FH Salzburg
MultiMediaTechnology

Deadcode Detection mittles 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 angegeben 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

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.

Inhaltsverzeichnis

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 Millarden Downloads pro Woche ist es zu einer zentralen Bibliothek von Frameworks, Libaries 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.(**DominikV**

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 Packete 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. (**RollupJSDocs**; **WebpackConcepts**)

(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. (**RichHarris**))

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 Libaries 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. (**SebastianPeyrott**)

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. (**SebastianPeyrott**)

```
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.(**SebastianPeyrott**)

3.2 Asynchronous Model Definition (AMD)

Das Entwickler Team von AMD spaltete sich wärend der Entwicklung von CommonJS ab. Der Haupt Unterschied zwischen diese beiden Systemen liegt in dem asynchronen laden von Ressourcen. (**SebastianPeyrott**)

```
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. Libaries 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. (**SebastianPeyrott**)

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). (**AxelRauschmayer**)

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. (**AxelRauschmayer**)

```
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 3.3.5 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. (**AxelRauschmayer**)

```
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. (**AxelRauschmayer**)

```
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 Pragrammcode ausführen, um herrauszufinden 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. (**AxelRauschmayer**)

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. (**AxelRauschmayer**)

```
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. (**AxelRauschmayer**)

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. (**AxelRauschmayer**)

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.(**AxelRauschmay**

3. Macro Support

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

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

```
import name from "module-name";
import * as name from "module-name";
import { member } from "module-name";
import { member as alias } from "module-name";
import { member1 , member2 } from "module-name";
import { member1 , member2 as alias2 , [...] } from "module-name";
import defaultMember, { member [ , [...] ] } from "module-name";
import defaultMember, * as alias from "module-name";
import defaultMember from "module-name";
import "module-name";
```

Listing 9: Import Möglichkeiten in ES6

```
export { name1, name2, ..., nameN };
export { variable1 as name1, variable2 as name2, ..., nameN };
export let name1, name2, ..., nameN; // oder: var
export let name1 = ..., name2 = ..., ..., nameN; // oder: var, const

export default expression;
export default function (...) { ... } // oder: class, function*
export default function name1(...) { ... } // oder: class, function*
export { name1 as default, ... };

export * from ...;
export { name1, name2, ..., nameN } from ...;
export { import1 as name1, import2 as name2, ..., nameN } from ...;
```

Listing 10: Export Möglichkeiten in ES6

4 Dependency Graph

5 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**)

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 11: helpers.js

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

Listing 12: index.js

Listing 12 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 13: bundle.js

Um die unnötigne Codezeilen zu entfernen werden diese zuerst von Webpack makiert. In Listing 13 wurde, sowohl der Default Export, als auch die `world` Funktion mit `unused harmony` makiert.

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**)

5.1 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 (**CanIUseES6**). Zweitens benutzt ein beachtlicher Teil der Clients nicht die aktuellste verfügbare Version (**CanIUseUsageTable**).

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**)

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**)

```
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 14: webpack.config.js

Listing 14 zeigt eine typische Tree Shaking Konfiguration von Babel und Webpack. Besonders wichtig ist der `modules: false` Flag 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**)

5.2 RollupJS

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

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` (**RichHarris**). Features wie On-demand Loading werden dadurch erst möglich. **NolanLawson** 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. (**RichHarris**)

5.3 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 Libary 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 Libaries verwendet. Wenn diese Tree Shaking unterstützt, können jedoch Bundler wie Webpack, den unnötig Code beim kompilieren der Bundle Datei entfernen.

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

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

5.3.1 Richtiges Importieren und Exportieren

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

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

5.3.2 Exportieren

keine default exporte ?

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

5.3.3 Side Effects

Umfangreiche und weitverbreitete JavaScript Libaries wie zum Beispiel Lodash[1] und jQuery[2] 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 dazuführen 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 Libary Entwickler und Entwicklerinnen diese so zu Implementieren ,dass der nicht benötigte Libary Sourcecode beim builden entfernt wird. Durch die Verwendung von `sideEffects: false` in der package.json Datei innerhalb der Libary, wird Webpack mitgeteilt das Tree Shaking nicht nur auf den üblichen Projektcode anzuwenden, sondern ebenfalls auf den Code der sich im Libary Ordner befindet.

1. https://lodash.com/ - besucht am 26.05.2018
2. https://jquery.com/ - besucht am 26.05.2018

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 Libary Code entfernt (**WebpackTreeShaking**).Zur Veranschaulichung dient ein ausführlich dokumentiertes Beispiel[3].

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. Side Effects sind somit Code Segmente, welche beim Tree Shaking nicht entfernt werden dürfen. Dies möchte ich mittels eines Beispiels näher erläutern.

```
1  import a from 'a'
2    import b from 'b'
3
4    console.log("das ist ein sideeffect")
5
6    export default {
7      a,
8        b
9    }
```

Listing 15: webpack.config.js

————-BSP von webpack————————

-vorallem wichtig für libary entwickler -was sind sideeffects? -wie schaut eine übliche sideeffect freie libary aus? -Aus den ES Specs wurden folgende AST Nodes ermittelt (ImportDeclaration, ExportNamedDeclaration, ExportDefaultDeclaration) -des weiteren muss jeder import auch exportiert werden -kann auch ein array von files angegeben werden (diese besitzen sideeffects)

(**TamasSallai**)

6 Linting

Der Begriff Lint wurde erstmals in den 1970er in Verbindung mit der Softwareentwicklung erwähnt. **Johnson1978** entwickelte es an den Bell Laboratories um die Schwächen der damaligen C Compiler auszugleichen und undendteckte Fehlerin 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 **Louridas2006** 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.

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

Die meisten Fehler fallen in die Kategorie Known Errors. Dies sind häufig auftretende sich wiederholende Fälle in dennen die Entwickler und Entwicklerinnen immer wieder hinnenstoplern. Mit Linting möchte man die sich ständig wiederholenden Fehler oder auch Code Smells so früh es möglich ist finden und ausbessern. (**Louridas2006**)

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. (**Louridas2006**)

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

Es exisitiert bereits reichlich Literatur zur Statischen Code Analyse. Dabei wurden nicht nur Aspekte wie Performance und Genauigkeit untersucht (**Bessey2010**), sondern auch verschiedene Einsatzgebiete gezeigt (**Bush**).

Johnson erforschten in ihrer Arbeit den Gebrauch und die Verbreitung von Statische 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 postives, 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** sollten zukünftige Tools, sowohl eine besser Integration in den Arbeitsabläufen von Entwicklern und Entwicklerinnen bieten, als auch die Arbeit in einem Team besser unterstüzen.

Bush implementierte ein statisches Analyse Tool zum auffinden von dynamischen Fehler in C und C++. Dazu zählen unter anderem die Verwendung von nicht intitialisierten Speicher oder falschen Operation auf Dateien (zum Beispiel das Schließen einer Datei die bereit 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 Topolischen Algorithmus sortiert. Die Funktionen werden daraufhin simuliert und darin befindliche Defekte gemeldet. Als Teil dieser Arbeit wird ein Plugin erstellt, welches wie bei **Bush** 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.

6.1 ES Lint

2013 erschuff 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**)

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**)

(Kurzer Einleitungstext für die nächsten Kapitel)

6.1.1 Core Rules

ESLint wird bereits mit mehr als 200 allgemein gültige Regeln ausgeliefert. Diese werden als Core Regeln bezeichnet. Gemäß (**ESLintNewRules**) 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 Poblem 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 ünd"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. Abhängikeiten zwischen zwei oder mehreren Regeln sind nicht erlaubt.

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

5. Unabhängig Regeln dürfen nur auf der JavaScript Laufzeitumgebung basieren und unabhängig von Libaries oder Frameworks sein. Core Regeln sollten stehts anwendbar sein und nicht davon abhängen ob spezielle Libaries 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 eioner Entwicklerin verlang 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)

6.1.2 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**

6.1.3 How to use

Client config verwendung

6.1.4 Formatters

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

6.2 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**)

——todo——

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

7 Untersuchung von JavaScript Libaries

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.

7.1 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 5.3.3 veranschaulicht diehnt 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 [4] die diese Module beinhaltet.

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

7.2 Ausgewählte Libaries

Anhand Auswahlkriterien wurde folgende fünf Projekte ermittelt die bereits Tree Shaking und ES6 implemnentieren:

- lodash[5]
- rxjs[6]
- vue[7]
- graphql[8]

8 Erstellen des Plugins

8.1 Setup

Um die Projektstruktur aufzusetzen verwende ich die von **ESLintNewRules** empfohlene Vorgehensweise. Ein wichtige Voraussetzung um ein ES Lint PLugin zu erstellen sind NPM und

4. https://www.npmjs.com/browse/depended - besucht am 06.06.2018
5. https://github.com/lodash/lodash - besucht am 15.06.2018
6. https://github.com/ReactiveX/rxjs - besucht am 15.06.2018
7. https://github.com/vuejs/vue - besucht am 15.06.2018
8. https://github.com/graphql/graphql-js - besucht am 15.06.2018

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:

```
{
  "name": "eslint-plugin-treeshaking",
  "version": "0.0.0",
  "description": "test",
  "keywords": [
    "eslint",
    "eslintplugin",
    "eslint-plugin"
  ],
  "author": "silltho",
  "main": "lib/index.js",
  "scripts": {
    "test": "mocha tests --recursive"
  },
  "dependencies": {
    "requireindex": "~1.1.0"
  },
  "devDependencies": {
    "eslint": "~3.9.1",
    "mocha": "^3.1.2"
  },
  "engines": {
    "node": ">=0.10.0"
  },
  "license": "ISC"
}
```

Listing 16: 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.

8.2 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

9 Die Implementierung neuer Regeln

Yeoman und der ESLint-Generator bietet ebenfalls Unterstützung 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 5.3 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 6.1.1 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.

9.1 Sideeffects

Tabelle 1: Yeoman Generator: Sideeffects

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

-modulebody besteht aus mehreren Moduleitems -¿ ImportDeclaration, ExportDeclaration, StatementListItem

10 Verwendung bestehender Core Regeln

Abkürzungsverzeichnis

TCP	Transmission Control Protocol
VR	Virtual Reality

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Anhänge löschen, die nicht verwendet werden.

Appendix

A git-Repository

Das Repository dient zur Dokumentation und Nachvollziehbarkeit der Arbeitsschritte. Stellen Sie sicher, dass der/die BetreuerIn Zugriff auf das Repository hat. Stellen im Sinne des Datenschutzes sicher, dass das Repository nicht für andere zugänglich ist.

Daten für Bachelorarbeit 1 und 2:

- LaTeX-Code der finalen Version der Arbeit
- alle Publikationen, die als pdf verfügbar sind.
- alle Webseiten als pdf

Daten für Bachelorarbeit 2:

- 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, ...)

Link zum Repository auf dem MMT-git-Server `gitlab.mediacube.at`:

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

B Vorlagen für Studienmaterial

Vorlagen für Studienmaterial müssen in den Anhang.

C Archivierte Webseiten

`http://web.archive.org/web/20160526143921/http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entit` letzter Zugriff 1.1.2016

`http://web.archive.org/web/20160526144551/http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides_with_notes.pdf`, letzter Zugriff 1.1.2016