Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Arthur dos Santos Dias

Application of Design Patterns for modern JavaScript to solve real world problems

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Arthur dos Santos Dias

Application of Design Patterns for modern JavaScript to solve real world problems

Bachelorthesisbased on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Aplied Sciences

Supervising examiner : Prof. Dr. Klaus Jünemann
Second Examiner : Prof. Dr. Robert Heß

Day of delivery 5. Oktober 2017

Arthur dos Santos Dias

Title of the Bachelorthesis

Application of Design Patterns for modern JavaScript to solve real world problems

Keywords

JavaScript, Programming, Design, Patterns, Design, Software, Web, ES6, ES7

Abstract

The aim of this Bachelor Thesis is to verify the usability of Design Patterns applied to Modern JavaScript in order to solve Common Problems in the industry. First an overview of the current state of JavaScript is presented. Then a closer look on Design Patterns is made, with focus on specific design patterns which can be optimized with modern JavaScript. After that a generator for design patterns is proposed and analyzed. Finally a few tests with real world scenarios are made using the generator, and a conclusion by measuring the benefits of using modern JavaScript to apply Design Patterns versus the classical methods.

Arthur dos Santos Dias

Titel der Arbeit

Entwurfsmuster für modernes Javascript zur Lösung von realen Anwendungsproblemen

Stichworte

JavaScript, Programmierung, Design, Design, Software, Web, ES6, ES7

Kurzzusammenfassung

Das Ziel dieser Bachelor-Arbeit ist es, die Verwendbarkeit von Entwurfsmustern für modernes JavaScript zu überprüfen, um typische Probleme in der angewandten Softwareentwicklung zu lösen. Zuerst wird ein Überblick über den aktuellen Stand von JavaScript gegeben. Dann wird ein genauerer Blick auf Entwurfsmuster in Javascript geworfen. Der Fokus liegt dabei auf denjenigen Entwurfsmustern, die mit modernem JavaScript optimiert werden können. Danach wird ein Generator für Entwurfsmuster entwickelt. Schließlich werden einige Tests mit realen Anwendungsszenarien mit dem Generator durchgeführt. Den Abschluss bildet ein Vergleich, in dem die Vorteile der Verwendung von modernem JavaScript mit Entwurfsmustern gegenüber herkömmlichen Methoden aufgezeigt werden.

Contents

List of Figures 6

1 Introduction 7
- 1.1 Goal . . . 7

2 Background 8
- 2.1 JavaScript . . . 8
- 2.2 Design Patterns . . . 9

3 Design Patterns in JavaScript 11
- 3.1 Iterator . . . 11
- 3.2 Decorator . . . 12
- 3.3 Singleton . . . 13
- 3.4 Observer . . . 14

4 Code Generator 15
- 4.1 Requirements . . . 16
- 4.2 Design . . . 17
- 4.3 Implementation . . . 19
  - 4.3.1 Index . . . 19
  - 4.3.2 Singleton . . . 19
  - 4.3.3 Iterator . . . 20
  - 4.3.4 Decorator . . . 20
  - 4.3.5 Observer . . . 21
- 4.4 Test . . . 22

5 Review 25
- 5.1 A Real World Problem Scenario . . . 26
- 5.2 Implementation . . . 27
  - 5.2.1 Modern JavaScript Approach . . . 27
  - 5.2.2 Standard design patterns approach . . . 31
- 5.3 Pattern Conclusion . . . 36
  - 5.3.1 Singleton . . . 36

5.3.2 Iterator 36
5.3.3 Decorator 37
5.3.4 Observer 38
5.3.5 Considerations 38

6 Conclusion **39**

Bibliography **40**

List of Figures

3.1 The iterator design pattern structure 12
3.2 The Decorator design pattern structure 13
3.3 The Singleton design pattern structure 13
3.4 The Observer design pattern structure 14

4.1 Class Diagram of the Code Generator 18

5.1 Comparison between loop methods on classical and modern Iterator approach 37

1 Introduction

The Software industry has changed a lot since the last years. Systems depend more and more on cloud solutions, asynchronous programming and interconnectivity in order to operate as required. At the same time, Software programming has changed as well to adapt to new challenges that were introduced, either by new scientific findings or new technologies that make usual tasks better.

At the same time, fundamental practices are still a reality. Design Patterns, for example, which are implemented by almost every language today, are still the best way to obtain a stable, reusable and safe system.

Although Design Patterns were created long ago, they are useful to build a software architecture that solves problems even from today. How exactly they can contribute to a software project with new technologies and languages that are constantly evolving, like JavaScript, is the key point. The focus of this Bachelor Thesis is to explore the new features of JavaScript inside a few design patterns, and verify whether these can be structured and implemented in a different way.

In order to achieve this, a generator for Design Patterns will be introduced. It aims to help applying a Design Pattern with the language's advantages directly into our problem avoiding writing too much base code, and by consequence saving time. The generator will be used in a real world example project with a few requirements that should be fulfilled with the help of the generated design patterns. The results will be analyzed in order to determine the benefits of applying Design Patterns in modern JavaScript, allowing us to conclude if Design Patterns can be improved for modern languages as they are defined in literature.

1.1 Goal

This thesis contains two main goals. One is to demonstrate how different design patterns can be applied compared to their original implementations, and what are the benefits of using Design Patterns in modern programming languages compared to the standard approach defined by literature. Another goal is to propose a design patterns generator for modern design pattern implementation, including its design, implementation and testing.

2 Background

2.1 JavaScript

JavaScript is a Object-oriented scripting language originated in the middle of 90's by Sun Microsystems. It was created as an intent to improve the usability of web browsers, which by the time could not do much more than display markup content. Its first release 1.0 ran on the most popular navigator on that time, Netscape Navigator 2. With time JavaScript evolved and gained its official standard, originating ECMAScript. 1998 followed a version 2 of ECMAScript with a few improvements, and one year later ECMAScript 3, which is the base of the language that is used today. Interesting improvements have been made in 2009 with the introduction of ECMAScript 5, although it was a hard migration due to the requirement of browser support. In 2015 the official committee TC39, which organizes ECMAScript releases, decided to move to an annual release approach, which caused ECMAScript 6 to be most commonly named ECMAScript 2015, or ES2015.

The main features added to ES2015 include

- *strict mode*: Introduces several changes to JavaScript semantics, exposing errors which were otherwise silently thrown, among other features;
- Several Array methods, such as *indexOf, filter, map*, and several others;
- JSON support, enabling parse or encode of values into / from a JSON format;
- The exponential operator **.

Some of the main features of the 2016 version of ECMAScript include:

- Arrow function syntax
- Classes, which is basically a simpler way to work on the prototype-based inheritance;
- Template strings, which allows one to embed expressions without concatenation;
- Block scoped constructs *let* and *const*, which improves the way block scope works inside code;
- The addition of modules

- *Maps* and *Sets* as data structures.

The next specification of ECMAScript, ES2017, has 2 major new features:

- *async* Functions: Enables a new way of writing asynchronous code based on generators, giving a more synchronous appearance to the operation
- *Shared memory* and *atomics*: An improvement on Parallelism capabilities of javaScript.

Although it has been created in only 10 days(according to Timms (2014)), JavaScript is a very complex language and can be used to solve most kind of problems. It's flexibility allows one to learn its basics without much effort, while at the same time it has evolved into a language which enables one to create very complex applications, be it in the client or the server.

In the past years software development has changed to adapt to be more modular and component-based. With the advanced of concepts, like cloud computing or Internet of Things(IoT), it is necessary to have a system which is enable to communicate and change its shape to adapt to different components around it. For that reason JavaScript was considered one of the 5 languages which employers look for the most (from Diakopoulos und Cass (2017)): it is a language which is constantly evolving as the world around it keeps changing.

2.2 Design Patterns

Design Patterns originates in the late 70's as an idea of an architecture, which turned later on to a more concrete realization in the late 90's. Since then, most of the programming languages haven been provided with implementations of design patterns. They are most commonly defined on Object Oriented (OO) Languages, such as Java or C++, but can be also identified on Procedural languages like C.

A good knowledge in design patterns can bring several benefits to a project, such as

- Reusability
- Maintainability
- Scalability

and several others.

Despite all the advantages, in many cases a software is built under no pattern at all. Although there is no official statistic on the topic, it is safe to say the chances that a product is developed without following the guidelines of a Design Pattern is big. It results on what literature defines as a "Big ball of Mud" (Foote und Yoder (1999)) or "Spaghetti Code (Dijkstra (1968))", terms used to describe a poor maintainable and bad written code. This is usually a case on the modern business, where pressure for faster results is getting higher and there is less time to think about the code, and also due to offer of jobs in IT area exceeding the amount of qualified and experienced professionals, which forces companies to sometimes lower their expectations in order to build their team.

In the other hand, when it comes to Design Patterns in fact being used, the choice of language seems to influence on how often a design pattern is chosen. Systems are connected on the Web usually require HTTP transactions for every single operations, which requires the Observer Pattern to be strongly present. Almost every system requires certain information to be unique and single-instanced, making it Singleton the pattern of choice.

According to (Gamma u. a. (1994)), one design pattern can be described as a composition of 4 basic elements:

- A **Name**, which should be simple and self-explanatory, not longer than a word or two;
- It should be related to a certain **Problem**, which is sometimes a list of requirements to fulfill certain objective, or a specific design problem that must be targeted;
- The **Solution** consists of a certain strategy that can be used to solve that problem; it is not a concrete solution that one can just copy from a source, but more like an idea of an architecture and its parts.
- **Consequences** are the outcome of applying the design pattern, and tells how is the overall look of the application after that approach has been taken, justifying why is the initial problem solved, and also warning of possible side effects or caveats of that design pattern. The effort of applying the pattern is also exposed to help decision.

Design Patterns capture solutions that have developed and evolved over time. Hence they aren't the designs people tend to generate initially (from Gamma u. a. (1994)). For this reason, it is important to analyze wether a programming language can somehow optimize the application of design patterns, or if it can use special features to make the classical procedures faster or with less lines of code. This has been proven for several languages, like C++ where several new features were introduced in 2014 with C++14 (Schwensen (2016)). Most important, the solution and consequences after applying a design pattern with a different approach must be taken into account. Otherwise it would be advisable to establish a new concept of design pattern.

3 Design Patterns in JavaScript

There are more than 20 famous design patterns in the literature, and it would not be possible to discuss all of them in the scope if this thesis. the focus will be given on design patterns which can suffer bigger changes with the new features of JavaScript. The main parts of these design patterns will be reviewed.

3.1 Iterator

The main purpose of the Iterator Pattern is to provide a way to access the members of a collection without the need of know the representation of those members. This pattern is also known in literature with the name or Cursor. An Iterator should provide basic features to access a group of elements, such as:

- *first()*: positions the pointer to the beginning of the collection;
- *next()*: moves the pointer 1 position in the collection;
- *end*: returns whether the end of the collection has been reached;
- *item*: returns the member which is located at the current pointer position;
- *index*: returns the index of the member which is located at the current pointer position.

There are mainly four actors in the classical Iterator Pattern:

- *Iterator*: An interface which provides methods for accessing elements;
- *ConcreteIterator*: Implements the Iterator interface;
- *Aggregate*: Interface for creating an Iterator
- *ConcreteAggregate*: handles the creation of an Iterator with the specified data structure.

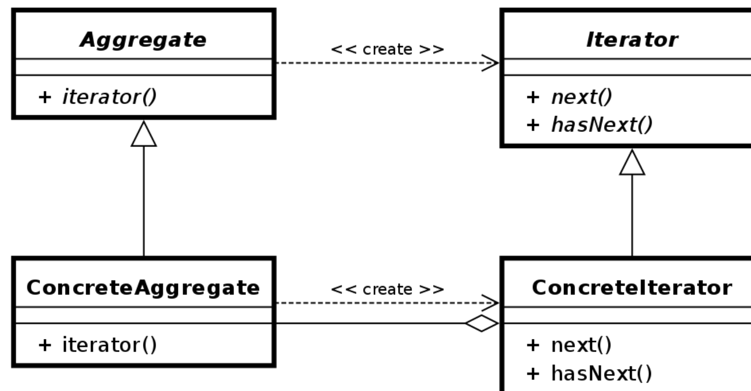

Figure 3.1: The iterator design pattern structure

ES5 release of JavaScript provides the *Symbol.Iterator* symbol, which is used to iterate over a collection. This can be used to add iterator functionalities to an object, plus describing how one would like to iterate over it (for example, backwards instead of normally going forward in a loop).

3.2 Decorator

the Decorator pattern can be used to dynamically add functionality to an Object, without changing its original values.

The classical members of a Decorator Pattern are

- Component: Defines an interface of objects to be decorated;
- ConcreteComponent: implements the interface and provides more information;
- Decorator: References the component, defines an interface that fulfills the Component's interface;
- ConcreteDecorator: implements additional features to the Component.

JavaScript has a simple way to implement Decorators via the Decorator annotation, a new feature currently in stage 2 of ES2017. This means it is not officially part of ECMAScript, but in development process. Our project will be able to use experimental features, in order to showcase the advantages of this approach.

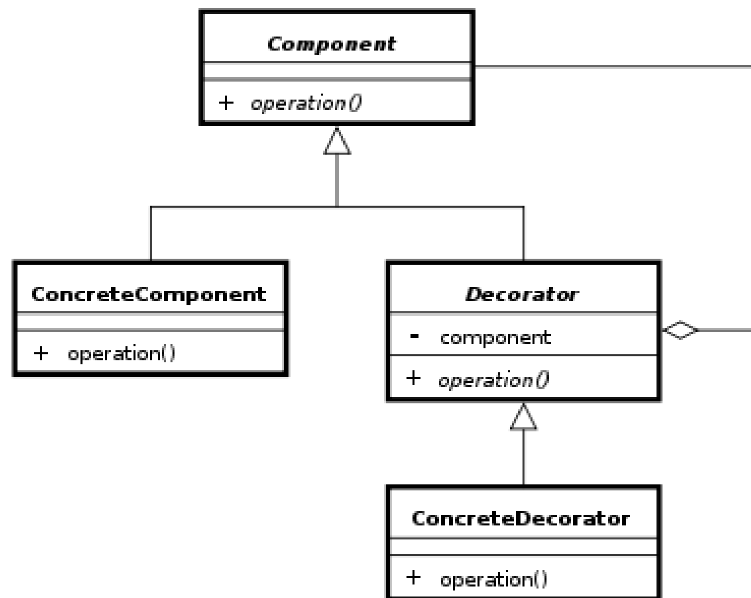

Figure 3.2: The Decorator design pattern structure

3.3 Singleton

The Singleton pattern is required where only one instance of a class should exist. There are no specific actors in a Singleton, except by the Singleton class itself. An implementation of a singleton consists of verifying that an instance of a class should be reused if it already exists.

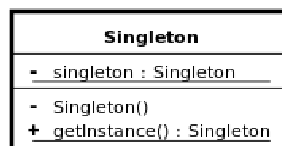

Figure 3.3: The Singleton design pattern structure

Singleton pattern is one of the most common and most widely used design patterns. Even though there might not be a direct advantage of implementing a singleton in JavaScript, it is worth showing that this pattern is also easily doable with the language.

3.4 Observer

The Observer pattern focuses on the problem of one-to-many communication: A system should be able to establish a communication between components, at the same time as it does not create a tight connection between them in order to preserve reusability. The most common participants of a Observer Pattern are:

- Subject: An Observable containing three main responsibilities: add observers, remove observers, and notify observers.
- Observer: An observer is what is look at and from where changes are expected from. It provides methods to notify observers.
- Concrete Observer: implements its *notify* method.

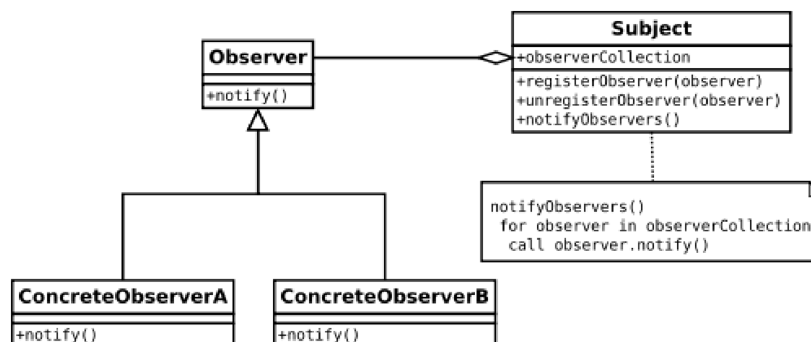

Figure 3.4: The Observer design pattern structure

The implementation of the Observer Pattern is done by using Rx.js, available as a Node.js module for Reactive programming in JavaScript. There is also work in progress to make Reactive Programming capabilities natively available in ECMAScript, but so far it has not been enabled in ES2017. As for now RX.js will be used for this matter.

4 Code Generator

Due to the growing community around javaScript and its flexibility to be used in several different contexts, many tools have been developed to improve the way the code is written. One good example is scaffolding tools.

Scaffolding enables the programmer to specify how the codebase should be used, and makes it possible to generate projects with specific template automatically. This not only saves time, but also avoid errors which happen when one has to manually type a lot of boiler-plate code. There are scaffolding frameworks for several programming languages, including JavaScript.

In order to better analyze Design Patterns implemented with Modern JavaScript, a code generator will be implemented. It will be wrapped around a scaffolding framework, providing the necessary methods to setup design patterns in a project.

4.1 Requirements

In order to operate according to what is expected, a few requirements can be said about the generator.

- Ask the user which design pattern should be generated;
- Ask the user some sort of identification for the class which will represent the design pattern implementation;
- Generate the correct template for design patterns according to user input;
- It should be possible to modify the generated template completely;
- The generated code should be able to represent the main components of the design patterns proposed;
- The generated templates should be written in JavaScript;
- The generated templates should have access to ES6 / ES7 features of JavaScript, which will later on be compiled to browser-readable code,
- The generator should contain assertion tests in order to verify that the template generation process works as expected.

The requirements above should make sure the generator can be fully used and will output the expected information.

4.2 Design

The design of the will mostly depend on which tool will be used. There are two main scaffolding tools available: Yeoman (Yeoman (b)) and Slush (Carlstein). Both tools are designed for the same purpose, which is providing base code for projects. Slush has the difference of using Gulp in the background, while Yeoman uses its own functionality to operate. Slush came only recently and has clearly fewer support support (hosting 452 open sourced generators by the time of this writing, against more than 6500 on Yeoman), and it even contains adapters to enable the usage with the Yeoman syntax.

Based on the requirements from previous section, and additionally the fact that there is a much bigger knowledge base and community effort invested, the scaffolding framework chosen for this work is Yeoman. Yeoman has, among several other features, a generator plugin which allows one to customize a project in matter, by providing the possible templates and prompt functions, enabling several possibilities, for example, asking the user which templates are going to be used before generating the final project. There are already thousands of open-source generators (available on Yeoman (a)), that can be installed in a project or used standalone.

At the time of this work, there are no published Yeoman generators for design patterns using ES6/ES7 JavaScript, which is the motivation to create one. Yeoman offers an open source library to create generators called *yeoman-generator*. We will use it in order to setup our design patterns generator.

The generator for design patterns is implemented as a Node.js package. Node.js is a JavaScript runtime environment platform for JavaScript applications. It provides several tools and plugins which can handle several tasks for the user. Node.js is the actor which is handling all packaging and optimization of the project, such as compilation, minifying, automated testing, lint, mapping and much more. It can be used to create modules which can be imported in another application. Based on that, the design patterns generator will be designed as an exportable module.

The basic generator process consists of:

- A set of questions which are prompted;
- Collection and verification of results;
- Creation of templates with classes implementing a certain functionality.

In order to operate, the generator extends its Base class defined in the *yeoman-generator* package. A new class Generator is created by extending the original class inheriting its main class. An object is passed on the extend call with the prototype functions of the base

class. It is then possible to define the prompt and write phase. The prompting phase is defined in the function *prompting()*, while the file writing is done in the *writing()* function. The prompt phase defines an array *genericPrompts* of question objects to be asked to the user and calls *prompt()* on this array. The write phase is basically a wrapper around a file system (*fs*) library. The generator can be adjusted (reduced or increased) by configuring its *genericPrompts* array. The writing phase is scalable for N number of questions. It is also possible to create a sub-array of questions on *prompting()*, which are conditionally sent to the user based on the initial prompts. The Templates are stored in the Project which the generator is installed. After they are generated, the corresponding classes can be imported in the user's Project due to exportable modules from modern JavaScript.

A description of the classes which compose the code generator can be visualized below.

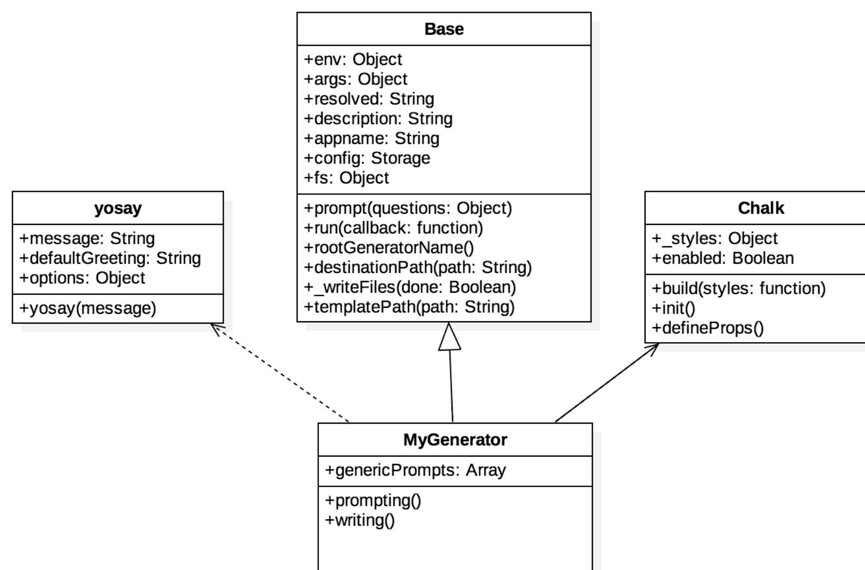

Figure 4.1: Class Diagram of the Code Generator

In the example above, the Design Patterns generator (MyGenerator) inherits from a Base class from Yeoman, and additionally has 2 dependencies: *yosay*, which is the base class for System Output to the user, and Chalk, which provides additional UI elements such as terminal coloring.

4.3 Implementation

Just like most of Node.js modules, the generator project will consist of an entry *index.js* file, which will have the prompt and decision code, and template files for each design pattern that can be generated.

4.3.1 Index

The structure of the index file should be as follows:

- A Welcome message explaining the user what the generator is about;
- Collecting general information: which design Pattern should be created and name of the class which represents it.
- The callback function which analyses the user input and sends the corresponding answers as variables to the templates.

A full output of the index file can be found in the appendix.

4.3.2 Singleton

The Singleton template is composed of a simple class with a constructor which is returning the same instance, if not created yet. The user is able to add his own methods in the class, but it should not be mandatory. The following code shows the singleton class.

```
1  // create an empty instance global to this class
2  let instance = null;
3
4  // the user answer to name of the singleton will be added in serviceName
5  export class <%= serviceName %> {
6
7    constructor() {
8    // if it is the first time this class is accessed, create the single
       instance
9      if (!instance) {
10       instance = this;
11     }
12     return instance;
13   }
14 }
```

4.3.3 Iterator

The iterator approach by ES6 has the main objective of adding iterator properties to a structure. It can be used to convert a complex object into something that can be iterated over. This is mainly done by adding a Symbol which gives iterator properties.

```
1
2  // add Symbol.iterator to the prototype
3  this[Symbol.iterator] = function() {
4    this.count = 0;
5    this.isDone = false;
6    this.properties = Object.keys(this.myIteratorObj);
7
8    // 'next' is called every time there is a loop in a iteration block
9    let next = () => {
10     if (this.count >= this.properties.length) {
11       this.isDone = true;
12     }
13     return { done: this.isDone, value: this.myIteratorObj[this.
          properties[this.count++]] };
14   }
15   return { next };
16 };
```

In order for an object to be iterator (so you can give it to a for loop), some control attributes are necessary: count is a pointer to current position; isDone is a flag to indicate the loop is finished; properties is a abstract representation of the data which is going to be iterated. In this case it is aimed to iterate over the keys of our array of cities. The function next is called every time the for loop proceeds, to internally control the collection.

4.3.4 Decorator

The Decorator template itself contains one single function block which will be filled according to its desired purpose.

```
1 export function <%= serviceName.toLowerCase() %> (target) {}
```

Additionally, it is necessary to import and mention this decorator in the target class by using the ES6 annotation @<decorator name>, but this step is left to be done in the project since our generator does not know when the decorator will be indeed used. The target parameter contains the class which the decorator is targeting, which allows us to modify its prototype.

4.3.5 Observer

The Observer template contains a definition of the StateManager and Observer classes. We can create observers which carry the ID of a interested component. The basic structure of the Observer is as follows:

```
1 export class NotificationServiceObserver {
2   id;
3   constructor(id) {
4     this.id = id;
5   }
6 }
```

Additionally a StateManager class is presented, to manage all observers and trigger their notifications:

```
1 export class StateManager {
2   subject;
3   observables;
4   constructor() {
5     this.observables = [];
6     this.subject = new Rx.Subject();
7   }
8 
9   // adds a new observer.
10  addListener(observer, cb) {
11    this.subject.subscribe({
12      next: (v) => {
13        cb;
14      }
15    });
16  };
17 
18  // deletes an  new observer.
19  removeListener(observer) {
20    observer.unsubscribe();
21  }
22 
23  notifyObservers() {
24    this.subject.next();
25  }
26 }
```

4.4 Test

In order to assert that the generator creates files correctly, one test case for each design pattern template creation process will be added. Our test should assert that the corresponding design pattern file is created and with the correct naming. In order to achieve this, the automatically generated *test* folder can be used. This folder should contain *spec* files, each representing a test case. A spec file is a javascript file which uses test functions for assertions. After installing yeoman, all the test libraries needed for this test purpose are included, providing full access to the list prompt answers and input prompt answers. This information is necessary to check whether a file destination exists based on which type of design pattern and which service name was chosen.

The contents of the test file is as follows:

```
1  'use strict';
2  var path = require('path');
3  var assert = require('yeoman-assert');
4  var helpers = require('yeoman-test');
5
6  // test file description
7  describe('decorator generation test', function () {
8  // function that is run before the test begins
9    before(function () {
10     return helpers.run(path.join(__dirname, '../generators/app'))
11       .withPrompts({serviceChosen: 'Decorator'})
12       .withPrompts({serviceName: 'testFile'})
13       .toPromise();
14   });
15
16   // test begin
17   it('creates files', function () {
18     assert.file([
19       'testFile.decorator.js'
20     ]);
21   });
22 });
```

Three packages from node modules are necessary to be able to run commands and select user prompt options: helpers package to be able to execute commands; path package cares about file path utilities; assert lib contains the assertion functions to compare file name. The before call makes sure that the prompting phase happens at first, only then it is safe to look for a file destination with the same name and type. The other test files are similar to this, so they will be skipped. The output of a test is as follows:

```
1  >> npm test
2
```

```
3  > generator-tsdp@0.0.5 test <USER_DIR>/generator-tsdp
4  > gulp
5
6  [20:32:15] Using gulpfile <USER_DIR>/gulpfile.js
7  [20:32:15] Starting 'static'...
8  [20:32:15] Starting 'test'...
9
10
11   decorator generation test
12     v creates files
13
14   Iterator generation test
15     v creates files
16
17   Observer generation test
18     v creates files
19
20   singleton generation test
21     v creates files
22
23
24   4 passing (797ms)
25
26 ----------|----------|----------|----------|----------|----------------|
27 File      |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
28 ----------|----------|----------|----------|----------|----------------|
29 ----------|----------|----------|----------|----------|----------------|
30 All files |      100 |      100 |      100 |      100 |                |
31 ----------|----------|----------|----------|----------|----------------|
```

The command *npm test* runs all the test files under test folder. After they are run, a report of the coverage of files is generated. This report is installed together from Yeoman and it is very useful when testing the line and branch coverage of a project.

Having a positive flag on tests, it is safe to run the generator and verify its results. A generator made via Yeoman can be activated with the command 'yo <name of your generator project>'.

```
1  >> yo tsdp
2
3  ? Choose theDesign Pattern you wish to implement: (Use arrow keys)
4  > Singleton
5    Iterator
6    Decorator
7    Observer
8
9
10 ? Choose theDesign Pattern you wish to implement: Singleton
11 ? Name of your Class: Unique
```

```
12 ? Name of your Class: Unique
13    create unique.singleton.js
```

The template is always created relative to the current path, so the file *unique.singleton.js* is located at the same level as where the command was given.

```
1  /*
2  * unique.singleton.js
3  * base template for Singleton Design Pattern
4  * author: Arthur Dias
5  */
6  let instance = null;
7
8  export class Unique {
9
10     constructor() {
11         if (!instance) {
12             instance = this;
13         }
14         return instance;
15     }
16 }
```

5 Review

In order to review and measure our design pattern implementation, it is necessary to setup a problem that design patterns can solve. Since the objective of this work is to verify the applicability of design patterns in JavaScript in real world problems, a situation is proposed in which our generated code will be applied.

In order to validate wether our implementation of Design Patterns work well in modern JavaScript, some quality criteria is set. Based on that a measurement of our solutions based on this criteria is possible, resulting in a final judgement whether it makes sense to try an optimization of the pattern.

- Safeness: Do the new approaches of implementation provide more security and less chances of errors?
- Simplification: What is the effort to implement the optimized version of the design pattern?
- Ease of writing: Do the approaches somehow make it quicker and more direct to use design patterns?
- Performance: Can one measure the time necessary to calculate the tasks described in the requirements?

Furthermore, in order to fulfill our objectives and analyze the impacts of modern JavaScript, it is important to provide a traditional version of the discussed Design Patterns applied in the same context. The classical approach, although also written in JavaScript, will not make use of the advanced features which were used in the modern approach (ex: imports, classes, annotations, third-party libraries like Rxjs), but rather following the definitions from literature as close as possible.

5.1 A Real World Problem Scenario

Consider a system which provides information about cities. it should respect the following requirements:

- Criteria 1: Any modification on the cities data should notify all interested parties.
- Criteria 2: There should be at least 2 forms to iterate over the city data.
- Criteria 3: The cities data should be represented as a unique instance e.g there should not be 2 lists of cities.
- Criteria 4: All new user added should be provided of a date stamp containing its creation. Cities data should also contain creation stamp.

Each of the items above can be addressed with a Design Pattern. Criteria 1 involves sharing information which can be done with an Observer. Criteria 2 is achievable via Iterator, criteria 3 can be solved with Singleton and criteria 4 will be handled via a Decorator.

The problem described above is not based on a concrete existing problem, thus it is only an example based on typical challenges that are faced on real world scenarios.

5.2 Implementation

The first implementation shown is a result of applying design patterns with modern JavaScript. The code for the patterns is initially generated by the code generator. The project files are stored in a folder called *modern*, which contains the proposed versions of Design Patterns and also other files as necessary. The classical implementation of design patterns will be stored in the *classic* folder. According to literature, it is a good practice to define an abstract class which composes the interface of the design pattern behavior. Since the classical implementation does not consider the use of classes in JavaScript, we will omit all abstraction of classes and implement the methods directly in the concrete members.

To represent the cities data, an existing resource from Haim will be used. It is not modified and it serves as an example of a kind of data structure to be handled in a real world scenario.

5.2.1 Modern JavaScript Approach

Singleton

The list of cities will be represented as a property of a class 'CityService', which should provide the necessary methods for accessing the cities:

```
1  ...
2  let instance = null;
3  export class CityService {
4    path;
5    id;
6    cityData;
7    constructor() {
8      this.id = this.created;
9      if (!instance) {
10       this.cityData = citiesData;
11       instance = this;
12     }
13     return instance;
14   }
15   ...
```

We keep a class variable *instance* as part of the class, so it is possible to return that same instance from different objects from the class. Although multiple objects of this class can be created, they will always share the same instance.

Iterator

The City iterator class contains an implementation of iterator functionality to our city listing. Since the data comes in a very specific JSON structure, some adjustments are done in the generated code:

```
1  export class RegularIterator {
2  ...
3    this[Symbol.iterator] = function() {
4    this.count = 0;
5    this.count2 = 0;
6    this.isDone = false;
7    this.doneCountry = false;
8    this.properties = Object.keys(this.myIteratorObj);
9    this.countriesLength = this.properties.length;
10
11   let next = () => {
12     if(this.myIteratorObj[this.properties[this.count]].length <= this.
           count2) {
13       this.doneCountry = true;
14       this.count2 = 0;
15
16       if (this.count == this.countriesLength - 1 ) {
17         this.isDone = true;
18       } else {
19         this.count++;
20       }
21     } else {
22       this.doneCountry = false;
23     }
24     return !this.isDone? { done: false,
25       value: this.myIteratorObj[this.properties[this.count]][this.count2
            ++]
26     } : {done: true};
27   }
28   return { next };
29 };
30   }
31   ...
```

Our City Data comes as a complex key-value pair, where keys are represented by Countries, and a value is represented by an array of cities corresponding to that country. In order to iterate only in the cities, the control flags are adjusted such that it loops through the arrays as it loops through cities in an external manner. The class can now be instantiated in the index file passing the data as argument.

```
1 let cityIterator = new RegularIterator(cityService.loadData());
2 // cityIterator.printResults();
```

Most of the code that was originally created in the generator remains. Since multiple ways of iterating are required, another Iterator class which can do the opposite is necessary:

```
1  export class InverseIterator {
2    ...
3    this.properties = Object.keys(this.myIteratorObj);
4    this[Symbol.iterator] = function() {
5      this.count2 = 0;
6      this.isDone = false;
7      this.doneCountry = false;
8      this.properties = Object.keys(this.myIteratorObj);
9      this.count = this.properties.length-1;
10
11     let next = () => {
12       if(this.myIteratorObj[this.properties[this.count]].length<= this.
           count2) {
13         this.doneCountry = true;
14         this.count2 = 0;
15
16         if (this.count == 0 ) {
17           this.isDone = true;
18         } else {
19           this.count--;
20         }
21       } else {
22         this.doneCountry = false;
23       }
24
25       return !this.isDone? { done: false,
26         value: this.myIteratorObj[this.properties[this.count]][this.
             count2++]
27       } : {done: true};
28     }
29     return { next };
30 };
31   ...
```

Decorator

The decorator has the task of providing an additional information to another classes in run-time. It is required to have the information about the timestamp of our CityService and UserService instances (to have a reference of when they were instantiated), so that it can be injected in these 2 classes. First the decorator itself is modified to add a time information to an object:

```
1  export function datestamp(target) {
```

```
2   target.prototype.setCreated = () => {
3     target.prototype.id =  new Date().getMilliseconds();
4   };
5 }
```

since the function *datestamp* is exported, it can be now injected into other constructors with the decorator syntax:

```
1  // city.service.js
2  import { datestamp } from './datestamp.decorator';
3
4  @datestamp
5  export class CityService {
6  ...
7
8  // user.service.js
9  import { datestamp } from './datestamp.decorator';
10 @datestamp
11 export class UserService {
12   ...
13   this.setCreated();
14   ...
```

The class decorator is defined once and only for all instances of the class. Which means the timestamp is defined per class and not per instance.

Observer

The Observer pattern will be composed of a StateManager, which can handle the subscriptions of Observers, and the Observer itself. The instances of User and CityService will contain an internal observer and callback method, so they get triggered whenever the StateManager commands.

```
1  import Rx from 'rxjs/Rx';
2
3  export class StateManager {
4    subject;
5    observables;
6    constructor() {
7      this.observables = [];
8      this.subject = new Rx.Subject();
9    }
10
11   // adds listeners and their callbacks to the subscription list.
12   addListener(observer, cb) {
13     this.subject.subscribe({
```

```
14        next: (v) => {
15          cb;
16        }
17      });
18    };
19
20    removeListener(observer) {
21      observer.unsubscribe();
22    }
23
24    notifyObservers() {
25      this.subject.next();
26    }
27  }
```

With the Manager it is now possible to add subscriptions from our index file. They can represent any observer which is interested in getting notified. The example below shows a case with User and CityService:

```
1  // index.js
2
3  let centralObserver = new StateManager();
4  let user1 = new User();
5  centralObserver.addListener(new NotificationServiceObserver(
       cityServiceInstance.created), cityServiceInstance.update());
6  centralObserver.addListener(new NotificationServiceObserver(user1.
       created), user1.update());
7  let user2 = new User();
8  centralObserver.addListener(new NotificationServiceObserver(user2.
       created), user2.update());
9  centralObserver.notifyObservers();
10
11 // output
12 city  775 subscription alert!
13 user  790 subscription alert!
14 user  790 subscription alert!
```

5.2.2 Standard design patterns approach

It is possible to reproduce all the design patterns from previous section in a classical manner, which means, without any experimental features of JavaScript and using the same structure as the original implementation of the design patterns. Some of the ideas in the patterns below are inspired by (Osmani (2012)).

Singleton

The singleton approach to keep a single CityService does not change in the casual design patterns code. There is no specific ES6 JavaScript or syntax sugar involved, so the code remains almost the same.

```
1   var CityService = (function () {
2     this.instance = null;
3     this.path = null;
4     this.id = null;
5
6     function createInstance(cityData) {
7       var object = new Object("I am the instance");
8       object.cityData = cityData;
9       return object;
10    }
11
12    function setSource(instance, path) {
13      instance.path = path;
14    }
15
16    return {
17      getInstance: function () {...}
18      update: function () {...}
19      ...
20    }
21  }
```

Iterator

The classical Iterator is implemented via 2 classes: the CircularIterator and the Aggregate. The Iterator class contains the default methods from literature, while the Aggregate contains the object which is iterated. The required methods are set as part of the prototype of the Iterator class.

```
1
2   var CircularIterator = function(items) {
3       this.index = 0;
4       this.items = items;
5   }
6
7   CircularIterator.prototype = {
8       first: function() {
9         this.index = 0;
10        this.items[this.index];
11      },
```

```
12      next: function() {
13        return this.items[this.index++];
14      },
15
16      currentItem: function() {
17        return this.items[this.index];
18      },
19
20      isDone: function() {
21        return this.index >= this.items.length-1;
22      }
23  }
24
25  // Aggregate
26  function aggregate() {
27      fs.readFile('../json/sample1.json', 'utf8', function (err, response)
          {
28        if (err) throw err;
29
30        var data = JSON.parse(response);
31        var regularIterator = new RegularIterator(data);
32        var totalCitites = 0;
33        for (var cities = regularIterator.first(); !regularIterator.isDone
            (); cities = regularIterator.next()) {
34          var cachedLength = cities.length;
35          for(var k = 0; k < cachedLength; k++);
36        }
37      });
38  }
```

To implement an alternative iterator, for instance an inverse iterator, a few changes are done in the prototype:

```
1
2  // InverseIterator
3
4  ...
5    first: function() {
6      this.index = this.items.length-1;
7    },
8    next: function() {
9      return this.items[this.index--];
10   },
11 ...
```

The functions first, next and isDone are adjusted so that the iterator works in reverse order.

Decorator

The Decorator classic pattern is composed by 2 main functions: A Component which represents the element to be decorated, and the Decorator, which receives the Component and decorates it. The following example has a simplified version of CityService that will be decorated:

```
1
2  // Component
3  var User = function(name) {
4    this.name = name;
5
6    this.printOut = function () {
7      console.log('user:' + this.name);
8    }
9  };
10
11 // Decorator
12 var DecoratedUser = function (user, created_date) {
13   this.name = user.name;
14   this.created_date = created_date;
15
16   this.printOut = function () {
17     console.log('user ' + this.name + ' , ' + this.created_date + '
         created.');
18   }
19 };
20
21 // Client
22
23  var normalUser = new User('testUser');
24  normalUser.update();
25  var specialUser = new DecoratedUser(normalUser, new Date());
26  specialUser.update();
```

The same procedure will have to be applied to the City class in order to add timestamps to cities. The timestamp is individual per instance of User. Ideally, both component and decorator classes should have the same methods.

Observer

The Observer can be implemented with the traditional design pattern rules with a StateManager very similar to the modern implementation. Since the third party support is not used, StateManager contains a list of functions and a prototype method notifyObservers, which

then iterates over an array of subscriptions and execute their callbacks. It is then possible to assign User and CityService functions to be registered by the StateManager.

```

function StateManager() {
  this.observables = [];
}

StateManager.prototype = {
  subscribe: function(observableFn) {
    this.observables.push(observableFn);
  },

  unsubscribe: function(fn) {
    this.observables = this.observables.filter(
      function(item) {
        if (item !== fn) {
          return item;
        }
      }
    );
  },

  notifyObservers: function(o, thisObj) {
    var scope = thisObj;
    this.observables.forEach(function(item) {
      item.call(scope, o);
    });
  }
}
...

function testClassicObserver() {
  var managerSample = new StateManager();
  managerSample.subscribe(notificationSampleHandler);
  managerSample.subscribe(notificationSampleHandler2);
  managerSample.notifyObservers();
  managerSample.subscribe(notificationSampleHandler3);
  managerSample.notifyObservers();
}
```

5.3 Pattern Conclusion

After successfully solving the proposed problems thanks to the generator and the appropriate design patterns, a few assumptions can be made concerning advantages and disadvantages of the approach that was taken. Additionally it is possible to review the modern patterns according to the initially proposed criteria (safeness, simplification, ease of writing and performance).

5.3.1 Singleton

The classical implementation of singleton consists of calling a self invoking function CityService, which will then return the main entry point of the Singleton, getting its instance. The modern implementation makes use of class to simplify the code. It contains a global instance outside the class scope, and the moment we instantiate the class the constructor checks for the instance. The singleton implementation is the one which differs the least from the classical approach. Therefore not much can be said about advantages or disadvantages; it is in JavaScript also a simple implementation and was enough to fulfill our requirements of a single point of access to city data. Regarding ease of writing, one can say it is a more straightforward approach to write a singleton via the modern approach using constructor rather than having to write self invoking functions.

5.3.2 Iterator

The classical approach consists of an Iterator class with the required prototype as defined by literature. The Aggregate class is well structured and very simple to handle, creating a new Iterator instance passing the list. The modern iterator contains less control methods and more implicit functionality, since it transforms a list by giving it *Symbol.iterator* properties, allowing a more straightforward iteration of the aggregate via native *for..in* loop for instance. Although it is a cleaner approach to obtain the final result (ease of writing), it provides fewer interfaces to iteration. For both classical and modern approaches, the challenge of molding the Iterator in a way to adjust to the type of data provided is the same.

Regarding performance of iteration considering classical and modern approaches, it is known that traditional JavaScript *for* loops deliver the same results Adams (2015) regardless of inverted or standard mode. It should be validated if modern loop structures (forEach, for..in) are comparably faster than the traditional loops. In order to verify this for the problem

described, the Iterator pattern was tested for 6 different samples of city Data. For each sample the classical Iterator with normal loop structure was compared to modern loop function on a iterator object resulting from a modern iterator.

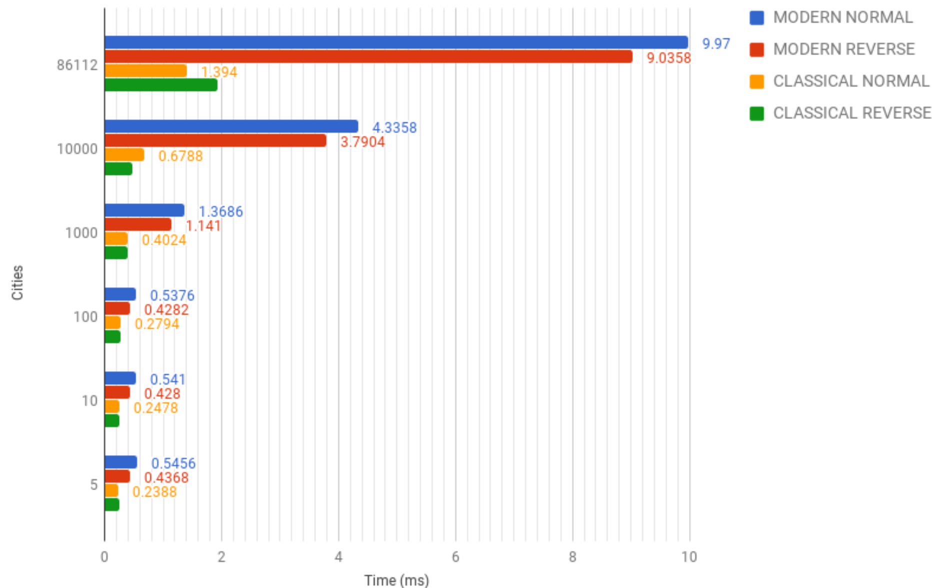

Figure 5.1: Comparison between loop methods on classical and modern Iterator approach

As it can be seen in the chart above, the theory that classical for loops show better performance remains true. The Modern Normal and modern reverse approaches, which make use of the *for..in* structure, take much longer to operate through a longer sample of cities than the classical approaches. The forward and reverse mode of the classical *for* loop performs the loop task in the shortest time.

5.3.3 Decorator

The decorator pattern via modern design patterns is implemented in a very simple way, yet it did exactly what a classical decorator would do: add certain logic to an object dynamically. The classical implementation requires more code: if we want to have a decorated city, it is necessary to implemented an appropriate DecoratedCity class with the same signature. However, classical approach by default provided instance-unique attributes, while modern decorator works as a class decorator rather than instance decorator.

5.3.4 Observer

The modern implementation of the Observer Pattern shows that by using Reactive programming it has become much more simple to control the application behavior. With much less code it was possible to solve the simple problem of notifying users about chances in other elements. Of course Ease of writing is affected since one has to rely on third parties to have an Observer working. In an classical Observer Approach, although the logic would be the same and nothing more than plain JavaScript is required (besides basic Array operations), writing more code is necessary.

5.3.5 Considerations

It is necessary to say that some of the above features are still in development by ECMA. This means, the level of similarity between modern implementations of Design Patterns and classical ones might change.

6 Conclusion

In this bachelor thesis it was analyzed how Design Patterns perform in real world-based scenarios with modern features of JavaScript and how different they can be structured depending on the libraries available. An approach without modern features into the same scenario was done and a comparison with a modern version regarding performance, security and ease of writing was made.

As an additional motivation to modern approaches, a code generator was introduced. The generator made possible to configure templates to shape as base code for design patterns, which can be installed in a project for customization and usage. The generator was used to setup the proposed modern versions of Singleton, Iterator, Decorator and Observer patterns. After comparing with traditional implementations, it has been verified that, in general, implementing alternative versions of design patterns with modern JavaScript contributes to the code quality and reduces the effort to setup a pattern, a benefit which comes with the cost of getting familiar with advanced features of JavaScript and relying sometimes in third party support when building the solutions, due to the fact that most of the features are still under development and not officially available for demonstration, such as Observable objects. When it comes to performance, original iterator methods from JavaScript still perform tasks faster than the new loop strategies being developed.

As Design Patterns were originated as a single methodology which can solve problems for any kind of system, the same can be said about modern versions of them. The Design patterns chosen are the ones which are directly impacted by modern JavaScript; for several other patterns, these extra features do not necessarily make the implementation different. In general, a good knowledge of design patterns as they were introduced in history is essential to maintain a good quality product, while keeping in mind that JavaScript is a very much "alive" language that evolves daily and will introduce many new features in the future, potentially improving the way our Software works today and tomorrow.

Bibliography

[Adams 2015] ADAMS, Chad R.: *Mastering JavaScript High Performance*. 1. Packt Publishing, March 2015

[Carlstein] CARLSTEIN, Joakim: *Slush - The streaming scaffolding system*. http://slushjs.github.io/#/. – Accessed: 2017-06-24

[Dahl] DAHL, Ryan: *Node.js® - a JavaScript runtime built on Chrome's V8 JavaScript engine*. https://nodejs.org/en/. – Accessed: 2017-07-24

[Diakopoulos und Cass 2017] DIAKOPOULOS, Nick ; CASS, Stephen: Interactive: The Top Programming Languages 2017. In: *IEEE Spectrum* (2017), July

[Dijkstra 1968] DIJKSTRA, Edsger: Goto Statement Considered Harmful. In: *Communications of the ACM* 11 (1968), March, Nr. 3, S. 147–148

[ECMA] ECMA: *Observable Type*. https://tc39.github.io/proposal-observable/. – Accessed: 2017-04-23

[Foote und Yoder 1999] FOOTE, Brian ; YODER, Joseph: Big Ball of Mud. (1999), June. – Accessed: 2017-06-15

[Gamma u. a. 1994] GAMMA, Erich ; HELM, Richar ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Addison Wesley, 1994

[Haim] HAIM, David: *Json mapping the countries of the world to their cities*. https://github.com/David-Haim/CountriesToCitiesJSON. – Accessed: 2017-08-20

[International 2017] INTERNATIONAL, Ecma: ECMAScript 2017 Language Specification. (2017), June

[Osmani 2012] OSMANI, Addy: *Learning JavaScript Design Patterns*. 1. O'Reilly, 2012

[Schwensen 2016] SCHWENSEN, Lars C.: Embedded reactive systems - The impact of modern C++ standards on selected design patterns. (2016)

[Timms 2014] TIMMS, Simon: *Mastering JavaScript Design Patterns*. Packt Publishing, November 2014

[Yeoman a] YEOMAN: *generators repository | Yeoman*. http://yeoman.io/generators/. – Accessed: 2017-04-09

[Yeoman b] YEOMAN: *The web's scaffolding tool for modern webapps | Yeoman*. https://github.com/yeoman/yeoman. – Accessed: 2017-04-09

Declaration

I declare within the meaning of section 25(4) of the Ex-amination and Study Regulations of the International De-gree Course Information Engineering that: this Bachelor report has been completed by myself inde-pendently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, October 5, 2017	
City, Date	sign