



FH Salzburg
MultiMediaTechnology

Deadcode Detection mittels statischer Analyse in Javascript

Bachelorarbeit 2

AutorIn: Thomas Siller
BetreuerIn: Mag Hannes Moser

Salzburg, Österreich, 26.02.2018

Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Weiter versichere ich hiermit, dass ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission weder im In- noch im Ausland vorgelegt und auch nicht veröffentlicht.

Datum

Unterschrift

Vorname *Nachname*

Kurzfassung

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean venenatis nulla vestibulum dignissim molestie. Quisque tristique tortor vitae condimentum egestas. Donec vitae odio et quam porta iaculis ut non metus. Sed fermentum mauris non viverra pretium. Nullam id facilisis purus, et aliquet sapien. Pellentesque eros ex, faucibus non finibus a, pellentesque eu nibh. Aenean odio lacus, fermentum eu leo in, dapibus varius dolor. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin sit amet ornare velit. Donec sit amet odio eu leo viverra blandit. Ut feugiat justo eget sapien porttitor, sit amet venenatis lacus auctor. Curabitur interdum ligula nec metus sollicitudin vestibulum. Fusce placerat augue eu orci maximus, id interdum tortor efficitur.

Abstract

Inhaltsverzeichnis

1	Einleitung	1
2	Static Module Bundler	1
3	Modules in JavaScript	1
3.1	CommonJS	2
3.2	Asynchronous Model Definition (AMD)	2
3.3	ECMAScript 6	3
3.3.1	Named Exports	3
3.3.2	Default Exports	4
3.3.3	Static Module Structure	4
3.3.4	Vorteile	5
3.3.5	Imports	6
3.3.6	Syntax	8
4	Tree Shaking	8
4.1	Webpack	10
4.2	RollupJS	11
4.3	Hindernisse	11
4.3.1	Richtiges Importieren und Exportieren	11
4.3.2	Exportieren	12
4.3.3	Side Effects	12
5	Linting	15
5.1	ES Lint	16
5.1.1	Core Rules	16
5.1.2	Komponenten eines Plugins	17
5.1.3	How to use	17
5.1.4	Formatters	17
5.2	Dependency Graph	17
6	Untersuchung von JavaScript Libraries	18
6.1	Auswahlkriterien	18
6.2	Implementierung von Tree Shaking	19

7	Erstellen des Plugins	19
7.1	Setup	19
7.2	Entwicklungsumgebung	20
8	Die Implementierung neuer Regeln	20
8.1	Empty Imports	21
8.2	Named Exports SideEffects	24
8.3	Entry Point SideEffects	28
	Appendices	35
A	git-Repository	35

1 Einleitung

Für Javascript Webentwickler ist das NPM Registry eines der wichtigsten Werkzeuge. Es bietet eine Plattform für Entwickler, um Sourcecode mit anderen zu teilen, bestehende Module herunterzuladen und in die eigenen Anwendung zu integrieren. Seit dem Release wurden über 600.000 verschiedene Software Pakete hinzugefügt und mit mehr als einer Milliarden Downloads pro Woche ist es zu einer zentralen Bibliothek von Frameworks, Libraries und Werkzeuge für die Javascript Entwicklung geworden. (Npmdocs)

Die Anzahl der Module und deren komplexe Abhängigkeiten zueinander führten jedoch im Bereich der Frontendentwicklung zu fundamentalen Problemen. Neben der Dependency Hell, dem wohl bekannteste dieser Probleme, ist ebenso entscheidend die an den Browser zu übertragende Datenmenge möglichst gering zu halten. Um dies zu erreichen muss der Entwickler oder die Entwicklerin unnötige Versions Duplikate und unerreichbare Zweige im Programmcode vermeiden. (Dominik Wilkowski 2018)

2 Static Module Bundler

Static Module Bundler wie Webpack und RollupJS ermittelt alle für die Anwendung benötigten Module und bündelt diese anschließend in einer oder mehrere Dateien. Dafür wird ausgehend von konfigurierten Einstiegspunkten ein Baum aus Abhängigkeiten erzeugt. Neben dem eigenen Programmcode beinhaltet dieser auch Npm Pakete und sogar statische Assets wie Bilder oder CSS Dateien. Die Abhängigkeiten werden aufgelöst um daraus wieder je nach Konfiguration eine oder mehrere Bundle Dateien zu erzeugen. Je nach Anwendungsfall kann es von Vorteil sein den Programmcode als eine größere Datei oder in mehreren Teilen an den Browser auszuliefern, um ein möglichst schnelles laden der Webseite zu gewährleisten. (RollupJS Docs 2018; Webpack Concepts 2018)

(Bundler so wie Webpack umhüllen jedes Code Module in Funktionen. Damit wird eine Browser freundliche Implementierung und das Laden der Assets je nach Bedarf gewährleistet. Wenn jedoch viele kleine Module benötigt werden führt dies zu einigen Schwierigkeiten. (Rich Harris 2018))

3 Modules in JavaScript

In traditionellen JavaScript Webprojekten werden alle Abhängigkeiten als Liste aus `<script>` tags definiert. Es ist die Aufgabe des Entwicklers oder der Entwicklerin sicherzustellen das diese in der richtigen Reihenfolge zur Verfügung gestellt werden. Je komplexer die Abhängigkeiten zwischen dem Programmcode, Frameworks und Libraries werden, desto schwieriger wird die Wartung und die richtige Platzierung neuer Einträge in der richtigen Reihenfolge. Es wurden bereits mit verschiedenen Modul Spezifikationen versucht diese Schwierigkeiten für Entwickler und Entwicklerinnen zu vereinfachen. (Sebastian Peyrott 2018)

3.1 CommonJS

Das Ziel von CommonJS ist eine Modul Spezifikation zur Erleichterung der Entwicklung von Serverseitigen JavaScript Anwendungen. NodeJS Entwickler verfolgte anfangs diesen Ansatz, entschieden sich im späteren Verlauf jedoch dagegen. Dennoch wurde die Implementierung stark von der CommonJS Spezifikation beeinflusst. (Sebastian Peyrott 2018)

```
1 // In circle.js
2 const PI = Math.PI;
3
4 exports.area = (r) => PI * r * r;
5
6 exports.circumference = (r) => 2 * PI * r;
7
8 // In some file
9 const circle = require('./circle.js');
10 console.log( `The area of a circle of radius 4 is ${circle.area(4)} `);
```

Listing 1: CommonJS Module

Sowohl bei CommonJS als auch bei der NodeJS Implementierung gibt es zwei Elemente um mit dem Modul System zu interagieren: `require` und `exports`. `require` wird benötigt um ein anderes Modul in den aktuellen Scope zu importieren. Dabei wird als Parameter die Modul-id angegeben. Diese ist entweder der Name des Moduls innerhalb des `node_modules` Ordner oder der gesamte Pfad. `exports` dient zur Definition einer Modul Schnittstelle. Jede Property des `exports` Objekts wird als öffentliches Element exportiert. Der Unterschied zwischen NodeJS und CommonJS liegt vorallem im `module.exports` Objekt. Dies erfüllt die selbe Aufgabe wie bei CommonJS das `exports` Objekt. Abschließend bleibt zu erwähnen das die Module synchron geladen werden. Sprich in dem Moment und in der Reihnfolge in der sie mit `require` angegeben wurden. (Sebastian Peyrott 2018)

3.2 Asynchronous Model Definition (AMD)

Das Entwickler Team von AMD spaltete sich während der Entwicklung von CommonJS ab. Der Haupt Unterschied zwischen diese beiden Systemen liegt in dem asynchronen laden von Ressourcen. (Sebastian Peyrott 2018)

```
1 //Calling define with a dependency array and a factory function
2 define(['dep1', 'dep2'], function (dep1, dep2) {
3
4     //Define the module value by returning a value.
5     return function () {};
6 });
7
8 // Or:
9 define(function (require) {
10     var dep1 = require('dep1'),
11         dep2 = require('dep2');
12 }
```



```
13   return function () {};  
14 });
```

Listing 2: Asynchronous Model Definition

Das asynchrone Laden wird ermöglicht durch einen Funktionsaufruf nach dem anfordern der Abhängigkeiten. Libraries die nicht voneinander abhängig sind können somit zur selben Zeit geladen werden. Dies ist besonders wichtig für die Frontend Entwicklung, da dort die Startzeit einer Anwendung essenziell für ein gutes Benutzererlebnis ist. (Sebastian Peyrott 2018)

3.3 ECMAScript 6

Ziel der ECMAScript 6 Modules war es ein Format zu kreieren, welches sowohl AMD, als auch CommonJS User zufriedenstellen. Die Spezifikation besitzt eine kompaktere Syntax als CommonJS und ähnlich zu AMD wird asynchrones Nachladen direkt unterstützt. Neben diesen Vorteilen wurde auch die Möglichkeit der statischen Code Analyse für Optimierungen geschaffen.

ES6 bietet 2 Arten von Exporte: Named Exports (mehrere pro Modul) und Default Exports (einen pro Modul). (AxelRauschmayer2014)

3.3.1 Named Exports

Mit Hilfe des Präfixes `export` ist es möglich mehrere Elemente aus einem Modul zu exportieren. Dies gilt sowohl für Variablen, als auch Funktionen und Klassen. Sie werden anhand ihres Namens unterschieden und werden deshalb auch als Named Exports bezeichnet. (AxelRauschmayer2014)

```
1  //----- lib.js -----  
2  export const sqrt = Math.sqrt;  
3  export function square(x) {  
4      return x * x;  
5  }  
6  export function diag(x, y) {  
7      return sqrt(square(x) + square(y));  
8  }  
9  
10 //----- main.js -----  
11 import { square, diag } from 'lib';  
12 console.log(square(11)); // 121  
13 console.log(diag(4, 3)); // 5
```

Listing 3: Named Exports

Neben dem Beispiel in 3, gibt es noch weitere Wege um Named Exports zu erzeugen. Diese werden im Kapitel 3.3.6 als Überblick dargestellt.

3.3.2 Default Exports

Es gibt den Fall das ein Modul auch nur ein Objekt exportiert. In der Frontendentwicklung tritt dies häufig auf wenn nur eine Klasse oder einen Konstruktor implementiert wird. Dabei wird er Default Export verwendet. Er ist wichtigste Element eines ECMAScript 6 Moduls und daher auch sehr leicht zu importieren. (AxelRauschmayer2014)

```
1 //----- myFunc.js -----
2 export default function () { ... };
3
4 //----- main1.js -----
5 import myFunc from 'myFunc';
6 myFunc();
7
8 //----- MyClass.js -----
9 export default class { ... };
10
11 //----- main2.js -----
12 import MyClass from 'MyClass';
13 let inst = new MyClass();
```

Listing 4: Default Exports

Wie 4 veranschaulicht, besitzt der Default Export keinen Namen. Beim Importieren wird daher meist der Modul Name für die Identifizierung verwendet.

In Wirklichkeit ist der Default Export auch ein Named Export mit dem speziellen Namen **default**. Somit sind die folgenden Import Deklarationen in dem Beispiel 5 gleichbedeutend. (AxelRauschmayer2014)

```
1 import { default as foo } from 'lib';
2 import foo from 'lib';
```

Listing 5: Import Deklaration

Das gleiche gilt für den Export von Elementen.

```
1 //----- module1.js -----
2 export default 123;
3
4 //----- module2.js -----
5 const D = 123;
6 export { D as default };
```

Listing 6: Export Deklaration

3.3.3 Static Module Structure

Bei anderen Module Systeme wie CommonJS muss man den Programmcode ausführen, um herauszufinden welche Importe und Exporte verwendet werden. Die Spezifikation von ES 6 zwingen den Entwickler oder die Entwicklerin zu einer statischen Modul Struktur. In diesem Kapitel wird erläutert was dies bedeutet und welche Vorteile dadurch entstehen. (AxelRauschmayer2014)

Eine statische Modul Struktur ermöglicht es durch kompilieren des Codes die darin befindlichen Importe und Exporte zu ermitteln. Es ist nicht nötig den Sourcecode auszuführen. In den folgenden Beispiel 7 wird gezeigt warum dies bei CommonJS nicht möglich ist. (AxelRauschmayer2014)

```
1 var mylib;
2 if (Math.random()) {
3   mylib = require('foo');
4 } else {
5   mylib = require('bar');
6 }
7
8 if (Math.random()) {
9   exports.baz = ...;
10 }
```

Listing 7: Flexibele Struktur bei CommonJS

Bei der CommonJS Deklaration aus Beispiel 7 wird erst bei der Ausführung des Codes entschieden welcher Import beziehungsweise Export verwendet wird. Es6 Module besitzen nicht diese Flexibilität. Sie zwingen den Entwickler oder die Entwicklerin dazu eine flache und statische Modul Strukturen zu verwenden. Neben den Verlust der Flexibilität bringt dies jedoch einige Vorteile mit sich. (AxelRauschmayer2014)

3.3.4 Vorteile

1. Schnellerer Lookup

CommonJS liefert beim Anfordern von abhängigen Modulen ein Objekt zurück. Beim Aufruf von zum Beispiel `lib.someFunc()`; muss ein langsamer Property Lookup durchgeführt werden.

Im Gegensatz dazu sind die Inhalte bei dem Import einer ES6 Moduls bekannt. Der Zugriff auf die Properties kann daher optimiert werden. (AxelRauschmayer2014)

2. Variablen Überprüfung

Dank der statischen Modul Struktur, sind die verfügbaren Variablen innerhalb eines Moduls immer bekannt. Dazu zählen:

- Globale Variablen
- Modul Importe
- Modul Variablen

Dies bietet eine große Hilfe für Entwickler und Entwicklerinnen. Durch eine statische Überprüfung mit einem Linter werden frühzeitige Tippfehler oder auch nicht verfügbare Variablen erkannt. (AxelRauschmayer2014)

3. Macro Support

(scheint nicht wirklich viel verwendet zu werden weiterer research nötig)

```

1 syntax hi = function (ctx) {
2   return #`console.log('hello, world!')`;
3 };
4 hi

```

Listing 8: Macros in Javascript

4. Type Support

(scheint nicht wirklich viel verwendet zu werden weiterer research nötig)

5. Unterstützung mehrerer Sprachen

(scheint nicht wirklich viel verwendet zu werden weiterer research nötig)

3.3.5 Imports

CommonJS sind Importe Kopien der exportierten Werte. Module Importe in ECMAScript 6 hingegen sind schreibgeschützte Views auf die exportierten Entitäten. Dies bedeutet das Variablen die innerhalb eines Modules deklariert wurden während der gesamten Programmlaufzeit bestehen bleiben. (AxelRauschmayer2018)

```

1 //----- lib.js -----
2 export let counter = 3;
3 export function incCounter() {
4   counter++;
5 }
6
7 //----- main1.js -----
8 import { counter, incCounter } from './lib';
9
10 // The imported value `counter` is live
11 console.log(counter); // 3
12 incCounter();
13 console.log(counter); // 4
14
15 // The imported value can't be changed.
16 counter++; // TypeError

```

Listing 9: Importe sind Views auf exportierte Entitäten

In anderen Worten jeder Import ist eine Live-Verbindung zu den exportierten Daten. AxelRauschmayer2018 zeigt die Unterschiede zwischen verschiedenen Importen in ES6:

- Unqualified imports wie zum Beispiel `import x from 'foo'` verhalten sich wie mit `const` deklarierte Variablen.
- Wird ein gesamtes Module Objekt importiert `import * as foo from 'foo'`, so verhält es sich wie ein mit `Object.freeze` geschütztes Objekt.

Gemäß ECMAScript (2015) erzeugt die Methode `CreateImportBinding(N, M, N2)` eine unveränderbare indirekte Bindung zu dem Exporten eines anderen Moduls. Dabei wird überprüft ob bereits ein Binding mit dem Namen **N** existiert. **M** ist die Bezeichnung des Moduls und **N2** ist der Name des von **M** exportierten Objekts. Im Weiteren ist ebenfalls zu beachten, dass es nicht möglich ist die Werte von Importen zu ändern, man kann jedoch die Objekte ändern auf welche verwiesen wird. (AxelRauschmayer2018)

```

1 //----- lib.js -----
2 export let obj = {};
3
4 //----- main.js -----
5 import { obj } from './lib';
6
7 obj.prop = 123; // OK
8 obj = {}; // TypeError

```

Listing 10: Importe sind Views auf exportierte Entitäten

Exporte in ES6 werden über den Export Eintrag verwaltet. Bis auf reexportierte Module besitzen alle Einträgen die folgenden 2 Namen:

- Local Name: Ist der Name unter dem der Export im Module gespeichert ist.
- Export Name: Importierende Module verwenden diesen Namen, um Zugang zu dem Export zu erhalten. Nachdem importieren einer Entität, wird diese stets mittels des Pointers erreicht. Der Pointer besteht dabei aus 2 Teilen Modul und Local Name und referenziert auf ein Binding innerhalb des Moduls.

(AxelRauschmayer2018)

Die nachfolgende angepasste Tabelle aus ECMAScript (2015) gibt einen Überblick über die Local und Export Namen verschiedener Export Varianten:

Tabelle 1: Local und Export Namen verschiedene Export Varianten

Export Statement Form	Export Name	Local Name
<code>export var v;</code>	v	v
<code>export default function f(){};</code>	default	f
<code>export default function(){};</code>	default	*default*
<code>export default 42;</code>	default	*default*
<code>export {x};</code>	x	x
<code>export {v as x};</code>	x	v
<code>export {x} from "mod";</code>	x	null
<code>export {v as x} from "mod";</code>	x	null
<code>export * from "mod";</code>	null	null

3.3.6 Syntax

```

1 import name from "module-name";
2 import * as name from "module-name";
3 import { member } from "module-name";
4 import { member as alias } from "module-name";
5 import { member1 , member2 } from "module-name";
6 import { member1 , member2 as alias2 , [...] } from "module-name";
7 import defaultMember, { member [ , [...] ] } from "module-name";
8 import defaultMember, * as alias from "module-name";
9 import defaultMember from "module-name";
10 import "module-name";

```

Listing 11: Import Möglichkeiten in ES6

```

1 export { name1, name2, ..., nameN };
2 export { variable1 as name1, variable2 as name2, ..., nameN };
3 export let name1, name2, ..., nameN; // oder: var
4 export let name1 = ..., name2 = ..., ..., nameN; // oder: var, const
5
6 export default expression;
7 export default function (...) { ... } // oder: class, function*
8 export default function name1(...) { ... } // oder: class, function*
9 export { name1 as default, ... };
10
11 export * from ...;
12 export { name1, name2, ..., nameN } from ...;
13 export { import1 as name1, import2 as name2, ..., nameN } from ...;

```

Listing 12: Export Möglichkeiten in ES6

4 Tree Shaking

Tree Shaking bedeutet im Javascript Umfeld Dead-Code Elimination. Der Begriff und das dahinterliegende Konzept wurde durch den ES2015 Modul Bundler Rollup populär. Das Konzept des Tree Shaking beruht auf der statischen Struktur von ES6 Modulen. Diese werden im Zusammenhang mit Bundlern wie Webpack oder RollupJS oft auch als Pure Modules oder Modules oder Harmony Modules bezeichnet. (WebpackTreeShaking 2018)

Wie bereits im Kapitel 3 gezeigt wurde, ermöglicht die ES6 Spezifikation, sowohl die statische Analyse von Modulen, als auch die Verwendung einer Teilmenge der zur Verfügung gestellten Exporte. Für das nachfolgende Beispiel wurde Webpack verwendet.

```

1 export function hello() {
2   return 'hello';
3 }
4 export function world() {
5   return 'world';
6 }
7
8 export default {

```

```
9   hello,  
10  world  
11 }
```

Listing 13: helpers.js

```
1 import {hello} from './helpers';  
2  
3 let elem = document.getElementById('output');  
4 elem.innerHTML = `Output: ${hello()}`;
```

Listing 14: index.js

Listing 14 zeigt die Verwendung des zuvor Implementierten `helpers.js` Moduls. Dabei ist zu beachten das die lediglich die Methode `hello` importiert und verwendet wird. `world` gilt somit als Dead Code oder Unused Export und wird in der gebündelten Anwendung nicht benötigt.

Damit Webpack

```
79 /* 1 */  
80 /**/ (function(module, __webpack_exports__, __webpack_require__) {  
81  
82   "use strict";  
83   /* harmony export (immutable) */ __webpack_exports__["a"] = hello;  
84   /* unused harmony export world */  
85   function hello() {  
86     return 'hello';  
87   }  
88   function world() {  
89     return 'world';  
90   }  
91  
92   /* unused harmony default export */ var _unused_webpack_default_export = ({  
93     hello: hello,  
94     world: world  
95   });
```

Listing 15: bundle.js

Um die unnötige Codezeilen zu entfernen werden diese zuerst von Webpack markiert. In Listing 15 wurde, sowohl der Default Export, als auch die `world` Funktion mit `unused harmony` markiert.

Webpack entfernt den unerwünschten Code nicht, erst der darauf folgenden Minifier erkennt die `unused harmony` Annotationen und entfernt diese Segmente.

Die daraus resultierende Bundle Datei beinhaltet eine stark verkleinerte `hello` Funktion und die nicht benötigte Methode `world` wurde aus der Datei entfernt. Auch wenn in diesem Beispiel der Gewinn nicht groß ist, kann Tree Shaking dazu beitragen die Bundle Datei in größeren Projekte signifikant zu verkleinern. (WebpackTreeShaking 2018)

4.1 Webpack

Um Tree Shaking mit Webpack zu ermöglichen sind einige Konfigurationen nötig. Zuvor ist es wichtig einen genaueren Blick darauf zu werfen wie Webpack mit ES6 Modulen arbeitet.

Eine große Schwierigkeit bei der Verwendung von ES6 ist die Browser Inkompatibilität. Erstens dauert es eine gewisse Zeit bis die neuen Spezifikationen in den Browsern von den Herstellern implementiert werden können (CanIUse 2018b). Zweitens benutzt ein beachtlicher Teil der Clients nicht die aktuellste verfügbare Version (CanIUse 2018a).

Um das Problem mit der Inkompatibilität zu umgehen werden JavaScript Compiler wie Babel verwendet. Diese verwandeln Es6 in einen für die meisten Browser Kompatiblen JavaScript Code. Somit können Entwickler und Entwicklerinnen bereits die neusten Funktionen verwenden ,ohne das diese von dem Großteil der Browser unterstützt werden muss.(Babel 2018)

Per Default wird von Babel jedes JavaScript Modul in CommonJS konvertiert. Durch die dynamische Implementierung von CommonJS Modulen können diese jedoch nicht statisch Analysiert werden. Es ist daher entscheidend um ein erfolgreiches Tree Shaking zu ermöglichen, dies zu ändern.(Babel 2018)

```
1 {
2   entry: './index.js',
3   output: {
4     filename: 'dist/es6-modules/bundle.js'
5   },
6   module: {
7     rules: [
8       {
9         test: /\.js$/,
10        exclude: /node_modules/,
11        loader: 'babel-loader',
12        query: {
13          presets: [
14            ['es2015', { modules: false }]
15          ]
16        }
17      }
18    ]
19  }
20 }
21 });
```

Listing 16: webpack.config.js

Listing 16 zeigt eine typische Tree Shaking Konfiguration von Babel und Webpack. Besonders wichtig ist die `modules: false` Option von Babel. Mit dieser kann man die automatische Modul Transformationen verändern. Neben dem `false` Wert der diese komplett deaktiviert, ist es ebenfalls möglich andere Optionen anzugeben (`"amd"` | `"umd"` | `"systemjs"` | `"commonjs"`). (Babel 2018)

4.2 RollupJS

Neben Webpack wurde das Tree shaking Konzept vor allem durch den ES6 Module Bundler RollupJS bekannt (WebpackTreeShaking 2018). Dieser wird vor allem empfohlen beim Entwickeln von Libraries. Webpack wiederum bietet eine bessere Unterstützung für Assets wie Bilder oder CSS und gilt daher als besseres geeignet für die Erstellung von Apps. (Rich Harris 2018)

Der Unterschied zwischen den Bundlern liegt in der Art und Weise wie die einzelnen Module im Bundle File verpackt sind. Webpack hüllt jedes Modul in eine Funktion mit einer browserfreundlichen Implementierung von `require` (Rich Harris 2018). Features wie On-demand Loading werden dadurch erst möglich. Nolan Lawson (2018) konnte jedoch zeigen das dies im weiteren auch dazu führt, dass je mehr Module verwendet werden auch der damit einhergehende Overhead wächst.

RollupJS verwendet im Gegensatz die Möglichkeiten von ES6. Der gesamte Code landet an einem Platz und wird in einem Durchgang verarbeitet. Damit wird der Code schlanker und kann schneller starten. (Rich Harris 2018)

4.3 Hindernisse

In dieser Arbeit wird Tree Shaking aus der Perspektive zweier Entwicklergruppen betrachtet.

Zum einen jene Entwickler und Entwicklerinnen welche viel projektinternen JavaScript Code in ihren Webpack Projekten verwenden. Ihr Ziel ist es eine oder mehrere möglichst kleiner Bundle Dateien an den Client auszuliefern, für schnelle Seitenaufrufe und eine angenehmes Benutzererlebnis.

Die zweite Gruppe ist jene der Javascript Library und Framework Entwickler. Ihr Sourcecode wird von vielen Projekten in verschiedener Art und Weise verwendet. In vielen Fällen wird in den Javascript Projekten nur ein kleiner klein der importierten Libraries verwendet. Wenn diese Tree Shaking unterstützt, können jedoch Bundler wie Webpack, den unnötig Code beim kompilieren der Bundle Datei entfernen.

In diesem Kapitel werden mehrerere Hindernisse für die beiden Benutzergruppen veranschaulicht und in weitere Folge Maßnahmen zu Vermeidung dieser ermittelt.

(<http://www.syntaxsuccess.com/viewarticle/tree-shaking-with-webpack>)

4.3.1 Richtiges Importieren und Exportieren

Wie bereits in den Kapitel 3.3.2 und 3.3.1 beschrieben können Javascript Module mittels Naming oder Default Export

nicht `import Test from 'module1'` `Test.func1()` besser `import func1 from 'module1'`

4.3.2 Exportieren

keine default exporte ?

alles was importiert wird sollte auch exportiert werden (stimmt nicht ganz es können sideeffects enthalten sein sideeffect: [*.css])

4.3.3 Side Effects

Umfangreiche und weitverbreitete JavaScript Libraries wie zum Beispiel Lodash¹ und jQuery² bieten eine große Anzahl an nützlichen Tools. Sie werden meistens als NPM Modul installiert und anschließend in den Projektdateien als Import eingebunden. Zu meist werden nur Teile von den zu Verfügung gestellten Funktionalitäten auch wirklich verwendet. Das kann dazu führen das ein erheblich Teil der von Webpack erzeugten Bundledatei aus nicht benötigten Code besteht. Plugins wie der Webpack Visualizer bieten eine einfache Möglichkeit, um festzustellen, welche Module in der von Webpack generierten Bundle Datei am meisten Speicherplatz einnehmen.

Jedoch bietet unter anderem Webpack eine Möglichkeit für Library Entwickler und Entwicklerinnen diese so zu Implementieren, dass der nicht benötigte Library Sourcecode beim builden entfernt wird. Durch die Verwendung von `sideEffects: false` in der package.json Datei innerhalb der Library, wird Webpack mitgeteilt das Tree Shaking nicht nur auf den üblichen Projektcode anzuwenden, sondern ebenfalls auf den Code der sich im Library Ordner `node_modules` befindet.

Dabei werden Importdeklarationen wie `import {a, b} from "big-module-with-flag"` von Webpack erkannt und beim Bau der Applikation zu `import a from "big-module-with-flag/a" import b from "big-module-with-flag/b"` umgeschrieben. Somit werden in der Bundle Datei auch nur die zwei von dem Entwickler oder der Entwicklerin verwendeten Methoden importiert und der restliche Library Code entfernt (WebpackTreeShaking 2018). Zur Veranschaulichung dient ein ausführlich dokumentiertes Beispiel³.

Wenn mit Hilfe von `sideEffects: false` ein Modul als rein bezeichnet wird, werden alle nicht verwendeten Methoden, Klassen oder Objekte aus dem Code entfernt. Dies kann jedoch die Funktionsweise erheblich stören und SideEffects können entstehen. Diese sind Code Segmente, welche beim Tree Shaking nicht entfernt werden dürfen. Dies wird im Weiteren mittels eines Beispiels näher erläutert.

```
1  import a from 'a'
2    import b from 'b'
3
4    console.log("das ist ein sideeffect")
5
6    export {
```

1. <https://lodash.com/> - besucht am 26.05.2018

2. <https://jquery.com/> - besucht am 26.05.2018

3. <https://github.com/webpack/webpack/tree/master/examples/side-effects> - besucht am 26.05.2018

```
7     a,  
8     b  
9 }
```

Listing 17: Library mit Side Effects

```
1 import { a } from 'lib'  
2  
3 console.log("a:", a)
```

Listing 18: Verwendung der Library

Listing 17 zeigt ein Beispiel für einen Library Entry Point. Dieser exportiert alle Funktionalitäten, die zur Verfügung gestellt werden.

Beim Tree Shaking werden alle Imports umgeschrieben. Dabei wird `import { a } from 'lib'` in Listing 18 zu `import a from 'lib/a'` und somit wird jeglicher Programmcode in `lib.js` nicht ausgeführt. Die `console.log` Nachricht in Zeile 4 wird somit nie angezeigt. Dies führt vor allem dann zu Problemen, wenn sich Side Effects auf Exporte in der Datei auswirken.

```
1 import { a } from 'lib'  
2  
3 a.value = 'this value is dangerous'  
4  
5 export {  
6     a  
7 }
```

Listing 19: Side Effect mit Auswirkung auf Exporte

In Listing 19 wird wie in Listing 17 `a` exportiert. Davor wird in der Zeile 3 die `importantValue` Eigenschaft hinzugefügt. Bei aktivem Tree Shaking wird `a` jedoch direkt mit `import a from 'lib/a'` importiert. `a.value` wird somit nie angelegt und kann nicht verwendet werden. Diese Unterschiede sind in späterer Folge für Benutzer und Benutzerinnen nur schwer nachzuvollziehen und Entwickler und Entwicklerinnen sollten bei der Erstellung einer Library unbedingt diese vermeiden. (WebpackTreeShaking 2018)

Mit `sideEffects: false` betrachtet Webpack den gesamten Projektcode als rein von Side Effects. Sind für die ordnungsgemäße Ausführung jedoch einige Importe nötig, können Entwickler und Entwicklerinnen diese als String Array angeben. Dieses akzeptiert relative Pfade, absolute Pfade und glob Pattern zu den relevanten Dateien. Webpack verwendet Micromatch⁴ zur Auflösung der Pfade. (WebpackTreeShaking 2018)

CSS Dateien sind ein gutes Beispiel für Importe, die in jedem Fall in dem gebündelten Programmcode enthalten sein müssen. Bei der Verwendung des CSS-Loader und dem Importieren in einer Projektdatei, wird der CSS Code analysiert und die darin befindlichen Klassen stehen zur Verfügung. Sollten diese aber durch Tree Shaking entfernt werden, fehlen die CSS Klassen und die dazugehörigen Styles in der Anwendung. (WebpackTreeShaking 2018)

4. <https://github.com/micromatch/micromatch> - besucht am 21.06.2018

```

1  import './styles.css'
2
3  console.log('sideeffectful file')
4
5  export const sideeffect = 'sideeffect'

```

Listing 20: sideeffectful.js mit CSS Side Effect

```

1  .myClass {
2      background-color: blue;
3  }

```

Listing 21: styles.css

Listing 20 zeigt ein JavaScript Modul, welches die CSS Datei `styles.css` importiert. Mit der `sideEffects: false` Webpack Konfiguration wird `import './styles.css'` in Zeile 1 als `unused export` erkannt und ist in den gebündelten Anwendung nicht mehr vorhanden. `console.log('sideeffectful file')` und `export const sideeffect = 'sideeffect'` werden jedoch noch ausgeführt. Im Gegensatz dazu wird `sideEffects: ['*.css']` in der Konfigurationsdatei verwendet. Wird der CSS Import mit in die Bundle Datei übernommen. Auch wenn dies wegen dem Minify Schritt nicht sofort erkenntlich ist. Ist folgendes Code Segment darin zu finden `function(e,n,o){(e.exports=o(2)(!1)).push([e.i,".myClass {\n background-color: blue;\n}",""])}.`(WebpackTreeShaking 2018)

Anhand dieser Eigenschaften wurden die folgenden Annahmen in Bezug zur Vermeidung von Side Effects ermittelt, diese werden bei der Implementierung berücksichtigt und am Ende diese Arbeit auf ihre Wirksamkeit überprüft.

- Besitzt ein Modul Named Exports ist darauf zu achten, dass auf diese keine SideEffects wirken.
- Bei der Verwendung eines Default Exports wird die Gesamte Datei eingebunden. Jegliche Side Effects innerhalb der Datei werden somit inkludiert.
- Importe welche den Body eines Moduls ausführen werden von Rauschmayer (2018) als Empty Imports bezeichnet, können beim bündeln entfernt werden und müssen deshalb als SideEffect in der `package.json` Datei vermerkt werden.
- Der Entry Point einer Library sollte für sich eigenständige Module importieren und diese als Named Exports zur Verfügung stellen. Dieser sollte keine SideEffects beinhalten.

(named export sind ok solange se nicht nur imports sind)

—————BSP von webpack—————

-des weiteren muss jeder import auch exportiert werden <https://stackoverflow.com/questions/49160752/what-does-webpack-4-expect-from-a-package-with-sideeffects-false>

(Tamás Sallai 2018)

5 Linting

Der Begriff Lint wurde erstmals in den 1970er in Verbindung mit der Softwareentwicklung erwähnt. Johnson (1978) entwickelte es an den Bell Laboratories um die Schwächen der damaligen C Compiler auszugleichen und unentdeckte Fehler in bereits kompilierten Programmen zu finden.

Es existieren viele Wege um die Anzahl an Bugs innerhalb einer Anwendung zu minimieren. Neben dem schreiben von Unit Tests, bieten Code Reviews laut Louridas (2006) wohl die beste Möglichkeit Bugs und Code Smells aufzufinden und zu eliminieren. Es fordert jedoch einen großen Zeitaufwand mehrere Entwickler zusammenzubringen und die gesamte Codebasis einer Anwendung gemeinsam zu reviewen.

Die meisten Fehler fallen in die Kategorie Known Errors. Dies sind häufig auftretende sich wiederholende Fälle in denen die Entwickler und Entwicklerinnen immer wieder hinstopplern. Mit Linting möchte man die sich ständig wiederholenden Fehler oder auch Code Smells so früh es möglich ist finden und ausbessern. (Louridas 2006)

Im Gegensatz zu einem Compiler muss beim Linting der Code nicht ausgeführt werden. So genannte Static Checker durchlaufen den Programmcode auf der Suche nach bestimmten Mustern, diesen Prozess wird als Statische Code Analyse bezeichnet. (Louridas 2006)

Es gibt verschiedene Wege um eine Statische Code Analyse durchzuführen. Wie zum Beispiel auf Anfrage eines Entwicklers oder einer Entwicklerin, kontinuierlich während der Erstellung einer Anwendung oder aber direkt bevor ein Entwickler oder eine Entwicklerin Codeänderungen auf ein Repository committet will. Je nach Projekt können somit Fehler entdeckt und behoben werden bevor diese in der Codebasis am Repository landen. (Johnson u. a., o.Dat.)

Es existiert bereits reichlich Literatur zur Statischen Code Analyse. Dabei wurden nicht nur Aspekte wie Performance und Genauigkeit untersucht (Bessey u. a. 2010), sondern auch verschiedene Einsatzgebiete gezeigt (Bush u. a., o.Dat.).

Johnson u. a. (o.Dat.) erforschten in ihrer Arbeit den Gebrauch und die Verbreitung von Statischen Analyse Tools. Ein Teil davon war die Ermittlung was Entwickler und Entwicklerinnen als bei dem Gebrauch diese Werkzeuge als störend empfinden. Ergebnis davon war der nicht zu unterschätzende Anteil an false positives, sprich die Anzahl an Warnungen die keine wirklichen Fehler sind. Im weiteren wurde auch noch die hohe Arbeitslast von Entwicklern und Entwicklerinnen genannt. Laut Johnson u. a. (o.Dat.) sollten zukünftige Tools, sowohl eine besser Integration in den Arbeitsabläufen von Entwicklern und Entwicklerinnen bieten, als auch die Arbeit in einem Team besser unterstützen.

Bush u. a. (o.Dat.) implementierte ein statisches Analyse Tool zum auffinden von dynamischen Fehler in C und C++. Dazu zählen unter anderem die Verwendung von nicht initialisierten Speicher oder falschen Operation auf Dateien (zum Beispiel das Schließen einer Datei die bereits geschlossen wurde). Das aus der Arbeit entstandene Tool wurde PREFIX getauft. Zur Analyse von Code wird der Abstract Syntax Tree generiert und anschließend die Funktionsaufrufe mittels eines Topologischen Algorithmus sortiert. Die Funktionen werden daraufhin simuliert und darin befindliche Defekte gemeldet. Als Teil dieser Arbeit wird ein Plugin erstellt, welches wie bei

Bush u. a. (o.Dat.) den Abstract Syntax Tree auf ähnliche Art und Weise untersucht. Jedoch soll dies für die Programmiersprache JavaScript erfolgen und sich lediglich auf fehlerhafte Muster in der Verwendung von ES6 Imports und Exports fokussieren.

5.1 ES Lint

2013 erschuf der Entwickler Nicholas C. Zakas ESLint. Ein erweiterbares Linting Tool für JavaScript und JSX. Neben vielen bereits vorhandenen Linting Regeln bietet ES Lint eine Möglichkeit für Entwickler und Entwicklerinnen solche auch selbst zu implementieren. Durch hinzufügen oder entfernen von Regeln wird der Linting Process je nach Projekt und Team konfiguriert. Dabei gilt für jede Regel die folgenden drei Grundsätze:

- Jede Regel sollte eigenständig funktionieren
- Bietet eine Möglichkeit zum An und Abschalten (nicht darf als zu wichtig zum deaktivieren gelten)
- Es kann zwischen Warnung und Fehler gewechselt werden

Um nicht jede Regel einzeln zu importieren gibt es ebenfalls die Möglichkeit mehrere Regeln als Bundle auszuliefern. (ESLintAbout 2018)

Ziel dieser Arbeit ist es bereits implementierte Regeln für die Optimierung von ES6 Exports und Imports zu identifizieren und diese zu einem Bundle zusammenzufügen. Des Weiteren werden ebenfalls eigene Regeln für ESLint erstellt, welche weitere schlechte Muster im Programmcode aufdeckt und auch Lösungen dafür anzeigt. (ESLintAbout 2018)

(Kurzer Einleitungstext für die nächsten Kapitel)

5.1.1 Core Rules

ESLint wird bereits mit mehr als 200 allgemein gültige Regeln ausgeliefert. Diese werden als Core Regeln bezeichnet. Gemäß (ESLintNewRules 2018) muss eine eigene Regel die folgenden 6 Eigenschaften erfüllen um diese der Core Liste hinzuzufügen:

1. Allgemeine Anwendbarkeit Core Regeln sollten relevant für eine große Anzahl an Entwickler und Entwicklerinnen sein. Regeln für individuelle Präferenzen oder Edge Cases werden nicht akzeptiert.

2. Generisch Neue Regeln sollten möglichst generisch sein. Ein Benutzer oder eine Benutzerin darf kein Problem haben zu verstehen wann diese zu benutzen ist. Als Richtlinie gilt eine Regel ist zu spezifisch wenn man zur Beschreibung was sie macht mehr als zwei „und“ benötigt. (zum Beispiel: Wenn a und b und c und d, dann wird der Benutzer oder die Benutzerin gewarnt)

3. Atomar Regeln sollten vollkommen selbstständig funktionieren. Abhängigkeiten zwischen zwei oder mehreren Regeln sind nicht erlaubt.

4. Einzigartig Überschneidung zwischen Regeln sind ebenfalls nicht erlaubt. Jede sollte eine eigenen Warnung erzeugen. Somit wird der Benutzer nicht unnötig verwirrt.

5. Unabhängig Regeln dürfen nur auf der JavaScript Laufzeitumgebung basieren und unabhängig von Libraries oder Frameworks sein. Core Regeln sollten stets anwendbar sein und nicht davon abhängen ob spezielle Libraries wie JQuery verwendet werden. Hingegen existieren bereits einige Regeln die nur angewendet werden wenn die Laufzeitumgebung NodeJS vorhanden ist.

6. Ohne Konflikte Keine Regel sollte sich mit anderen Regeln überschneiden. Zum Beispiel gibt es eine Regel, welche von einem Entwickler oder einer Entwicklerin verlangt jedes Statement mit einem Semikolon zu beenden. Es ist nicht erlaubt eine neue Regel zu entwerfen, die Semikolons verbietet. Statt dessen existiert die Regel Semi, diese kann beides und kann mit Hilfe der Konfiguration gesteuert werden.

(ich mache keine core regeln wegen dem 1. punkt - die anderen Eigenschaften sollten jedoch erfüllt werden)

<https://eslint.org/docs/developer-guide/working-with-rules>

(Runtime Rules vs Core Rules)

5.1.2 Komponenten eines Plugins

<https://eslint.org/docs/developer-guide/working-with-plugins> (kurze einleitung danach wie wird das in meinem plugin verwendet?) **Rules Environments Configs Processors**

5.1.3 How to use

Client config verwendung

5.1.4 Formatters

<https://eslint.org/docs/user-guide/formatters/>

5.2 Dependency Graph

Neben statischen Mustern wie nicht initialisierten Funktionen und Variablen oder Formatierungsfehler, gibt es eine weitere Kategorie, die der dynamischen Bugs. Dabei werden unter anderen die Ausführungspfade innerhalb des Applikation Codes verfolgt und Fehler in der dynamischen Natur einer Programmiersprache entdeckt. (Bush u. a., o.Dat.)

—todo—

Static Checker können Fehler in verschiedener Art entdecken und markieren. Dazu zählen: Code Smells, Formatierungs Fehler und Fehler im Ablauf

6 Untersuchung von JavaScript Libraries

Es existieren bereits Projekte auf NPM die Tree Shaking verwenden. Durch eine Sourcecode Analyse sollen zusätzliche Erkenntnisse gewonnen werden, über die zum Einsatz kommenden Design Patterns und Möglichkeiten die Entwickler und Entwicklerinnen mit Linting bei der Implementierung dieser zu unterstützen.

In weitere Folge werden am Ende dieser Arbeit, die ausgewählten Projekte mit dem ES Lint Tree Shaking Plugin auf Fehler und Warnungen getestet. Dabei sollten keine Fehler angezeigt werden, um die richtige Funktionweise des Plugins sicherzustellen.

6.1 Auswahlkriterien

Die wichtigsten Kriterien für die zur Analyse ausgewählten Projekte ist zum einen die Verwendung eines Static Bundler und zum anderen eine bereits bestehende Implementierung von Tree Shaking. Um dies festzustellen wird zuerst geprüft ob eine package.json Datei im Repository existiert. Anschließend wird in dieser nach `sideeffects` gesucht. Wie bereits im Kapitel 4.3.3 veranschaulicht die `sideeffect` Einstellung dazu dem Bundler anzuzeigen, dass nur die verwendeten Exporte in dem gebuildeten Programmcode benötigt werden.

Die ausgewählten JavaScript Repositories sollten für eine möglichst große Anzahl an Entwicklern und Entwicklerinnen von relevanter Bedeutung sein. Für die Auswahl werden daher jene NPM Module verwendet, welche am häufigsten in Webprojekten verwendeten werden. Es existiert bereits eine generische Liste ⁵ die diese Module beinhaltet.

Auf NPM befinden sich neben Bibliotheken welche ES6 verwenden, auch welche die für NodeJS entwickelt wurden. Dabei werden der NodeJS Module verwendet und sind daher für diese Arbeit nicht geeignet (NodeJSModules 2018). Ein weiteres Kriterium für die Auswahl der zu untersuchenden Projekte ist daher die Verwendung von ES6 Modulen.

Anhand dieser Auswahlkriterien wurde folgende drei Projekte ermittelt, welche Tree Shaking und ES6 implementieren:

- `lodash`⁶
- `rxjs`⁷
- `graphql`⁸

5. <https://www.npmjs.com/browse/depended> - besucht am 06.06.2018

6. <https://github.com/lodash/lodash/tree/es> - besucht am 15.06.2018

7. <https://github.com/ReactiveX/rxjs> - besucht am 15.06.2018

8. <https://github.com/graphql/graphql-js> - besucht am 15.06.2018

6.2 Implementierung von Tree Shaking

In den ausgewählten Repositories wurde Tree Shaking mit `SideEffects: false` in der `package.json` Datei aktiviert. Somit werden bei der Verwendung Named Imports wie zum Beispiel `import {add} from 'lodash'`, diese aufgelöst zu `import add from 'lodash/add'`. In der weiteren Folge wird sämtlicher nicht genutzter Programmcode der Library entfernt und nur die verwendeten Module in die Bundle Datei integriert.

Webpack startet mit der Erzeugung des Bundles bei dem definierten Entry Point. In den untersuchten Repositories dient diese Datei, um die gewünschten Funktionalitäten nach außen zu exportieren. Wie bereits im Kapitel 4.3.3 festgestellt wurde können mögliche Sideeffects in dieser Datei entfernt werden. In allen drei Fällen werden daher lediglich `import` und `export` verwendet.

Da Programmcode außerhalb des benötigten Moduls nicht garantiert in der Bundle Datei enthalten ist, wurde der gesamte Sourcecode in für sich eigenständige Module aufgeteilt. Dies bedeutet, dass Module wie zum Beispiel `add.js` alle Ressourcen, welche für die ordnungsgemäße Funktionalität benötigt werden, importieren müssen.

7 Erstellen des Plugins

7.1 Setup

Um die Projektstruktur aufzusetzen verwende ich die von ESLintNewRules (2018) empfohlene Vorgehensweise. Eine wichtige Voraussetzung um ein ES Lint Plugin zu erstellen sind NPM und NodeJS, diese müssen zuerst installiert werden. In dieser Arbeit wird NodeJS v8.0.0 und NPM 5.8.0 verwendet.

Des Weiteren wird das Commandline Tool Yeoman benötigt. Die aktuellste Version ist 2.0.2, diese wird mit dem Befehl `npm install -g yo` global installiert und steht somit in der Konsole an jedem Ort zur Verfügung. Zuletzt wird das NPM Modul `eslint-generator` installiert es dient dazu ein leeres Gerüst für den Plugin Code zu erstellen.

Nach der Installation wird mit dem Befehl `yo eslint:plugin` der Generator gestartet. Folgende Daten wurden dabei angegeben:

- **Name:** silltho
- **Plugin-Id:** threeshaking
- **Beschreibung:** –todo–
- **Werden Benutzerdefinierte Regeln verwendet?:** Ja
- **Werden Processors verwendet?:** Nein

Die daraus resultierende `package.json` Datei sieht wie folgt aus:

```
1 {
2   "name": "eslint-plugin-treeshaking",
3   "version": "0.0.0",
4   "description": "test",
5   "keywords": [
6     "eslint",
7     "eslintplugin",
8     "eslint-plugin"
9   ],
10  "author": "silltho",
11  "main": "lib/index.js",
12  "scripts": {
13    "test": "mocha tests --recursive"
14  },
15  "dependencies": {
16    "requireindex": "~1.1.0"
17  },
18  "devDependencies": {
19    "eslint": "~3.9.1",
20    "mocha": "^3.1.2"
21  },
22  "engines": {
23    "node": ">=0.10.0"
24  },
25  "license": "ISC"
26 }
```

Listing 22: generierte package.json Datei

In weiterer Folge wurde neben der `README.md` Datei auch ein `lib` und `tests` Ordner im Projektverzeichnis erstellt. Der Ordner `lib/rules` wird später den Programmcode für die neu erstellten Regeln beinhalten.

7.2 Entwicklungsumgebung

Für die Programmierung wird die integrierte Entwicklungsumgebung (IDE) WebStorm von JetBrains in der Version 2018.1.2 verwendet.

`-eslint 3.9.1 -astexplorer -mocha test suite -prettier -husky`

8 Die Implementierung neuer Regeln

Yeoman und der ESLint-Generator unterstützen Entwickler und Entwicklerinnen nicht nur beim Erzeugen der Projektdatendateien sondern ebenfalls bei dem Anlegen neuer Regeln innerhalb des Plugins. Die dafür nötigen Dateien werden mit dem Befehl `yo eslint:rule` erzeugt. Die neu erstellten Regeln innerhalb des Plugins sollen die Aufmerksamkeit von Entwickler und Entwicklerinnen auf die Probleme aus 4.3 richten und Wege aufzeigen um diese zu vermeiden. In diesem Kapitel wird dokumentiert wie die Regeln erstellt wurden.

Beim Aufruf des Generators mit `yo eslint:rule` müssen mehrere Fragen beantwortet werden:

- What is your name?
- Where will this rule be published?
- What is the rule ID?
- Type a short description of this rule
- Type a short example of the code that will fail

In dieser Arbeit wird als Name `silltho` verwendet. Für den Publishing Ort kann zwischen ESLint Core und ESLint Plugin gewählt werden. Wie bereits im Kapitel 5.1.1 beschrieben wurde, eignen sich die in dieser Arbeit erstellten Regeln nicht als Core Regeln, es wird somit bei allen Regeln die Option ESLint Plugin gewählt.

Alle weiteren Fragen können je nach Regel unterschiedlich beantwortet. In weiterer Folge werden diese als Tabelle bei der Implementierung jeder Regel dargestellt.

Vom Generator werden abschließend die folgenden Dateien im Projektverzeichnis erzeugt:

- **docs/rules/(rule-id).md** - Dokumentation der Regel und der Konfiguration Möglichkeiten.
- **lib/rules/(rule-id).js** - Programmcode.
- **tests/lib/rules/(rule-id).js** - Tests um sicherzustellen das die Regel wie gewünscht funktioniert.

Die nachfolgenden Kapitel beschreiben die Implementierung der Regeln die benötigt werden um Code Smells beim Tree Shaking für Entwickler und Entwicklerinnen anzuzeigen.

8.1 Empty Imports

Tabelle 2: Yeoman Generator: Empty Imports

Rule-ID:	empty-imports
Beschreibung:	todo
Beispiel für fehlerhaften Code:	todo

Gemäß den Erkenntnissen aus Kapitel 4.3.3, müssen Empty Imports wie zum Beispiel `import 'module1'` als Side effect gekennzeichnet werden. Diese werden sonst beim bündeln des Programmcodes als Unused Imports erkannt und entfernt. Ziel dieser Regel ist es die Aufmerksamkeit von Entwicklern und Entwicklerinnen auf diese Art von Imports zu richten und sie daran zu erinnern diese in der `package.json` Datei als SideEffect zu vermerken.

Um diese Funktionsweise zu gewährleisten wurden folgende Test Datei `test/lib/rules/empty-imports.js` erstellt:

```

1  var ruleTester = new RuleTester();
2  ruleTester.run("empty-imports", rule, {
3
4    valid: [
5      {
6        options: [['empty-import']],
7        code: `
8          import 'empty-import'
9          import lib1 from 'lib1'
10         export const export1 = 'export1'
11         export default 'default'
12       `,
13        errors: []
14      },
15      {
16        options: [],
17        code: `
18         import name from "module-name";
19         import * as name2 from "module-name";
20         import { member } from "module-name";
21         import { member as alias } from "module-name";
22         import { member1 , member2 } from "module-name";
23         import { member3 , member2 as alias2 } from "module-name";
24         import defaultMember, { member4 } from "module-name";
25         import defaultMember2, * as alias3 from "module-name";
26         import defaultMember3 from "module-name";
27       `,
28        errors: []
29      }
30    ],
31
32    invalid: [
33      {
34        code: `
35         import 'empty-import'
36         import lib1 from 'lib1'
37         export const export1 = 'export1'
38         export default 'default'
39       `,
40        errors: [{
41          message: 'empty imports are removed by tree shaking. Make
42                    sure you add empty-import to package.json sideEffects
43                    option.',
44          type: 'ImportDeclaration'
45        }]
46      }
47    ]
48  });

```

Listing 23: Empty Imports Unit Tests

Als ersten Schritt werden mit dem AstExplorer⁹ die verschiedenen Import Nodes aus dem Kapitel 3.3.6 analysiert. Der NodeType für Importe in ES6 ist `ImportDeclaration`. In weiterer Folge konnte festgestellt werden, dass Empty Import Nodes keine specifier besitzen, somit ist `importNode.specifiers.length === 0` true, falls es sich um einen Empty Import handelt. Mit Hilfe dieser Erkenntnisse ist es bereits möglich Empty Imports aufzuspüren und zu kennzeichnen.

```

1      function isEmptyImport(importNode) {
2          if(importNode.specifiers.length === 0) {
3              reportEmptyImport(importNode)
4          }
5      }
6
7      return {
8          'ImportDeclaration': isEmptyImport
9      };

```

Listing 24: Code zum Aufspüren von Empty Imports

Nach der Implementierung von Listing ?? werden zwei der drei Tests bereits erfolgreich durchgeführt. Beim ersten validen Test wird jedoch der Fehler `AssertionError [ERR_ASSERTION]: Should have no errors but had 1` angezeigt. Empty Imports können in der `package.json` Datei als SideEffects vermerkt werden. Diese werden anschließend im Programmcode inkludiert und können verwendet werden. Die Empty Imports Regel sollte dahingehend konfigurierbar sein.

Hierfür sollte man als Option ein Array von Pfaden und Glob Pattern übergeben, gleich zu der SideEffects Einstellung von Webpack. WebpackTreeShaking (2018) zeigt, dass für die Auflösung der SideEffect Property die Bibliothek Micromatch¹⁰ verwendet wird. Für die Auflösung der Array Daten innerhalb der Regel wird die selbe Library verwendet, um die Unterschiede zwischen Webpack und Regel Konfiguration möglichst gering zu halten.

Damit die Empty Imports Regel ein Array von Strings als Property akzeptiert muss die Schema Konfiguration in `lib/rules/empty-imports.js` wie folgt angepasst werden:

```

1      schema: [
2          {
3              type: 'array'
4          }
5      ]

```

Listing 25: Empty Import Option Schema

Anschließend kann auf die Konfiguration mittels `context.options[0]` zugegriffen werden.

```

1      function isKnownSideEffect(emptyImportNode) {
2          return mm.any(emptyImportNode.source.value, knownSideEffects,
3                          null)

```

9. <https://astexplorer.net/> - besucht am 26.06.2018.

10. <https://github.com/micromatch/micromatch> - besucht am 26.06.2018

```
3      }
```

Listing 26: isKnownSideEffect Funktion

In weiterer Folge wird eine zusätzliche Funktion `isKnownSideEffect` implementiert. Diese vergleicht den Value eines Imports mit den übergebenen Array. Micromatch bietet dafür die Methode `any(str, patterns, options)` an. Diese vergleicht den übergebenen String mit dem Patternsarray und liefert `true` falls ein beliebiges Pattern in dem Array mit dem String übereinstimmt.

Zuletzt muss noch die `isEmptyImport` angepasst werden, um mit der neuen Funktion alle Empty Imports zu überprüfen.

```
1      function isEmptyImport(importNode) {
2          if(importNode.specifiers.length === 0 && !isKnownSideEffect(
3              importNode)) {
4              reportEmptyImport(importNode)
5          }
6      }
```

Listing 27: isKnownSideEffect Funktion

Beim erneuten Ausführen der Tests werden alle erfolgreich abgeschlossen. Die Regel ist somit erfolgreich Implementiert und wird am Ende dieser Arbeit auf ihre Wirksamkeit geprüft.

8.2 Named Exports SideEffects

Tabelle 3: Yeoman Generator: Named Exports SideEffects

Rule-ID:	named-exports-sideeffects
Beschreibung:	todo
Beispiel für fehlerhaften Code:	todo

Ein weiterer Code Smell aus Kapitel 4.3.3 ist der Gebrauch von SideEffects auf Named Exports.

```
1  let counter = 1
2
3  const incCounter = (newValue) => {
4      counter++
5  }
6
7  export {
8      incCounter,
9      counter
10 }
```

Listing 28: counter.js

```

1  import { counter, incCounter } from "counter";
2  incCounter();
3  export { counter }

```

Listing 29: lib.js

```

1  import { counter } from "lib";
2  console.log(counter) // 1

```

Listing 30: index.js

Listing 28, 29 und 30 zeigen ein Beispiel für einen solchen SideEffect. Obwohl die Methode `incCounter` in `lib.js` aufgerufen wird, wird durch Tree Shaking `import { counter }` from `"lib"` zu `import { counter }` from `"counter"`. Somit wird die Datei `lib.js` nicht inkludiert und auch der `incCounter` Aufruf geht in weiterer Folge verloren.

Für die Implementierung werden Imports (`import`), Named Exports (`export`) und Expressions (`incCounter` oder `counter.test = '123'`) benötigt. Mit dem ASTexplorer wurden erneut die dem entsprechenden Node Typen `ImportDeclaration`, `ExportNamedDeclaration` und `ExpressionStatement` ermittelt.

Die folgende Test Datei, soll die Funktionsfähigkeit der Regel sicherstellen. Insgesamt wurden 4 valide und 4 invalide Szenarien ausgearbeitet.

```

1  var ruleTester = new RuleTester()
2  ruleTester.run('named-exports-sideeffects', rule, {
3    valid: [
4      {
5        options: [],
6        code: `
7          import name from "module-name";
8          import name2 from "module-name2";
9          name2.sideeffect = 'sideeffect'
10         export {
11           name
12         }
13         export default name2
14       `,
15        errors: []
16      },
17      {
18        options: [],
19        code: `
20         testFunction = () => {
21           console.log('test')
22         }
23         export {
24           testFunction
25         }
26       `,
27        errors: []
28      },
29    ]

```

```

30     options: [],
31     code: `
32         import name from "module-name";
33         import name2 from "module-name2";
34         console.log('test123')
35         export default name2
36     `,
37     errors: []
38 },
39 {
40     options: [],
41     code: `
42         import name from "module-name";
43         import name2 from "module-name2";
44         const tmp = 'tmp'
45         export {
46             tmp as name,
47             name as name2
48         }
49     `,
50     errors: []
51 }
52 ],
53
54 invalid: [
55     {
56         options: [],
57         code: `
58             import test2 from "module-name2";
59             test2.sideeffect = 'sideeffect'
60             export { test2 }
61         `,
62         errors: [
63             {
64                 message:
65                     'Effects on reexported modules (test2) could be prune by
66                     TreeShaking.',
67                 type: 'ExpressionStatement'
68             }
69         ],
70     },
71     {
72         options: [],
73         code: `
74             import test2 from "module-name2";
75             test2.init('initSomething')
76             export { test2 }
77         `,
78         errors: [
79             {
80                 message:

```



```

81         TreeShaking.',
82         type: 'ExpressionStatement'
83     ]
84 },
85 {
86     options: [],
87     code: `
88         import { counter as temp1, incCounter as temp2 } from "module-
89             name2";
90         temp2();
91         export { temp1 as tmp }
92     `,
93     errors: [
94         {
95             message:
96                 'Effects on reexported modules (temp1) could be prune by
97                 TreeShaking.',
98             type: 'ExpressionStatement'
99         }
100     ],
101     options: [],
102     code: `
103         import { counter, incCounter } from "module-name2";
104         incCounter();
105         export { counter }
106     `,
107     errors: [
108         {
109             message:
110                 'Effects on reexported modules (counter) could be prune by
111                 TreeShaking.',
112             type: 'ExpressionStatement'
113         }
114     ]
115 }
116 })

```

Listing 31: Named Exports SideEffects Unit Tests

Die invaliden Testfälle aus Listing 31 können in zwei Kategorien eingeteilt werden. Zum einen direkte SideEffects wie zum Beispiel `test2.init('initSomething')`. Dieser wirkt direkt auf das zuvor importierte `test2` Modul, welches anschließend als Named Export exportiert wird. Die zweite Kategorie werden in dieser Arbeit als indirekte SideEffects bezeichnet. Ein Beispiel dafür ist der Aufruf der Methode `incCounter()` in Listing 29. Dieser hat Auswirkungen auf den Export von `counter`. Da die Variable `counter` jedoch nie verwendet oder verändert wird, sind diese wesentlich schwerer zu erkennen als direkte Side Effects.

Für die Erkennung der direkten Side Effects werden alle Imports und Named Exports in einem

Array gespeichert. Dies geschieht mit der Hilfe des `ImportDeclaration` und `ExportNamedDeclaration` Node Selectors. Diese rufen die Funktionen `saveImport` und `saveNamedExport` auf. Darin werden die einzelnen Specifier zu in Array gespeichert. Zum Beispiel bei folgenden Export `export {test1, test2}` werden jeweils die Specifier Node `test1` und `test2` gespeichert. Als nächsten werden alle SideEffects mit dem `ExpressionStatement` Selector und der Funktion `saveExpression` ebenfalls in einem Array vermerkt. Schließlich kann mit dem Selector `Program:exit` am Ende eines jeden Moduls, der Identifier jeder gespeicherten Expression mit den Named Exports und Imports verglichen werden und bei Überschneidungen diese Expression Node als SideEffect melden. Somit werden SideEffects die direkt einen Named Export betreffen erkannt und eine entsprechende Warnung dem Benutzer oder der Benutzerin angezeigt.

Das Erkennen von indirekten SideEffects hingegen gestaltet sich komplizierter. Zuerst werden alle Named Imports wie zum Beispiel `import {counter, incCounter} from 'counter'` ebenfalls in einem Array gespeichert. Der dafür verwendete Node Selector ist `ImportDeclaration`. Für indirekte SideEffects können entstehen wenn Importe mehrer Specifier besitzen. Als Beispiel die Import Specifier aus Listing 29 sind `counter` und `incCounter`, der Import deklariert also zwei Variablen aus dem selben Modul. Wenn nun eine der Expressions sich mit einem der Einträge in `import` übereinstimmt und das Modul einen im selben Import befindlicher Specifier als Named Export exportiert, wird davon ausgegangen das es sich dabei um einen SideEffect handelt. Die entsprechende Expression Node wird somit reported und der Benutzer erhält einen Hinweis, dass dies zu ungewollten Effekten in seinem Programm führen kann.

Indirekte SideEffects wie `incCounter()` aus Listing 29 werden somit zusätzlich zu den direkten SideEffects erkannt und alle Tests werden erfolgreich abgeschlossen.

8.3 Entry Point SideEffects

Tabelle 4: Yeoman Generator: Entry Point SideEffects

Rule-ID:	entry-point-sideeffects
Beschreibung:	todo
Beispiel für fehlerhaften Code:	todo

Die Startdatei oder Entry Point einer JavaScript Library ist ein wichtiger Bestandteil für die Effektivität von Tree Shaking. (einleitungsteil)

```

1 var ruleTester = new RuleTester()
2 ruleTester.run('entry-point-sideeffects', rule, {
3   valid: [
4     {
5       options: [],
6       filename: 'test/usr/src/entry.js',
7       code: `
8         import module from 'module'
9         console.log('sideeffect')

```

```

10     export const export1 = 'export1'
11     export default 'default'
12   },
13   errors: []
14 },
15 {
16   options: [],
17   filename: 'test/usr/src/entry.js',
18   code: `
19     import module from 'module'
20     module.test = 'tmp'
21     export const export1 = 'export1'
22     export default 'default'
23   `,
24   errors: []
25 },
26 {
27   options: [],
28   filename: 'test/usr/src/entry.js',
29   code: `
30     import module from 'module'
31     console.log('test')
32     export const export1 = 'export1'
33     export default 'default'
34   `,
35   errors: []
36 }
37 ],
38
39 invalid: [
40   {
41     options: ['**/src/entry.js'],
42     filename: 'test/usr/src/entry.js',
43     code: `
44       import module from 'module'
45       module.test = 'tmp'
46       export const export1 = 'export1'
47       export default 'default'
48     `,
49     errors: [
50       {
51         message: 'Sideeffects in the entry-point are not allowed.',
52         type: 'ExpressionStatement'
53       }
54     ]
55   },
56   {
57     options: ['**/src/entry.js'],
58     filename: 'test/usr/src/entry.js',
59     code: `
60       import module from 'module'
61       console.log('test')

```

```
62     export const export1 = 'export1'
63     export default 'default'
64   },
65   errors: [
66     {
67       message: 'Sideeffects in the entry-point are not allowed.',
68       type: 'ExpressionStatement'
69     }
70   ]
71 }
72 ]
73 })
```

Listing 32: Entry Point SideEffects Unit Tests

Die Regel soll, jede Node im Entry Point bei der es sich weder um einen Import noch einen Export handelt, für Entwickler und Entwicklerinnen eine Meldung anzeigen. Für die Implementierung werden die Node Types `ImportDeclaration`, `ExportNamedDeclaration`, `ExportDefaultDeclaration` und `Program` benötigt. In weiterer Folge wird zusätzlich der `:not Selector` benötigt. Eine Beschreibung dazu lieferte der ESLint Selector Guide¹¹. Der gesamte Selector um alle möglichen SideEffects in einem Modul zu finden ist: `Program > :not(ImportDeclaration, ExportNamedDeclaration, ExportDefaultDeclaration)`.

Anschließend muss noch überprüft werden, ob es sich bei dem aktuellen Modul, um den Entry Point handelt. Der Benutzer oder die Benutzerin kann dies durch eine Option steuern. Die Entry Point SideEffects Regel erwartet als Parameter einen Glob Pattern String. Dieser dieht dazu den Entry Point festzulegen. Nur dieser soll absolut frei von SideEffects sein und darf lediglich Importe und Exporte Deklarationen beinhalten.

11. <https://eslint.org/docs/developer-guide/selectors> - besucht am 29.06.2018.

Abkürzungsverzeichnis

TCP	Transmission Control Protocol
VR	Virtual Reality

Abbildungsverzeichnis

Listings

1	CommonJS Module	2
2	Asynchronous Model Definition	2
3	Named Exports	3
4	Default Exports	4
5	Import Deklaration	4
6	Export Deklaration	4
7	Flexibele Struktur bei CommonJS	5
8	Macros in Javascript	6
9	Importe sind Views auf exportierte Entitäten	6
10	Importe sind Views auf exportierte Entitäten	7
11	Import Möglichkeiten in ES6	8
12	Export Möglichkeiten in ES6	8
13	helpers.js	8
14	index.js	9
15	bundle.js	9
16	webpack.config.js	10
17	Library mit Side Effects	12
18	Verwendung der Library	13
19	Side Effect mit Auswirkung auf Exporte	13
20	sideeffectful.js mit CSS Side Effect	14
21	styles.css	14
22	generierte package.json Datei	20
23	Empty Imports Unit Tests	22
24	Code zum Aufspüren von Empty Imports	23
25	Empty Import Option Schema	23
26	isKnownSideEffect Funktion	23
27	isKnownSideEffect Funktion	24
28	counter.js	24
29	lib.js	25
30	index.js	25

<i>TABELLENVERZEICHNIS</i>	32
----------------------------	----

31	Named Exports SideEffects Unit Tests	25
32	Entry Point SideEffects Unit Tests	28

Tabellenverzeichnis

1	Local und Export Namen verschiedene Export Varianten	7
2	Yeoman Generator: Empty Imports	21
3	Yeoman Generator: Named Exports SideEffects	24
4	Yeoman Generator: Entry Point SideEffects	28

Literaturverzeichnis

- Babel. 2018. *Babel · The compiler for writing next generation JavaScript*. Besucht am 22. März. <http://babeljs.io/>.
- Bessey, Al, Dawson Engler, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky und Scott McPeak. 2010. »A few billion lines of code later«. *Communications of the ACM* 53, Nr. 2 (Februar): 66–75. ISSN: 00010782. doi:10.1145/1646353.1646374. <http://portal.acm.org/citation.cfm?doid=1646353.1646374>.
- Bush, WR, JD Pincus, DJ Sielaff - Software-Practice And und undefined 2000. o.Dat. »A static analyzer for finding dynamic programming errors«. *web2.cs.columbia.edu*. <http://web2.cs.columbia.edu/%7B~%7Djunfeng/08fa-e6998/sched/readings/prefix.pdf>.
- CanIUse. 2018a. *Browser Usage Table*. Besucht am 22. März. <https://caniuse.com/usage-table>.
- . 2018b. *Support tables for ES6*. Besucht am 22. März. <https://caniuse.com/%7B%5C#%7Dsearch=es6>.
- Docs, Npm. 2018. *What is npm?* Besucht am 11. März. <https://docs.npmjs.com/getting-started/what-is-npm>.
- Dominik Wilkowski. 2018. *npm and the front end*. Besucht am 11. März. <https://medium.com/dailyjs/npm-and-the-front-end-950c79fc22ce>.
- ECMAScript. 2015. *ECMAScript 2015 Language Specification*. Besucht am 22. Januar 2018. <http://www.ecma-international.org/ecma-262/6.0/>.
- ESLintAbout. 2018. *About - ESLint - Pluggable JavaScript linter*. Besucht am 29. März. <https://eslint.org/docs/about/>.
- ESLintNewRules. 2018. *New Rules - ESLint - Pluggable JavaScript linter*. Besucht am 4. April. <https://eslint.org/docs/developer-guide/contributing/new-rules>.
- export - JavaScript* — MDN. 2018. Besucht am 13. März. <https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/export>.
- import - JavaScript* — MDN. 2018. Besucht am 13. März. <https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/import>.
- Johnson, B, Y Song, E Murphy-Hill und R Bowdidge. o.Dat. »Why Don't Software Developers Use Static Analysis Tools to Find Bugs?« *pdfs.semanticscholar.org*. <https://pdfs.semanticscholar.org/060c/c5c0fa6bed9a39bdb6f9c995586f4709006e.pdf>.

- Johnson, SC. 1978. *Lint, a C program checker*. <https://pdfs.semanticscholar.org/7461/7cffa3c6438d04aa99bef1cca415de47d0d3.pdf>.
- Louridas, P. 2006. »Static code analysis«. *IEEE Software* 23:58–61. doi:10.1109/MS.2006.114. <http://ieeexplore.ieee.org/abstract/document/1657940/>.
- McConnell, Steve. 2004. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press. ISBN: 0735619670.
- NodeJSMODULES. 2018. *Modules — Node.js v10.4.1 Documentation*. Besucht am 15. Juni. <https://nodejs.org/api/modules.html>.
- Nolan Lawson. 2018. *The cost of small modules — Read the Tea Leaves*. Besucht am 25. März. <https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/>.
- Rauschmayer, Axel. 2014. *ECMAScript 6 modules: the final syntax*. Besucht am 13. März 2018. <http://2ality.com/2014/09/es6-modules-final.html>.
- . 2018. *Exploring ES6*. Besucht am 25. Juni 2018. <http://exploringjs.com/es6/index.html>.
- Rich Harris. 2018. *Webpack and Rollup: the same but different – webpack – Medium*. Besucht am 25. März. <https://medium.com/webpack/webpack-and-rollup-the-same-but-different-a41ad427058c>.
- RollupJSDocs. 2018. *RollupJS*. Besucht am 12. März. <https://rollupjs.org/guide/en>.
- Sebastian Peyrott. 2018. *JavaScript Module Systems Showdown: CommonJS vs AMD vs ES2015*. Besucht am 12. März. <https://auth0.com/blog/javascript-module-systems-showdown/>.
- Tamás Sallai. 2018. *Why Webpack 2's Tree Shaking is not as effective as you think - Advanced Web Machinery*. Besucht am 25. März. <https://advancedweb.hu/2017/02/07/treeshaking/>.
- Vandevoorde, David, und Nicolai M. Josuttis. 2002. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, November. ISBN: 0201734842.
- WebpackConcepts. 2018. *Concepts*. Besucht am 12. März. <https://webpack.js.org/concepts/>.
- WebpackTreeShaking. 2018. *Webpack - Tree Shaking*. Besucht am 20. März. <https://webpack.js.org/guides/tree-shaking/>.

Appendix

A git-Repository

Daten für Bachelorarbeit 2:

- LaTeX-Code der finalen Version der Arbeit
- alle Publikationen, die als pdf verfügbar sind.
- alle Webseiten als pdf
- Quellcode für praktischen Teil
- Vorlagen für Studienmaterial (Fragebögen, Einverständniserklärung, ...)
- eingescanntes, ausgefülltes Studienmaterial (Fragebögen, Einverständniserklärung, ...)
- Rohdaten und aufbereitete Daten der Evaluierungen (Log-Daten, Tabellen, Graphen, Scripts, ...)

<https://gitlab.mediacube.at/fhs123456/Abschlussarbeiten-Max-Muster>