetzt.

2.4 Reaktive Programmierung

Das Reaktive Manifest, welches im September 2014 veröffentlicht wurde, stellt folgende Anforderungen an ein reaktives System. Die Systeme sollen stets antwortbereit, widerstandsfähig, elastisch und nachrichtenorientiert sein[JB14]. Das Frontend sollte möglichst viele dieser Anforderungen erfüllen, damit das Gesamtsystem als reaktiv bezeichnet werden kann. Im folgenden wird der Unterschied zwischen dem proaktiven und dem reaktiven Ansatz beschrieben und wie dieses Konzept mithilfe von Streams in Cycle.js umgesetzt wird.

2.4.1 proaktiver vs. reaktiver Ansatz

In diesem Beispiel wird bei einem Klick auf einen Button eine Funktion aufgerufen, die eine Ausgabe auf der Konsole durchführt. Die Benutzeroberfläche mit dem Button ist in diesem Fall proaktiv, da sie verantwortlich für den Aufruf der `hello()`-Funktion ist. Die Funktion ist passiv, da sie nichts von der Existenz des Buttons weiß und anderen Komponenten erlaubt, sie aufzurufen oder den eigenen Zustand zu verändern. Dieser Ansatz bringt aber einige Probleme mit sich: zum einen herrscht eine starke Kopplung zwischen der Darstellung und der Logik. Wird etwa der Name der Funktion geändert oder die Funktion soll ausgetauscht werden, so muss an beiden Stellen eine Anpassung vorgenommen werden. Ein weiteres Problem des proaktiven Ansatz, ist die fehlende Skalierbarkeit. Es ist schwierig mithilfe des `onClick()`-Events mehrere Funktionen aufzurufen oder dynamisch den Aufrufer festzulegen.

```
1 function hello() {
2     console.log('Hallo')
3 }
4
5 <button onClick={hello()} class="myButton">
6     Button
7 </button>
```

Listing 6: proaktiver Ansatz

Eine Alternative dazu ist der reaktive Ansatz, der hier mithilfe von Eventlistenern umgesetzt wurde. Die `hello()`-Funktion ist bei diesem Ansatz selbst für ihren Zustand verantwortlich und entscheidet selbst, wann das Event von dem Button benötigt wird. Die Darstellung ist hier von der Logik entkoppelt und hat kein Wissen über den Zustand des Programms. Diese strikte Trennung zwischen der Anwendungslogik und der Darstellung der Daten, ist eines der Kernkonzepte von Cycle.js. Das Vorgehen bringt auch eine bessere Skalierbarkeit und Wartbarkeit mit sich, da problemlos mehrere Eventlistener auf den Button registriert werden können. Zudem kann die `hello()`-Funktion problemlos umbenannt oder verschoben werden.

```
1 function hello() {
2     mybutton.addEventListener('click', () => {
3         console.log('Hallo!')
4     })
5 }
6
7 <button class="myButton">
8     Button
9 </button>
```

Listing 7: reaktiver Ansatz

2.4.2 Reaktive Programmierung mit xstream

Reaktive Programmierung kann mit den gerade verwendeten Eventlistenern umsetzt werden. Diese haben aber keinen großen Funktionsumfang, weswegen üblicherweise eine Stream-Bibliothek verwendet wird. Cycle.js unterstützt eine Reihe verschiedener Stream-Bibliotheken: zum Beispiel rxjs[3] oder xtream[4]. In den folgenden Abschnitten wird das Konzept, sowie die praktische Umsetzung der Reaktiven Programmierung, mithilfe xstream vorgestellt. Zu Beginn der Entwicklung von Cycle.js wurde zunächst die Bibliothek Rxjs verwendet. Anfang 2016 die Bibliothek xstream vorgestellt, welche auf die Verwendung von Cycle.js zugeschnitten ist.

Spricht man von Reaktiver Programmierung, so steht das Konzept von asynchronen Streams im Mittelpunkt. Unter einem Stream versteht man die Abfolge von Events, welche Daten enthalten können. Events können z.B. Benutzerinteraktionen über eine Benutzungsoberfläche oder ein intern durch das Programm ausgelöstes Ereignis sein. Auf diesem Stream von Events können nun verschiedene Operationen wie `map()`, `filter()` oder `repeat()` angewandt werden. Um auf die Events reagieren zu können, kann man sich am Datenbus als Observer anmelden und erhält so die Daten auf dem Stream.

Abbildung 2: Daten auf einem Stream

Auf dem zuvor dargestellten Stream werden nun zum Beispiel die Operationen `filter()` und `map()` ausgeführt. Die Operation `filter()` nimmt eine Funktion entgegen und wendet diese auf den Stream an. In diesem Fall werden nur Zahlen die größer als 4 sind akzeptiert. Da alle Operatoren von xstream pure sind, wird das Ergebnis aber nicht auf dem ursprünglichen Stream abgespeichert, sondern es wird ein neuer Stream erstellt.

```
1 ----Zeitraum---->
2  [1..3..5....6]
3    .filter( n => n > 4 )
4    .map(n => console.log(n + 1))
```

Listing 8: Operationen auf dem Stream

[3]Webseite der RxJs-Bibliothek: `https://github.com/Reactive-Extensions/RxJS`
[4]Github-Seite der xstream-Bibliothek: `https://github.com/staltz/xstream`

Die `map` Operation gibt den neuen Stream dann auf der Konsole aus.

```
1 > [6]
2 > [7]
```

Listing 9: Operationen auf dem Stream

Konvention: Wird in den folgenden Programmausschnitten ein Stream in einer Variable abgespeichert, so wird dies durch die Verwendung des Suffix $ gekennzeichnet. Soll z.B. in der Variable `inputStream` ein Stream gespeichert werden, so verwenden wir die folgende Schreibweise `inputStream$`. Dies soll die Lesbarkeit der Programme, sowie die Analyse des Datenflusses erleichtern.

3 Installation und Konfiguration

In den folgenden Abschnitten werden die ersten Schritte zur Erstellung einer lauffähigen Cycle.js Anwendung vorgestellt. Dabei wird zuerst die Installation der notwendigen Pakete, das Modulsystem und der Transpiler Babel vorgestellt. Anschließend wird gezeigt, wie neue Pakete mithilfe des Node Package Manager (npm) installiert werden können.

3.1 Installation

Um Cycle.js Applikationen erstellen zu können, ist die Installation der JavaScript-Laufzeitumgebung Node.js notwendig. Die Plattform ist unter anderem für die Betriebssysteme Linux, Windows und macOS verfügbar[5]. Neben der manuellen Installation der notwendigen Cycle.js Module, gibt es die Möglichkeit das Command-Line-Tool (CLI) *create-cycle-app* zu verwenden. Damit kann man sich ohne großen Aufwand die Grundstruktur, sowie die notwendigen Abhängigkeiten für eine lauffähige Cycle.js Anwendung automatisch erstellen lassen.

Über die Konsole wird nun zuerst das CLI-Tool installiert. Dazu wird der Paketmanager npm von Node.js verwendet.

```
1 $ npm install create-cycle-app
```

Anschließen kann in einem beliebigen Arbeitsverzeichnis eine neue Anwendung erstellt werden. Im untenstehenden Beispiel wird eine Anwendung mit dem Namen *explore-cyclejs* erstellt. In der Konsole muss man nun zuerst die Sprache und das Modulsystem wählen. Für die folgenden Anwendungen wird ES6 und Webpack verwendet. Anschließend kann man zwischen verschiedenen Stream-Bibliotheken wählen. Für diese Beispiele wird xstream ausgewählt.

[5]Node.js kann unter `https://nodejs.org/en/download/` heruntergeladen werden.

```
$ create-cycle-app explore-cyclejs
```

Nun kann man in das neu erstellte Verzeichnis wechseln und die Anwendung konfigurieren. Mit den folgenden Befehlen werden alle Abhängigkeiten von dem CLI-Tool in das eigene Projekt übertragen und installiert.

```
$ npm run take-off-training-wheels
$ npm install
```

Anschließend kann die entweder Applikation lokal gestartet werden oder für die produktive Auslieferung gebaut werden.

```
$ npm start
$ npm run build
```

3.2 Weback und Babel

Webpack ist ein Modulsystem, welches die Hauptaufgabe besitzt, JavaScript-Dateien sowie andere Ressourcen, die im Browser verwendet werden, zu transformieren und zu binden[Kop16, 1]. Die Build-Prozesse werden in den JavaScript-Dateien im Ordner *.script* konfiguriert. Untenstehend ist die leicht vereinfachte Konfigurationsdatei für den start-Prozess dargestellt. Die Konfiguration wird in einem Objekt festgelegt. In der Eigenschaft *entry* wird der Einstiegspunkt für das Bundle festgelegt. Unter *output* wird die Ausgabedatei konfiguriert. Wie im Abschnitt 2.1 beschrieben, muss aus Kompatibilitätsgründen ES6 zu ES5 transpiliert werden. Dies kann mithilfe von Babel umgesetzt werden. Dazu wird Babel in *module* konfiguriert. Anschließend wird Webpack mit diesen Konfigurationen initialisiert und der Entwicklungsserver gestartet.

```
var config = {
  entry: [
    './src/'
  ],
  output: {filename: 'bundle.js' },
  module: {
    loaders: [{
        loader: 'babel',
        query: { presets: ['es2015'] }}
    ]
  }
```

```
12 }
13
14 var compiler = webpack(config)
15 var server = new WebpackDevServer(compiler)
```

Listing 10: Aufbau einer Webpack Konfiguration

3.3 Paketverwaltung

Um die bereits im Projekt installierten Abhängigkeiten verwalten zu können und neue npm-Pakete[6] hinzuzufügen, wird die JSON-Datei *package.json* im Stammverzeichnis verwendet. Darin können sowohl die Abhängigkeiten die nur für die Entwicklung benötigt werden *(devDependencies)*, als auch die Pakete welche bei einem Release mit ausgeliefert werden (*dependencies*), verwaltet werden.

Neue Pakete können mit dem folgenden npm-Befehl installiert. Mithilfe des Parameters `-save` wird die Abhängigkeit in der Datei *package.json* persistiert. Im nachfolgenden Beispiel wird Bibliothek ramda installiert.

```
1   $ npm install ramda --save
```

4 Cycle.js Anwendung

Im Folgenden wird der Grundaufbau von Cycle.js Applikationen anhand eines Beispiels beschrieben. Es wird zuerst die Architektur sowie der Datenfluss eingegangen. Dabei werden die verschiedenen Komponenten des Frameworks vorgestellt, sowie wie die Kommunikation zwischen diesen Teilen abläuft. Anschließend werden die Cycle.js Funktionen vorgestellt, welche die Anwendungslogik beinhalten. Zum Schluss wird darauf eingegangen, wie das Framwork intern funktioniert.

4.1 Architektur

Das Framework bietet eine Architektur zum Erstellen von reaktiven Benutzeroberflächen auf dem Client. Eine Cycle.js Applikation besteht grundsätzlich aus drei wichtigen Bestandteilen. Zum einen aus dem eigentlichen Programm, der `Filter()` Funktion, welche die Programmlogik bereitstellt. Zum anderen aus dem `DOMDriver()`, welcher eine Interaktion mit dem Document Object Model (DOM), also den Elementen einer Webseite, möglich macht. Die Dritte Komponente ist die direkt vom Framework bereitgestellte `run()` Methode. Der unterstehende

[6] Weitere npm-Pakete können unter `https://www.npmjs.com/` gefunden werden.

Programmausschnitt zeigt die *index.js*-Datei, also den Einstiegspunkt der Cycle.js Anwendung. Der komplette Quelltext für diesen und für die folgenden Programmausschnitte befinden sich im Anhang.

```
1 const drivers = {
2   DOM: makeDOMDriver('#app'),
3   HTTP: makeHTTPDriver()
4 }
5 run(Filter, drivers)
```

Listing 11: Ausschnitt aus den Einstiegspunkt der Cycle.js Applikation

4.2 Datenfluss

Die `Filter()`-Funktion nimmt einen Stream als Parameter entgegen, welcher Sources genannt wird. Nach dem Ausführen der Funktion wird auch ein Stream zurückgeliefert. Dieser Stream wird als Sinks bezeichnet. Die Rückgabewerte (Sinks) der `Filter()` Funktion sind nach André Staltz Anweisungen an einen Treiber (Driver) um Seiteneffekte zu erzeugen und die Eingaben (Sinks) sind lesbare Seiteneffekte[Sta16a, 1]. Unter Seiteneffekten lassen sich alle Vorgänge zusammenfassen, die eine Interaktion mit der Außenwelt erfordern. Das kann etwa die Manipulation einer Benutzeroberfläche sein oder das Speichern oder lesen von Daten auf einer Datenbank. Immer wenn in einem Cycle.js Programm Seiteneffekte erzeugt werden sollen, nutzt man dafür Treiber (mehr zu Treibern in Abschnitt 5).

In einer Cycle.js Anwendung werden Daten immer nur mithilfe von Streams übertragen. Da diese Streams immer nur in eine Richtung übertragen werden, spricht man von einem unidirektionalen Datenfluss.

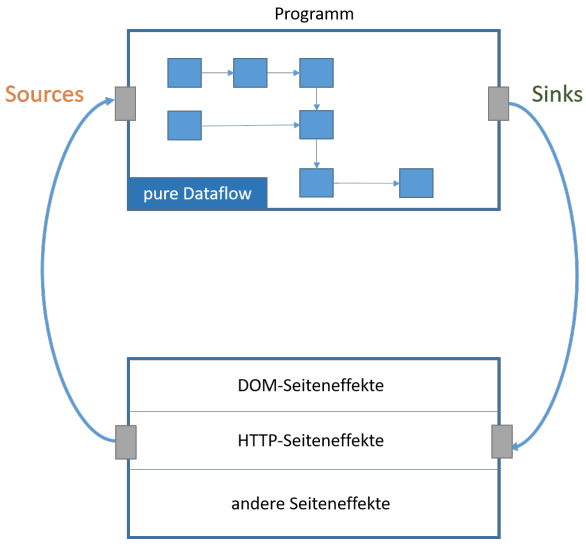

Abbildung 3: Datenfluss einer Cycle.js App nach [Sta16b]

4.3 Cycle.js Funktionen

Im untenstehenden Codebeispiel kann über das Eingabefeld ein beliebiger Text eingeben werden. Durch die Eingabe des Nutzers wird ein DOM-Event ausgelöst, welches dazu führt, dass der `Filter()` Funktion ein Stream von Events übergeben wird. Tätigt der Nutzer aufeinanderfolgend mehrere Eingaben, so entstehen mehrere Events. Diese werden nun nacheinander auf den Stream gelegt und der `Filter()` Methode übergeben.

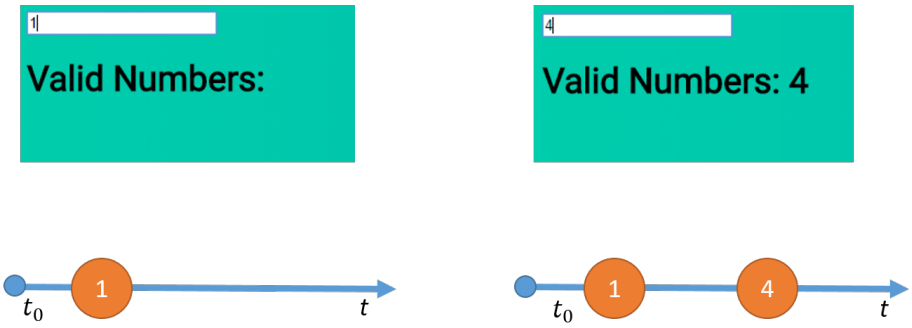

Abbildung 4: Zusammenhang Eingabe und Streams

Nun wird die DOM-Source des Eingabefelds mithilfe von `sources.DOM.select('.field')` adressiert und mithilfe `.events('input')` alle Input-Events, die davon ausgehen, ausgewählt. Daraufhin wird der Wert des Eingabefelds verarbeitet und mithilfe der `.filter()` Funktion, alle Elemente die keine Zahlen, die durch 2 teilbar sind herausgefiltert. Ohne einer Initialisierung des Streams würde der Kreislauf nicht starten. Mit der Funktion `.startWith()` wird beim Start des Streams, ein Event mit einem festgelegten Wert erzeugt. Anschließend wird der virtuelle DOM aufgebaut und die vom Nutzer erzeugten Seiteneffekte zurückgegeben.

```
1  function Filter (sources) {
2    const dom$ = sources.DOM.select('.field').events('input')
3        .map(ev => ev.target.value)
4        .startWith('')
5        .filter(input => !isNaN(input) && input % 2 == 0)
6        .map(name =>
7            div([
8              input('.field', {attrs: {type: 'text'}}),
9              h1('Valid Numbers:' + name),
10           ]))
11
12   const sinks = {
13     DOM: dom$
14   }
15   return sinks
16 }
```

Listing 12: Filter Funktion

Zusammen mit den definierten Treibern, wird die `.filter()` Funktion der `run()`-Funktion, welche den Kern einer Cycle.js Anwendung darstellt, übergeben (Mehr dazu im nächsten Abschnitt). Die `run()` Funktion sorgt dafür, dass die zyklische Abhängigkeiten zwischen den Sinks und den Sources aufgelöst werden und die Seiteneffekte ausgeführt werden. Es besteht also ein stetiger Zyklus zwischen den Ein -und Ausgängen der Cycle.js Funktion, welcher von der `run()`-Funktion aufrechterhalten wird.

```
1 run(Filter, drivers)
```

Listing 13: run()-Funktion

Im Folgenden wird der Datenfluss zwischen den Sinks und den Sources betrachtet. Je nachdem welche Events mit welchen Daten auf dem Stream vorhanden sind, wird ein andere Output-Stream erzeugt. Das liegt daran das in diesem Fall die Filteroperation die Menge der erlaubten Werte einschränkt. Das hat zur Folge das bei einer ”2” auf dem Stream der DOM anders gerendert wird als bei einer ”3”.

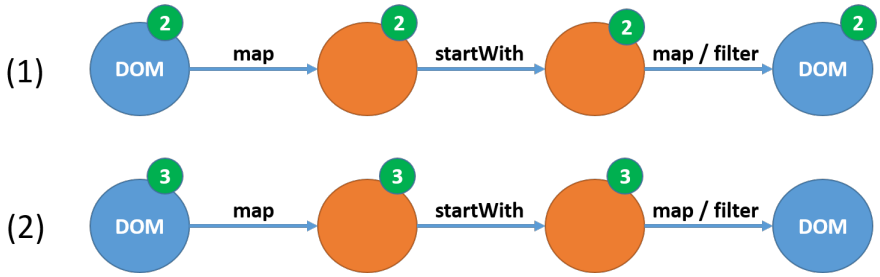

Abbildung 5: innerer Datenfluss der Filter Funktion

4.4 Die Cycle.js `run()` Funktion

Wie in Abbildung 3 auf Seite 11 beschrieben, besteht zwischen den Ein -und Ausgaben einer Cycle.js Funktion eine zyklische Abhängigkeit.

$$sources = drivers(sinks)$$

$$sinks = main(sources)$$

Diese Ausrücke lassen sich aber nicht einfach in JavaScript-Code abbilden, da es nicht erlaubt ist auf eine Ressource zuzugreifen die noch nicht deklariert wurde. Die Hauptaufgabe der `run()` Funktion ist es, diese zyklische Abhängigkeit zwischen den Sinks und den Sources aufzulösen. Anschließend wird beleuchtet, wie diese Auflösung der Abhängigkeiten technisch umgesetzt ist. Dabei werden die einzelnen Bestandteile der `run()` Funktion vereinfacht Schritt für Schritt erklärt.

Zu Beginn wird ein temporäres Observable `sinkProxies` erzeugt, um den Output-Stream des Programms darin abzuspeichern.

```
1 var sinkProxies = makeSinkProxies(drivers, streamAdapter)
```

Dieses zuvor erstellte Observable wird nun zusammen mit den Treibern der `callDrivers()` Funktion übergeben. Hier werden die Treiber zusammen mit den Daten aus den `sinkProxies` ausgeführt. Anschließend wird der von den Treibern erzeugte Sources-Stream zurückgegeben.

```
1 var sources = callDrivers(drivers, sinkProxies)
```

Nun wird die eigentliche Programmfunktion aufgerufen, welche den echten Sink-Steam zurückliefert.

```
1 var sinks = Filter(sources)
```

In dieser Funktion werden die `sinks` und die `sinkProxies` miteinander verbunden um den Zyklus zwischen den Sources des Programms, also den Streams welche vom Treiber zum Programm übertragen werden und den Sinks, also den Streams die vom Programm zu den Treibern übertragen werden, herzustellen. In dieser Funktion wird der Sinks-Stream auch von einem Observer abonniert. Dies ist notwendig, da Observables Abonnenten benötigen damit sie funktionieren. Würde man den Stream in diesem Schritt nicht abonnieren, so würde es zu einem Stillstand kommen und der Datenfluss wäre unterbrochen.

```
1 var replication = replicateMany(sinks, sinkProxies, streamAdapter)
```

Damit wurde der Kreislauf aufrechterhalten, welcher notwendig für die Reaktivität des Programms ist.

5 Treiber

Treiber oder *Driver* sind Funktionen, welche einen Input-Stream verarbeiteten und daraufhin Seiteneffekte erzeugen. Erzeugen diese eine Ausgabe, so wird ein Output-Stream von dieser Funktion zurückgegeben[Sta16e, 1]. Wie im vorherigen Abschnitt beschrieben, sind Seiteneffekte Vorgänge, welche mit der Außenwelt interagieren. Diese Interaktionen sollten in Treibern behandelt werden. Doch warum nutzt man dieses Konzept und behandelt die Seiteneffekte nicht direkt im eigentlichen Programm?

Trennt man die Programmlogik von Vorgängen mit der Außenwelt, stellt man sicher, dass die

Abläufe in der Programmfunktion pure sind. So kann man sich immer sicher sein, dass das Programm bei gleicher Eingabe der Programmablauf immer vorhersagbar ist und die erzeugten Rückgaben immer die gleichen sind. Das bringt vor allem Vorteile für die Testbarkeit und Wartbarkeit eines Programmes mit sich.

Neben den offiziellen Cycle.js Treibern für die DOM-Manipulation und dem HTTP-Treiber gibt es noch eine Reihe von weiteren inoffiziellen Treibern[7]. Aber natürlich können auch eigene Treiber programmiert werden um Side-Effekte von der Logik zu trennen.

5.1 Interaktion mit dem DOM

In fast jeder Anwendung sollen dem Benutzer Informationen auf dem Bildschirm bereitgestellt werden oder Informationen, die von der Anwendung benötigt werden, abgefragt werden. Um Benutzeroberflächen erstellen zu können muss ein Treiber installiert werden, der eine Manipulation des DOM ermöglicht. Cycle.js stellt dafür den *@cycle/dom* Treiber zur Verfügung. Der *@cycle/dom* Treiber baut auf der Virtual-DOM Bibliothek *snabbdom* auf. Es ist aber auch möglich den virtuellen DOM mit zum Beispiel *JSX*-Syntax[8] zu erstellen. Für die Beispielprogramme in dieser Seminararbeit wird die erste Variante verwendet.

Der DOM-Treiber wird mithilfe der `makeDOMDriver()` Funktion erzeugt. Als Parameter wird mithilfe eines CSS-Selektors ein Element innerhalb der *index.html* angegeben, in welches die Applikation hinein gerendert werden soll. Dieser hat zum einen die Aufgabe, den mithilfe der *snabbdom* Bibliothek erstellten virtuellen DOM auf den DOM zu schreiben und zum anderen die vom Anwender auf dem DOM erzeugten Events zu lesen.

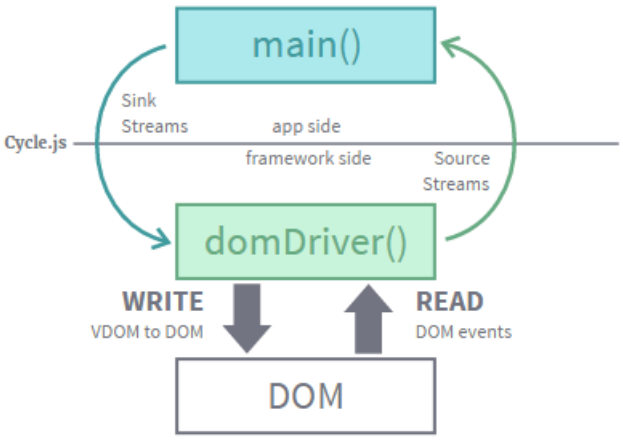

Abbildung 6: Einordung des DOM-Treibers innerhalb der Anwendung [Sta16e]

[7] Weitere Treiber sind unter `https://github.com/cyclejs-community/awesome-cyclejs#drivers` verfügbar.
[8] Dokumentation der *snabbdom*-Bibliothek: `https://www.npmjs.com/package/snabbdom-jsx`

5.1.1 virtueller DOM und nativer DOM

Bevor der Syntax vorgestellt wird, mit dem Objekte auf dem virtuellen DOM erstellt werden können, wird zunächst auf die Unterschiede zwischen dem normalen und dem virtuellen DOM eingegangen.

Laut dem W3C Standardisierungsgremium gibt das *Document Object Model* (DOM) eine logische Struktur für HTML-Dokumente vor. Dieses Dokument wird dann mithilfe von DOM-Knoten repräsentiert, welche durchsucht und modifiziert werden können[JR16]. Bei heutigen Webanwendungen mit dynamischen Benutzeroberflächen wird oftmals eine sehr große Anzahl von Elementen benötigt, welche ständig verändert werden sollen. Diese hohe Anzahl an Knoten ständig zu modifizieren, ist sehr rechenaufwendig und benötigt eine lange Berechnungszeit. Zudem muss der Entwickler selbst die Änderungen im DOM auslösen, was sehr komplex werden kann.

Um die Performance des DOM-Rendering zu verbessern, kann eine Virtual-DOM-Bibliothek, wie zum Beispiel *snabbdom*, verwendet werden. Der virtuelle DOM ist im Grunde eine Abstraktion des nativen HTML-DOM. In der *snabbdom* Bibliothek werden dafür zwei DOM-Bäume verwendet. Der erste repräsentiert, wie der DOM aktuell aufgebaut ist und der andere wie er aussehen soll. Daraufhin werden die Unterschiede beider Bäume herausgefunden und der native DOM wird verändert [Vep16]. Dabei wird nur der Knoten im DOM verändert, bei dem ein Unterschied festgestellt wurde. Dieser Änderungsvorgang wird von *snabbdom* übernommen und der Entwickler der Anwendung muss sich darum nicht kümmern. Da immer nur partielle Änderungen am DOM vorgenommen werden, ist dieses Verfahren sehr performant.

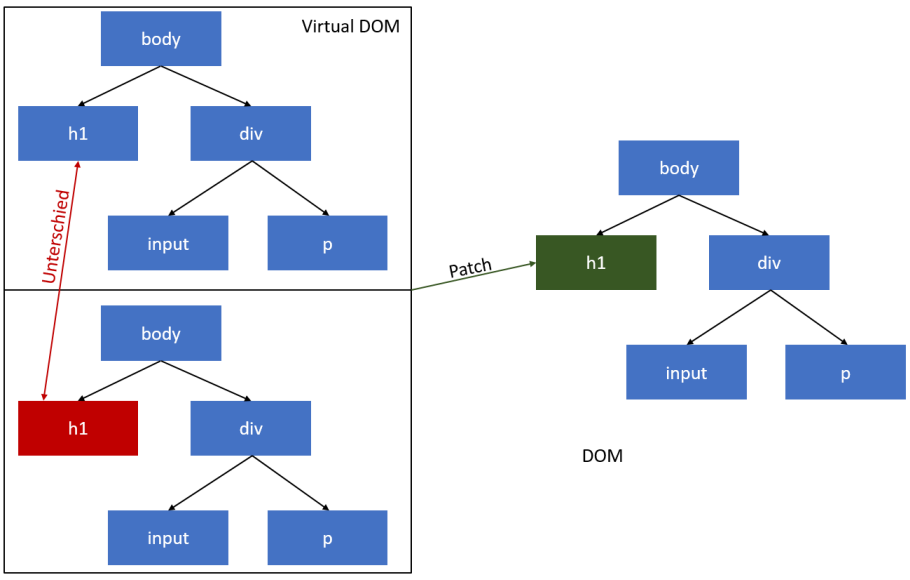

Abbildung 7: Funktionsweise des virtuellen DOM

5.1.2 Erstellen von Elementen mit dem virtuellen DOM

Um einen Knoten im virtuellen DOM zu erstellen, werden sogenannte *Hyperscript* Funktionen verwendet. Diese Objekte werden auch VNodes genannt. Für jeden HTML-Tag gibt es eine eigene Funktion, welche analog zum Namen des Tags benannt ist. Zum Beispiel ist die passende Funktion um eine Überschrift erstellen zu können `h1()` bzw. `h2()`. Im Unterstehenden Beispiel wird ein rot gefülltes `div` mit der Klasse `classname` erstellt. Im letzten Parameter kann ein Array von Kindknoten übergeben werden. In diesem Fall sind keine Kindknoten eingefügt worden.

```
1 div('.classname', {style: {color: 'red'}}, [])
```

Listing 14: Erstellen von VNodes

Sollen noch weitere Kindknoten eingefügt werden, so kann man das obenstehende Beispiel erweitern. In diesem Fall wird innerhalb des `div()` eine Überschrift eingefügt.

```
1 div('.classname', {style: {color: 'red'}}, [
2   h1('I am inside the red div')
3 ])
```

Listing 15: Erstellen von VNodes mit Kindknoten

Um einen Dialog zwischen dem Programm und dem Benutzer herzustellen werden oft Eingaben vom Benutzer benötigt. In diesem Fall kann zum Beispiel ein Eingabefeld erstellt werden. Macht der Anwender eine Eingabe, so wird ein Event auf dem Output Stream emittiert. Im nachfolgenden Code Beispiel wird das obige Beispiel um ein Eingabefeld erweitert.

```
1 div('.classname', {style: {color: 'red'}}, [
2   h1('I am inside the red div'),
3   input('.input', {attrs: {type: 'text'}}),
4 ])
```

Listing 16: Erstellen von VNodes mit Kindknoten

Die durch die Eingaben des Nutzers wird eine DOM-Source erzeugt, welche dann über `DOMSource.select(selector)` adressiert werden kann. Anschließend kann ein bestimmter Event Typ gewählt werden und Eingaben vom Nutzer ausgewertet werden.

5.1.3 Design der HTML-Elemente

Reaktivität und eine durchdachte Kommunikation mit dem Benutzer sind heutzutage sehr wichtige Elemente einer guten Benutzeroberfläche. Aber auch das Design einer Anwendung trägt zu einem großen Teil zum Nutzererlebnis bei. In diesem Abschnitt wird vorgestellt, wie man CSS mithilfe des DOM-Treibers definiert.

Wie im vorherigen Abschnitt vorgestellt, werden HTML-Elemente in sogenannten *Hyperscript* Funktionen definiert. Will man nun das Aussehen dieser Elemente verändern, so kann man die gewünschten Eigenschaften in einem JavaScript Objekt zusammenfassen. Anschließend werden diese Attribute dann der Funktion als Parameter übergeben. Im Gegensatz zu normalem CSS können so Eigenschaften dynamisch je nach Programmzustand gesetzt oder verändert werden. Da in JavaScript-Objekten keine Bindestriche für Eigenschaften zugelassen sind, müssen die CSS-Eigenschaften in die CamelCase-Notation umgeschrieben werden. Bis auf diese Einschränkung können alle CSS-Eigenschaften wie gewohnt verwendet werden. Im untenstehenden Beispiel wird ein Eingabefeld gestylt.

```
1 const inputStyle = {
2     width: "400px",
3     height: "40px",
4     margin:"20px",
5     font: "22px Roboto, Arial",
6     borderRadius: "5px",
7     border: "none"
8 }
```

Listing 17: CSS als JavaScript Objekt

Dieses Objekt kann nun als Parameter in der Hyperscript Funktion übergeben werden. Die CSS-Eigenschaften werden dann später Inline im HTML eingebunden.

```
1     input('.searchInput', {attrs: {type: 'text'}, style: inputStyle})
```

Listing 18: Hyperscript Eingabefeld

5.2 HTTP

Neben Ein -und Ausgaben auf dem Bildschirm, werden in den meisten Webapplikation Daten von einer Datenquelle benötigt oder sie sollen in einem Zielsystem verarbeitet werden. Da eine HTTP-Anfrage eine Kommunikation mit der Außenwelt darstellt, wird auch hierfür ein Treiber verwendet. Cycle.js bietet dafür eine HTTP-Treiber Bibliothek an. Im Folgenden wird die Vorgehensweise vorgestellt, wie HTTP-Anfragen in einer Cycle.js Anwendung realisiert werden können.

5.2.1 Erstellen von HTTP-Anfragen

Im nachfolgenden Beispiel wird durch das Betätigen eines Buttons auf der Benutzeroberfläche ein Event auf dem Source-Stream erzeugt. Bei jedem Auftreten eines solchen Events wird eine GET-Anfrage an einen Server gesendet und anschließend die Antwort verarbeitet.

Zuerst muss die Anfrage an den Server modelliert werden. Die verschiedenen Informationen werden in einem Anfrage-Objekt zusammengefasst. Für jede Anfrage an einen Server muss die URL der Ressource definiert werden. Dazu wird die Eigenschaft `url` verwendet. In der Eigenschaft `method` kann definiert werden welcher Typ die Anfrage darstellt, also ob es zum Beispiel eine GET, POST oder PUT Anfrage ist. Zudem kann in dem Feld `category` ein Schlüssel festgelegt werden, mit dem die Antwort in dem Source-Stream der Anwendung adressiert werden kann. Will man zum Beispiel Daten mithilfe POST an einen Server senden, so müssen noch weitere Eigenschaften definiert werden. So können mit dem Attribut `query` die Nutzdaten (Payload) der Anfrage festgelegt werden. Neben diesen existiert noch eine Reihe weiterer Eigenschaften[9], die analog zu den zuvor vorgestellten Eigenschaften funktionieren.

```
1 const request$ = sources.DOM
2         .select('.infm')
3         .events('click')
4         .map(()=> {
5         url: `https://www.iwi.hs-karlsruhe.de/Intranetaccess/REST/
              newsbulletinboard/INFB`,
6         method:'GET',
7         category: 'blackboard'
8     })
```

Listing 19: HTTP-Anfrage an einen Server

Hat man die Anfrage modelliert, so kann man die Antwort aus dem Source-Stream der Programmfunktion auswählen. Diese wird analog zu Events auf dem DOM umgesetzt: `sources.http.select(category)`. Anschließend kann man mithilfe der `map()` Funktion auf

[9] Vollständige API des HTTP-Treibers: `https://github.com/cyclejs/cyclejs/tree/master/http#api`

die Antwort des Servers zugreifen und sie im weiteren Programm verwenden, zum Beispiel um dem Benutzer Teile der Antwort darzustellen.

```
1     const response$ = sources.HTTP
2         .select('blackboard')
3         .flatten()
4         .map(res => res.body)
```

Listing 20: Verarbeiten der Antwort vom Server

Damit der erzeugte Stream nun auch mit den festgelegten Treibern verbunden wird, muss man ihn der Cycle.js `run()` Funktion übergeben.

```
1 run({DOM: vdom$, HTTP: request$}, drivers)
```

Listing 21: Die Treiber als Parameter von `run()`

6 Modularisierung

In den bisherigen Beispielen wurde immer das ganze Programm in einer Funktion geschrieben, welche die komplette Logik beinhaltet. Entwickelt man aber eine größere Anwendung, so würde diese Funktion weiter anwachsen und es wird immer schwieriger den Überblick zu behalten und das Programm zu warten. Im Folgenden wird vorgestellt wie man Anwendungen mithilfe des Model-View-Intent Pattern (MVI) strukturiert und ein Programm modularisieren kann, indem man ein Programm aus mehreren unabhängigen Cycle.js Funktionen zusammensetzten kann.

6.1 Model-View-Intent Pattern

Um die Zuständigkeiten eines Programms besser zu verteilen und damit die Wartbarkeit verbessern zu können, wird das Model-View-Intent Pattern verwendet. Dieses Entwurfsmuster teilt eine Cycle.js Funktion in drei Teile ein: Model, View und Intent [Sta16f]. Jeder dieser drei Teile

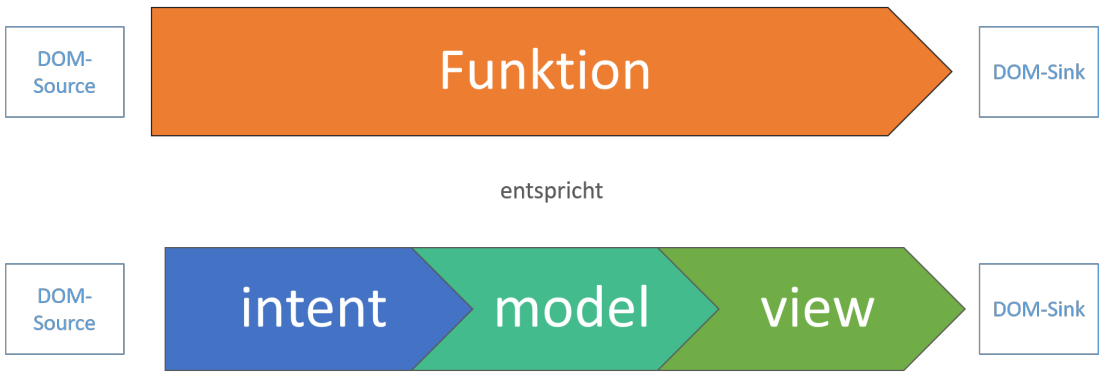

Abbildung 8: Model-View-Intent Pattern

hat einen anderen Input und Output. Betrachtet man diese drei Teile im Zusammenspiel, so stellt man fest, dass von außen betrachtet auch ein Source-Stream entgegen genommen wird und ein Sink-Stream zurückgegeben wird. Der Datenfluss wird also durch die Anwendung dieses Pattern nicht verändert.

```
1 sinks = main(sources)
```

Die Anwendung wurde hier mithilfe des MVI-Patterns strukturiert, ist aber funktional betrachtet gleich wie die Version ohne Strukturierung.

```
1 sinks = view(model(intent(sources)))
```

Model, View und Intent haben jeweils einen eigenen Aufgabenbereich und unterscheiden sich in ihrem Input und Output. Die folgende Tabelle stellt die Unterschiede dar:

	Verwendung	**Input**	**Output**
Model	Verwaltet den Zustand des Programms	Action Streams	State Stream
View	Stellt den Zustand des Programms graphisch dar	State Stream	Stream aus virtuellen DOM-Elementen ≙ DOM-Treiber Sink
Intent	Verarbeitet Events auf dem DOM ≙ Benutzeraktionen	DOM-Source Stream	Action Streams

Wie das Model-View-Intent Pattern praktisch angewandt wird, kann in Anhang A gesehen werden. Dort wurde das oben beschriebene `filter()`-Beispiel mithilfe dieses Entwurfsmusters umgeschrieben. Das Model-View-Intent Pattern ist an das Model-View-Control (MVC) Pattern angelehnt. Da aber der proaktive Controller (C) im MVC-Pattern für einen reaktiven Datenfluss nicht optimal ist, benötigt man einen reaktiven Controller um die Input-Events verwalten zu können. Ersetzt wurde der Controller durch den Intent Teil. Wie schon im Abschnitt 5 beschrieben, sind alle Teile unseres Programms pure. Diese Vorgabe darf auch das MVI-Pattern nicht verletzen. Das hat zur Folge, dass Seiteneffekte nicht in einem der drei Teile erzeugt werden dürfen. Die Treiber müssen also separat betrachtet werden.

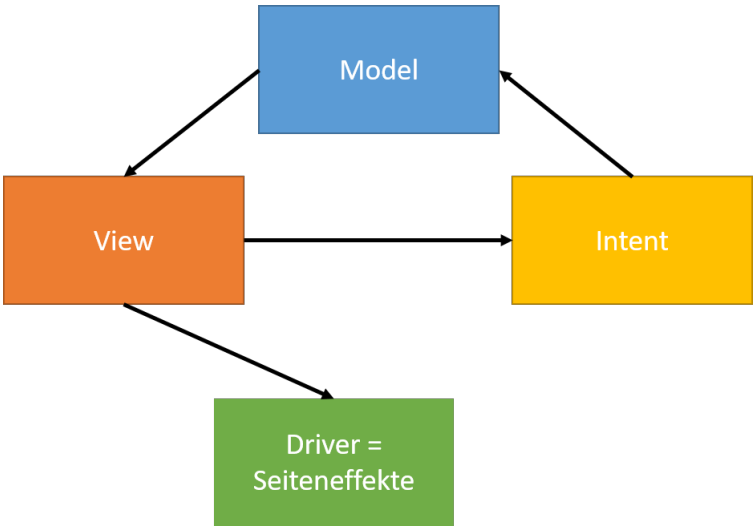

Abbildung 9: Struktur Model-View-Intent Pattern

6.2 Komponenten

Im Normalfall besteht eine Anwendung aus vielen kleinen Teilen, die man atomar betrachten kann. So gibt es zum Beispiel eine Gruppe von Eingabefelder, die zu einem Formular zusammengefasst werden, Buttons oder Komponenten die Informationen darstellen. Oft sollen diese Teile auch in einem Programm wiederverwendet werden. Deswegen ist es oft notwendig das Programm, welches bisher nur aus einer Funktion besteht, in viele kleine Funktionen aufzuteilen.

6.2.1 Aufbau einer Komponente

Jede Komponente hat Ein -und Ausgänge. Als Eingang bekommt eine Komponente von der darüberliegenden Komponente Events aus dem DOM, sowie Vorgaben über das Aussehen oder des Verhaltens, welche auch *props* genannt werden. Zurückgegeben wird von dieser Komponente ein Virtual-DOM-Stream sowie Werte, die für die darüberliegende Komponenten relevant ist.

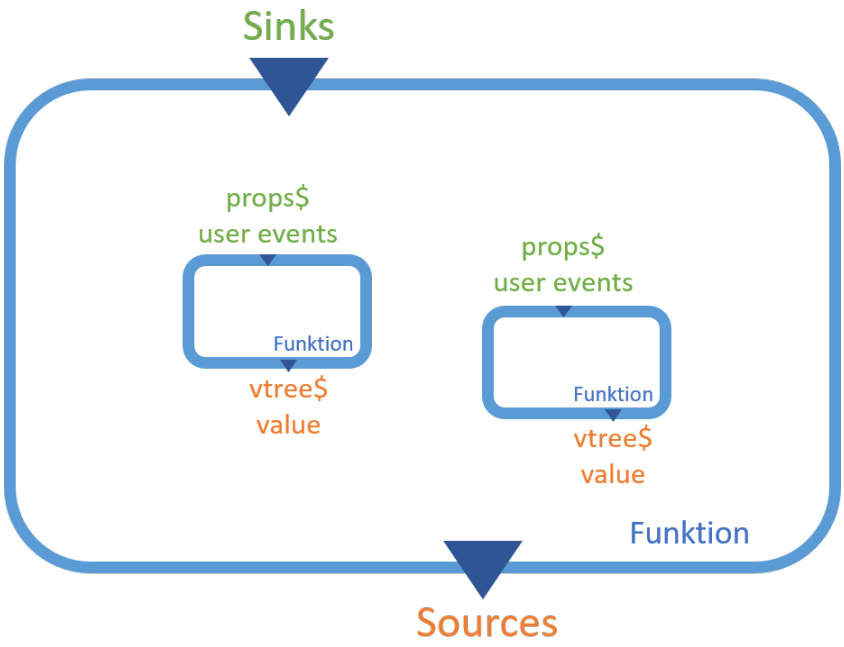

Abbildung 10: Komponentenbasierte Strukturierung

Bisher haben alle Komponenten den gleichen Sichtbarkeitsbereich (Scope). Um Komponenten voneinander zu isolieren, kann man die *Isolate*-Bibliothek[10] verwenden, welche die Methode `isolate(dataflowComponent)` zur Verfügung stellt. Diese nimmt eine Komponente entgegen und gibt eine isolierte Version zurück.

[10]Die Dokumentation der *Isolate*-Bibliothek kann unter `https://github.com/cyclejs/cyclejs/tree/master/isolate` eingesehen werden.

7 Vergleich mit anderen Frameworks

Im folgenden wird Cycle.js mit Elm[11] und Angular2[12] verglichen. Es werden die Gemeinsamkeiten und Unterschiede, gerade in Bezug auf die Umsetzung des Datenflusses und der Reaktivität, herausgearbeitet.

7.1 Elm

Elm wurde 2012 von Evan Czaplicki im Rahmen seiner Masterarbeit vorgestellt[Cza12]. Elm ist eine funktionale Programmiersprache, welche zu JavaScript kompiliert wird[Cza16] und genauso wie Cycle.js dazu verwendet wird, um reaktive Webanwendungen für den Client zu erstellen. Elm ist Cycle.js an vielen Stellen sehr ähnlich. Das Datenflussmodell von Elm, das Model-View-Update Pattern, ist dem Model-View-Intent Pattern sehr ähnlich. Beide Frameworks setzen einen unidirektionalen Datenfluss um. Die Unterschiede finden sind in der Zuständigkeit der View Komponente. Elm verarbeitet dort auch Nutzereingaben, während die View bei Cycle.js nur die Präsentation der Daten zuständig ist.

Beide Frameworks verfolgen den Ansatz der Funktionalen Reaktiven Programmierung. Da JavaScript keine rein funktionale Programmiersprache ist, liegt es bei der Entwicklung von Anwendungen mithilfe von Cycle.js, in der Hand des Entwicklers, die Konzepte der funktionalen Programmierung diszipliniert umzusetzen. Elm dagegen ist eine rein Funktionale Programmiersprache, es ist also nicht möglich außerhalb dieses Paradigmas zu entwickeln. Elm grenzt sich in einigen Punkten gegenüber klassischem JavaScript ab. So bietet Elm zum Beispiel eine strenge statische Typisierung, was zur Folge hat, dass keine Fehler zur Laufzeit auftreten. Der streng funktionale Ansatz und die statische Typisierung, macht Elm zu einer sehr interessanten Alternative um reaktive Webanwendungen zu entwickeln.

7.2 Angular2

Angular2 ist ein komponentenbasiertes Framework zum Erstellen von interaktiven Benutzeroberflächen und verwendet wie Cycle.js zu Beginn seiner Entwicklung, das reaktive Framework RxJs. In einer Angular2 Anwendung werden die Daten auch über asynchrone reaktive Streams übermittelt. Es werden aber keine spezifischen Vorgaben gemacht, in welcher Art diese eingesetzt werden. So kann zum Beispiel das Flux-Pattern verwendet werden[13]. Angular2 verwendet zwar das RxJs-Framework, welches einen funktionale Ansatz verfolgt. Es ist aber nicht vorgesehen mit Angular2 rein funktionale Anwendungen programmieren zu können. Angular2 ist sehr stark in der Industrie verbreitet und es gibt eine große Anzahl an Entwicklern, die mit diesem

[11] Webseite von Elm: `http://elm-lang.org/`
[12] Webseite von Angular2: `https://angular.io/docs/ts/latest/quickstart.html`
[13] Github-Repository der Flux-Implementierung *ng2-redux*: `https://github.com/angular-redux/ng2-redux`

Framework arbeiten. Das hat zu Folge, dass es sehr viele Bibliotheken gibt, mit denen sich eine Angular2-Anwendung an fast jede Anforderung anpassen lässt.

8 Fazit

In dieser Arbeit wurde eine Einführung in die funktionale und reaktive Programmierung mit JavaScript gegeben. Es wurde beschrieben, wie man mit dem leichtgewichtigen Framework Cycle.js clientseitige Anwendungen, die einen übersichtlichen Datenfluss aufweisen, entwickeln kann. Im Abschnitt 6 wurde beschrieben, wie man eine Anwendung mithilfe des Model-View-Intent Pattern, auch wachsende Anwendungen strukturieren kann. Das Cycle.js Framework hebt sich an vielen Stellen von anderen Frameworks wie React, Angular oder Angular2 ab. All diese Frameworks setzten eher auf den imperativen oder objektorientierten Ansatz, während Cycle.js bei der rein funktionalen Programmierung unterstützt. Aber es liegt in der Verantwortung des Entwicklers, die Regeln der funktionalen Programmierung umzusetzen. Cycle.js gibt im Grunde nur ein Grundgerüst vor, um reaktive Anwendungen auf Grundlage von asynchronen Streams zu entwickeln. Es werden wenig Vorgaben bei der Auswahl der Stream-Bibliothek gemacht. Das bringt zum einen Flexibilität mit sich, aber auf der anderen Seite muss sich der Entwickler vorab entscheiden, welche Bibliothek er verwenden möchte. Trotz des etwas schwierigen Einstiegs, ist das Konzept des zyklischen Datenflusses sehr attraktiv, da es sehr nah an einer echten Kommunikation angelehnt ist. Zudem ist Cycle.js ein sehr leichtgewichtiges Framework: die Core-Bibliothek hat gerade einmal eine Größe von 4kb und auch zusammen mit der xtream und der DOM-Bibliothek, bleibt man unter 100kb Gesamtgröße[Sta16c]. Angular2 hat zusammen mit der notwendigen Stream-Bibliothek etwa 800kb[14]. Das bringt Vorteile bei der initialen Ladezeit der Webseite und ist gerade bei mobilen Geräten ein wichtiger Faktor.

Das Framework hat im letzten Jahr stark an Bekanntheit gewonnen, steht aber im Schatten etablierter Frontend-Frameworks. Das hat zur Folge, dass das Ökosystem um Cycle.js noch klein ist, was den Einsatz in einer produktiven Umgebung noch etwas riskant macht. Es lohnt sich bei der Auswahl eines geeigneten Frameworks, einen Blick auf die Alternativen wie Elm oder Angular2 zu werfen. Aber die klare Sichtweise auf die Mensch-Computer-Interaktion und die konsequente technische Umsetzung dieses Konzeptes ist sehr beeindruckend. Hat man sich die benötigten Grundlagen angeeignet, so kann man gut testbare, übersichtliche und schnelle Anwendungen erstellen.

[14] Größenvergleich verschiedener Frameworks: `https://gist.github.com/Restuta/cda69e50a853aa64912d`

9 Quellenverzeichnis

[BF16] Beate Freesl, Wolfgang K.: Dynamische Entwicklung bei mobiler Internetnutzung sowie Audios und Videos. In: *Media Perspektiven* (2016), September, 418-437. http://www.ard-zdf-onlinestudie.de/fileadmin/Onlinestudie_2016/0916_Koch_Frees.pdf. – ISSN 0170–1754. – Veröffentlicht von: Intendant des Hessischen Rundfunks in Zusammenarbeit mit der ARD-Werbung

[Con16] Contributors, Mozilla: *Array.prototype.map() Dokumentation* . https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array/map, 2016. – Veröffentlicht von: Mozilla Foundation Online, Zugriff am 23-November-2016

[Cza12] Czaplicki, Evan: *Elm: Concurrent FRP for Functional GUIs*. https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf, März 2012. – Veröffentlicht von: Harvard University, Online; Zugriff am 10-Dezember-2016

[Cza16] Czaplicki, Evan: *An Introduction to Elm*. https://guide.elm-lang.org/, 2016. – Veröffentlicht von: Evan Czaplicki, Online; Zugriff am 01-Dezember-2016

[Hud89] Hudak, Paul: Conception, Evolution, and Application of Functional Programming Languages. In: *ACM Comput. Surv.* 21 (1989), September, Nr. 3, 359–411. http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf. – ISSN 0360–0300. – Veröffentlicht von: ACM

[JB14] Jonas Bonér, Roland Kuhn und Martin T. Dave Farley F. Dave Farley: *Das Reaktive Manifest*. https://github.com/reactivemanifesto/reactivemanifesto, 2014. – Veröffentlicht von: GitHub, Online; Zugriff am 06-Dezember-2016

[JR16] Jonathan Robie, Texcel R.: *What is the Document Object Model?* https://www.w3.org/TR/WD-DOM/introduction.html, 2016. – Veröffentlicht von: World Wide Web Consortium (W3C), Online; Zugriff am 13-November-2016

[Kop16] Koppers, Tobias: *Webpack*. https://github.com/webpack/webpack, 2016. – Veröffentlicht von: GitHub, Online; Zugriff am 29-November-2016

[Sta16a] Staltz, André: *Basic Examples*. https://cycle.js.org/basic-examples.html, 2016. – Veröffentlicht von: André Staltz, Online; Zugriff am 12-November-2016

[Sta16b] Staltz, André: *Cycle.js, A functional and reactive JavaScript framework for cleaner code*. https://cycle.js.org, 2016. – Veröffentlicht von: André Staltz, Online; Zugriff am 18-Oktober-2008

[Sta16c] Staltz, André: *Cycle.js Streams*. https://cycle.js.org/streams.html, 2016. – Veröffentlicht von: André Staltz, Online; Zugriff am 27-Oktober-2016

[Sta16d] Staltz, André: *DIALOGUE ABSTRACTION*. `https://cycle.js.org/dialogue.html`, 2016. – Veröffentlicht von: André Staltz, Online; Zugriff am 18-Oktober-2016

[Sta16e] Staltz, André: *DRIVERS*. `https://cycle.js.org/drivers.html`, 2016. – Veröffentlicht von: André Staltz, Online; Zugriff am 26-Oktober-2016

[Sta16f] Staltz, André: *MODEL-VIEW-INTENT*. `https://cycle.js.org/model-view-intent.html`, 2016. – Veröffentlicht von: André Staltz, Online; Zugriff am 01-November-2016

[Tor96] Torgersson, Olof: *A note on declarative programming paradigms and the future of definitional programming*. `http://www.cse.chalmers.se/~oloft/Papers/wm96.pdf`, 1996. – Veröffentlicht von: Chalmers University of Technology and Goeteborg University, Online; Zugriff am 05-Dezember-2016

[Vep16] Vepsäläinen, Juho: *Snabbdom - a Virtual DOM Focusing on Simplicity - Interview with Simon Friis Vindum*. `http://survivejs.com/blog/snabbdom-interview/`, 2016. – Veröffentlicht von: Juho Vepsäläinen, Online; Zugriff am 18-Oktober-2008

A Quellcode der index.js

```
import {run} from '@cycle/xstream-run'
import {makeDOMDriver} from '@cycle/dom'
import {makeHTTPDriver} from '@cycle/http'

import {Filter} from './filter/filter'
import {FilterMVI} from './filter/filterMVI'

//Choose active Cycle.js Function here
const main = Filter

const drivers = {
  DOM: makeDOMDriver('#app'),
  HTTP: makeHTTPDriver()
}

run(main, drivers)
```

B Quellcode der Filter Funktion

```
import {div, input, h1} from '@cycle/dom'

export function Filter (sources) {
  const dom$ = sources.DOM.select('.input').events('input')
      .map(ev => ev.target.value)
      .startWith('')
      .filter(input => !isNaN(input) && input % 2 == 0)
      .map(name =>
          div([
            input('.input', {attrs: {type: 'text'}}),
            h1('Valid Numbers: ' + name),
          ]))

  const sinks = {
    DOM: dom$
  }
  return sinks
}
```

C Quellcode der Filter Funktion (Model-View-Intent Pattern)

```
1  import {div, input, h1} from '@cycle/dom'
2
3  function intent(domSource) {
4      return {
5          inputField$: domSource.DOM.select('.input').events('input')
6              .map(ev => ev.target.value)
7      }
8  }
9
10 function model(actions) {
11     return {
12         input: actions.inputField$.startWith('')
13             .filter(input => !isNaN(input) && input % 2 == 0)
14     }
15 }
16
17 function view(state$) {
18     return state$.input.map(name =>
19         div([
20             input('.input', {attrs: {type: 'text'}}),
21             h1('Valid Numbers: ' + name)
22         ]))
23 }
24
25 export function FilterMVI (sources) {
26     const actions = intent(sources)
27     const state$ = model(actions)
28     const vdom$ = view(state$)
29
30     const sinks = {
31         DOM: vdom$
32     }
33     return sinks
34 }
```