TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ALEKSI SINISALO

WEB FRONTEND COMPONENT QUALITY MODEL

Master of Science thesis

Examiner: Prof. Hannu-Matti Järvinen
The examiner and topic of the thesis were approved on 31st May 2017

ABSTRACT

ALEKSI SINISALO: Web Frontend Component Quality Model
Tampere University of Technology
Master of Science Thesis, 73 pages, 3 Appendix pages
September 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen

Keywords: software quality, quality model, web frontend development, web frontend components

Web frontend application developers utilize many components in their work that provide functionality required by the application under development. The components are typically written in JavaScript and may have been developed by 3rd parties or inside the company. The quality of the selected components plays a major role in the overall quality of the web frontend application that they are utilized in. Additionally, the component quality affects the desirability of the component in the eyes of the web application developers that might potentially utilize the component in their application. As an implication, the developers of these components want to them to be high-quality and easy to use.

Thus, the problems that this thesis is seeking answers to are how to develop easy-to-use high-quality components and how to measure web frontend component quality. This thesis presents the web frontend component quality model as an answer to these problems. The model is based on web frontend development and component characteristics and research on software component quality models. Both are discussed in the literature review part of this thesis.

The web frontend component quality model divides the component quality hierarchically to 4 levels that are quality characteristics, quality sub-characteristics, quality attributes and quality measures. Quality characteristics are high-level abstractions of quality such as functionality and usability that are further specified by the sub-characteristics and attributes. The quality measures are concrete instructions on how to measure values for the quality attributes. The web frontend component quality model consists of 6 quality characteristics, 13 quality sub-characteristics, 30 quality attributes and measures for them.

The quality model was tested and evaluated by measuring the quality of the report editor component that is developed by Wapice Ltd. The quality evaluation was able to measure values for the quality attributes according to the model. Additionally, numerous suggestions were provided on how to improve the quality of the report editor component implementation and documentation. Among the improvement suggestions were for example, improving the configurability of the component through configurations object and events interface, providing HTML-based documentation and loading type coverage improvement by adding support to CommonJS and AMD module types.

TIIVISTELMÄ

ALEKSI SINISALO: Web frontend komponenttien laatumalli
Tampereen teknillinen yliopisto
Diplomityö, 73 sivua, 3 liitesivua
Syyskuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: ohjelmistojen laatu, laatumalli, web frontend kehitys, web frontend komponentit

Web frontend-sovellusten kehittäjät hyödyntävät työssään monia komponentteja, jotka tarjoavat toiminnallisuuksia kehitettävänä olevaan sovellukseen. Nämä komponentit on usein kirjoitettu JavaScript-ohjelmointikielellä ja ne ovat kolmannen osapuolen tai yrityksen itsensä toteuttamia. Web-sovelluksessa käytettävien komponenttien laatu on suuressa roolissa sovelluksen kokonaislaadun kannalta. Lisäksi komponentin laatu vaikuttaa sen kiinnostavuuteen niiden web-kehittäjien näkökulmasta, jotka voisivat mahdollisesti käyttää sitä omassa sovelluksessaan. Tästä seuraa, että komponenttien kehittäjät haluavat komponenttiensa olevan korkealaatuisia ja helppokäyttöisiä.

Ongelmat, joihin tässä diplomityössä etsitään vastauksia ovat, kuinka kehittää helppokäyttöisiä ja korkealaatuisia komponentteja ja kuinka mitata web frontend-komponenttien laatua. Tämä diplomityö esittää vastauksena web frontend-komponenttien laatumallin. Malli pohjautuu web frontend-kehityksen ja komponenttien erityispiirteisiin sekä tutkimuksiin ohjelmistokomponenttien laatumalleista. Molempia aiheita käsitellään tämän diplomityön kirjallisuuskatsaus-osiossa.

Web frontend-komponenttien laatumalli jakaa komponenttien laadun hierarkkisesti neljään tasoon, jotka ovat laadun erityispiirteet, laadun alierityispiirteet, laatuominaisuudet ja laatumittaukset. Laadun erityispiirteet ovat korkean tason abstraktioita, kuten toiminnallisuus ja käytettävyys, joita tarkennetaan edelleen ali-erityispiirteillä ja ominaisuuksilla. Laatumittaukset ovat konkreettisia ohjeita laatuominaisuuksien mittaamiseen. Web frontend-komponenttien laatumalli koostuu 6 laadun erityispiirteestä, 13 laadun alierityispiirteestä, 30 laatuominaisuudesta ja niiden mittauksista.

Laatumallia testattiin ja arvioitiin mittaamalla raporttieditorikomponentin laatua. Raporttieditorikomponentti on Wapice Oy:n kehittämä. Laatuarviointi suoritettiin mittaamalla arvot laatuominaisuuksille mallin ohjeiden mukaisesti. Lisäksi, mittausten perusteella tuotettiin useita ehdotuksia siihen, kuinka parantaa raporttieditori-komponentin totetuksen ja dokumentaation laatua. Kehitysehdotusten joukossa oli esimerkiksi konfiguroitavuuden kehittäminen toteuttamalla konfiguraatio-objekti ja tapahtumarajapinta komponentille, tarjoamalla HTML-pohjainen dokumentaatio ja komponentin lataustyyppikattavuuden parantaminen tarjoamalla tuki CommonJS- ja AMD-moduulityypeille.

PREFACE

I would like to thank professor Hannu-Matti Järvinen for his comments and counseling during the writing process of this thesis. Additionally, thank you my parents Hannele and Seppo for your support to me during my studies. Thank you also my brother and sister Markus and Susanna for your support in our studies together.

Tampere, 17.9.2017

Aleksi Sinisalo

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AJAX	Asynchronous JavaScript and XML
AMD	Asynchronous module definition
API	Application programming interface
CBSD	Component based software development
CBSE	Component based software engineering
CORS	Cross-origin resource sharing
COTS	Commercial off the shelf
CSS	Cascading style sheets
DOM	Document object model
ES6	ECMAScript 6
HTML	Hypertext markup language
HTTP	Hypertext transfer protocol
IoT	Internet of things
JSON	JavaScript object notation
MVC	Model view controller
NPM	Node package manager
ORM	Object-relational mapping
PHP	Hypertext preprocessor
REST	Representational state transfer
SPA	Single-page application
SQL	Structured query language
UI	User interface
UMD	Universal module definition
UML	Unified modeling language
URL	Uniform resource locator
XML	Extensible markup language

1. INTRODUCTION

Web application developers are nowadays utilizing different kinds of commercial, open source and company internally developed libraries and components. This applies to both backend and frontend development. The libraries may range from providing mathematical functions or data handling to interactive user interface (UI) components. The aim for utilizing these libraries and components is to lower the development time and costs by reducing the amount of programming that needs to be done for each application.

The developers of these libraries and components naturally want their product to succeed and be utilized in different applications. Thus, it is important for the component developers to produce a high-quality component that is easy-to-use and configurable to suit the needs of various web applications.

JavaScript-based web frontend UI libraries or components are one branch of these libraries. These libraries provide interactive widgets such as charts, dialogs and tabs that developers can configure and use in their own applications. This thesis focuses on providing means to web frontend UI component developers to assess the quality of their component and improve it in a way that would make it more appealing to web application developers.

Thus, the goal of this thesis is to develop a web frontend component quality model that can be utilized to evaluate the quality of web frontend components. The aim of the quality evaluation is to assess the overall quality of the component. Additionally, the aim of the quality evaluation is to provide information to act as a base for suggestions on how to improve the component quality. The web frontend component quality model is evaluated by executing a quality evaluation of the report editor component that is extracted from a web based internet of things product IoT-Ticket. The report editor component has been extracted to be utilized in other applications that require its functionality. IoT-Ticket is developed by a Finnish software and electronics company Wapice Ltd.

In order to develop a web frontend component quality model, the characteristics of web applications, web application development and web frontend components need to be identified. Software quality and software component quality are also relevant subjects for this thesis. Software component quality has seen some research on general software components regarding component based software development (CBSD) and their results can also be utilized when developing the quality model.

Thus, the contents of this thesis are first, introducing the context in Chapter 2 by providing information of web frontend applications and their development. Chapter 3 describes the

software component quality evaluation by first discussing general software quality evaluation and second presenting results of software component quality evaluation research. In Chapter 4, the web frontend component quality model is derived based on the software component quality research results presented in Chapter 3 and web application characteristics presented in Chapter 2. Chapter 5 introduces the IoT-Ticket product and the report editor component in addition to executing the actual quality evaluation of the report editor component utilizing the web frontend component quality model derived in Chapter 4. Additionally, in Chapter 5, the web frontend component quality model is evaluated based on how well it performed and produced results. Chapter 6 presents the conclusions of the thesis.

2. WEB FRONTEND APPLICATIONS

With the emerge of popular JavaScript frameworks such as AngularJS and React.js the web development industry is leaning towards transferring more application logic from the backend to the frontend side. In addition to frameworks, the rising demands for the frontend web development has also generated a myriad of open source and commercial JavaScript libraries and components that developers can use in their own projects. Practically every modern web application utilizes these libraries and it has become increasingly important to choose the right components from the ever-growing pool of available solutions.

The transition in the industry has created a web application structure called single-page application (SPA) which promotes a well-structured business logic oriented frontend code. The separation to frontend and backend in web development is also becoming clearer as the frontend applications are developed and deployed with their own tools independent of the backend applications.

Previously, the frontend logic was more strongly coupled with the backend logic as the backend utilized server-side rendering to construct dynamic web pages presented to the user. The frontend could then be perceived as a collection of JavaScript files that provided enhancements to the pre-rendered web page rather than a standalone application. (Fink & Flatow 2014a, para 2) Modern web frontend application however, developed usually with the aid of a framework, can be identified as a distinct entity separated from the backend.

Thus, modern web frontend application is the part of a web application that the user directly interacts with. The part visible to the user of the frontend application is called a user interface (UI). The frontend application is usually accessed by a web browser software but it is also possible to develop and utilize a separate client software.

Usually web frontend application is communicating with web backend application that, for example, offers access to user authentication service and data from persistent data storage. Though, it is also possible to develop a frontend application with no backend connection and run it on a browser, its uses are mainly limited to simple non-data related tasks or UI demonstration purposes.

2.1 Web application structure

Together with a backend application, a web frontend application forms an entity called web application. Web application is running on a web server software that receives and processes requests from clients. Web applications can be roughly divided into two categories: the traditional web application and the single page application. Communication

between the clients and the web server is achieved by using the hypertext transfer protocol (HTTP).

2.1.1 Traditional web application

The traditional web application has a server-centric approach. This implies that almost all user interactions are sent as HTTP requests to the web server. One characteristic of traditional web applications is that the browser page is refreshed every time the server responds to a HTTP request of the client. The requests can be made to directly access static resources on the server such as images or hypertext markup language (HTML) files. Additionally, with the introduction of server-side scripting languages like hypertext preprocessor (PHP) the server can also provide dynamic content by executing server side scripts to generate HTML pages that are then sent as HTTP response to the client. (Fink & Flatow 2014a, para 7)

However, nowadays traditional web applications have usually implemented a router which handles the requests which implies that the user does not have to access the server files directly by their name. The use of a router usually implies the use of the Model-View-Controller pattern (MVC) or similar that divides the web application logic to distinct parts. In the case of the MVC the router may be configured to map uniform resource locator (URL) routes to controller action methods for example route */News/Create* could be mapped to NewsController's *Create* method. The called method has access to the HTTP request and its parameters and may use them to form the response and return it to the client. When router is routing the HTTP request it has access to the HTTP header fields which allows the reuse of routes for different actions to the same resource. For example, */News/Create* with HTTP GET could be used to return a view for creating a news item and HTTP POST could be used to create a news item (docs.microsoft.com 2017a). Web application that has implemented a router may still serve static files but the router will map the reserved URL routes to the controller methods. (msdn.microsoft.com 2017)

Web application is accessed by the user through views on the browser. These views may be constructed by the server using HTML syntax in combination with templating engine language. When the controller loads a view to be sent as a response to the client's request it passes view data parameters or view model object to the view which allows the templating engine to access the data it needs to show on the view. Typically, templating engines provide support for more advanced templating other than printing the view model values on the view. For example, support for iterating over arrays or logical operations based on view model values. After the view is constructed the application may send a HTTP response with the HTML content of the view to the client. (docs.microsoft.com 2017b; docs.microsoft.com 2017c)

Web application usually needs to provide persistent data storage to store business data. This is usually achieved by using database software. Database software stores application data to disk on the server it is running on. Most commonly used database type is relational database which stores data in tables. Tables consist of rows and columns. Rows represent business data in the application for example a news item. Columns in a row represent properties of the rows for example title, author and text content of the news item. Relational database can be queried with Structured query language (SQL) to receive the data needed by the application. For example, ”get news item with author A.S.” would translate to SQL as `SELECT * FROM NewsItems WHERE Author = 'A.S.'`. (docs.oracle.com 2017)

Usually database is accessed on application level by utilizing object-relational mapping (ORM) or similar technique that abstracts the database handling. Thus, the data can be accessed using the same programming language as the application and the database rows are mapped to objects called models in the MVC pattern. The ORM may utilize SQL in its internal implementation but programmers may use language supported queries that may be more intuitive. (entityframeworktutorial.net 2017)

2.1.2 Modern single-page application

The fundamental difference of SPA to traditional web application is that the client content of the SPA consists of a single HTML page called entry point while traditional web application constructs different HTML pages based on user's request. The SPA entry point page dynamically alters its contents in response to user's actions. With dynamic content alteration, the client doesn't need to send request to the server to change the ”web page” and a full page reload is not required by the browser. The content alteration is achieved by utilizing browser supported script language called JavaScript. Usually SPA structure is achieved by programming with the aid of a frontend framework which supports the division of the pages to modules like the views in traditional web application. (Fink & Flatow 2014a, para 8)

Traditional web application provides the router to give human readable URLs to the view pages. Additionally, requesting an individual page for each request also enables the browser's history functionalities: back and forward. Similar effects are possible with SPA even though its content consists of a single HTML page. The routing can be defined on the frontend code to load a specific frontend module as the page content when a certain URL is entered usually by utilizing a frontend framework or library that is using the HTML5 history API. (Fink & Flatow 2014b, para 5)

SPA is usually connected to backend Representational State Transfer (REST) API. REST architecture sets several restrictions, for example, to the structure and the HTTP method usage of the API. The API is used, for example, to authenticate users and to access persistent data storage to load data content to the frontend application. When certain module

is loaded on a SPA it usually sends an XML HTTP Request to the API using Asynchronous JavaScript and XML (AJAX) technology. AJAX request is like a regular HTTP request and its response is usually in extensible markup language (XML) or JavaScript object notation (JSON) format. Frontend application can parse the response data and show it on the page content utilizing JavaScript. (Fink & Flatow 2014a, para 8; Fink & Flatow 2014b, para 8)

Additionally, when API is operating on a different web-server or web server port than the frontend application cross-origin resource sharing (CORS) must be considered. CORS is a mechanism that allows requests to be made to a different domain than from where the request originated which is normally prevented by same origin policy. For example, request from localhost:80 (frontend application) to localhost:8080 (REST API). By default, CORS is disabled and must be enabled for specific domains the API wants to allow access from. (developer.mozilla.org 2017b)

2.2 Web frontend technologies

Web frontend development utilizes many technologies. The primary technologies that are used in practically every web frontend application are HTML, document object model (DOM), JavaScript and cascading style sheets (CSS).

2.2.1 HTML

Hypertext Markup Language (HTML) is a language used to construct web pages. HTML documents are interpreted by the web browsers that form the visual representation of the HTML document. While the HTML itself is standardized all the browsers are not interpreting it the same way. Therefore, there may be differences in the presentation of the same HTML document across different browsers and cross browser testing is usually required to verify that the web application utilizing HTML is operating correctly on all required browsers. (Brooks 2007, p. 1)

The HTML document consists of elements. Elements in turn consist of tags and content. Majority of the elements have a starting tag and an ending tag but there are also elements which consist only of a single tag. The starting and ending tags define the ”body” of the element and the content related to the element is inserted between the tags. The browser interprets the element and applies its effect to its content which can be observed on the visual representation of the HTML document. (Freeman 2011, pp. 13-15)

The elements have relationships between themselves depending on their position in relation to each other in the HTML document. The HTML document supports a tree like structure which allows elements to contain other elements. An element that contains other elements is called a parent. An element that is defined inside another element is called a child element. Two elements that are child elements to the same parent element are called

siblings. An element that is more than one level deep in the element tree is called a descendant to the parent element that is more than one level up in the element tree. A parent element that is more than one level up in the element tree is called an ancestor. In addition to providing hierarchical structure to the HTML document the element relationships can also be used to apply Cascading Style Sheets (CSS) styles to elements that are in similar position in the element tree. (Freeman 2011, p. 23) CSS is introduced later in this chapter.

HTML elements may be configured by multiple optional attributes. Some attributes are global that can be used with any element and some are element specific. Most attributes are name-value pairs that provide extra information about the element or configure it in some way. For example, an *a*-element which is used to create hyperlinks has a *href* attribute which can be used to provide a reference to another location. (Freeman 2011, p. 19)

The most important global attributes are the *class* and *id* attributes. The class attribute is used to classify or categorize elements (Freeman 2011, p. 26). Multiple elements may have the same class attribute which usually implies that they have a similar purpose in the application. Element may also have multiple values for class attribute separated by space character (Freeman 2011, p. 26). The main use cases for class attribute are applying CSS styles to elements with certain class attribute value or executing JavaScript code in the context of those elements (Freeman 2011, pp. 27-28). The id attribute functions similar to class attribute except that it provides a unique identifier to an element (Freeman 2011, p. 32). While the id attribute value should be unique in an HTML document (W3.org 2017) it is still possible to assign multiple elements the same id. However, this may cause unexpected results when executing scripts that rely on the id attribute.

While some HTML elements affect the visual representation of the web page their main purpose is to provide structure for the content. For example, the *H1* element describes a level 1 heading and the *p* element describes a text paragraph. The use of these elements should only provide information on what the content is, not how it should be presented. Mainly for historical reasons originating from the early versions of HTML some elements still affect the representation of the content. Now CSS has been widely accepted to provide the means to alter the presentation of the content separate from the structure. (Freeman 2011, p. 16)

A minimum valid HTML document contains *html*, *head*, *title* and *body* elements (Brooks 2007, p. 1) and document type declaration at the start of the document. These elements are required for HTML document to be valid but most browsers can interpret also invalid HTML documents. The *html* element contains all other elements on the HTML document. The *head* element contains metadata information of the document including the *title* element. Head element may also be used to load CSS and script files. The *body* element contains the displayable content of the HTLM document. (Freeman 2011, pp. 22-23)

2.2.2 DOM

Document Object Model (DOM) is an object model representing the structure of an HTML document. DOM allows programmatical manipulation of the HTML document by utilizing JavaScript scripting language. With DOM it is possible, for example, to dynamically create and remove elements and listen to events that elements send in response to user actions. In DOM, each element in the HTML document is represented by a JavaScript object that contains properties and methods that can be used to manipulate the element. Most browsers provide native JavaScript support to access DOM but there are also JavaScript libraries such as jQuery that can be used to access DOM. (Freeman 2011, pp. 633-635)

2.2.3 JavaScript

JavaScript is an interpreted object-oriented programming language that is executed on the client side (Brooks 2007, p. 3). JavaScript does not need to be compiled and most web browsers have an interpreter which executes the JavaScript code. JavaScript was developed to create interactive HTLM pages by utilizing the DOM without the need of loading a new HTML page from the server. (Brooks 2007, pp. 3-4)

The simplest way of including JavaScript code into a HTML page is by using the *script* tags. The JavaScript code is written inside the script tags as their content. Browser will execute the JavaScript when the HTML document is loaded. (Brooks 2007, p. 9) An alternative to writing the JavaScript code inside the script tags is to provide a path to a JavaScript file using the script tag *src* attribute (Brooks 2007, p. 31). While it is possible to write all JavaScript code in the HTML pages inside the script tags, using separate JavaScript source files is the advised way when writing web applications.

The most important concepts in JavaScript programming are *variables*, *functions* and *objects*. JavaScript is a loosely typed language which indicates that variable types are not explicitly defined. The variable type will be determined by the value assigned to the variable. A variable is also not locked into a certain type once a value has been assigned to it. The type may change if a new value is assigned into the same variable. In addition to primitive types (*string*, *number*, *Boolean*), it is also possible to assign functions or objects into a variable. This aspect of the JavaScript language might be controversial to programmers that are used to strongly typed languages. Thus, JavaScript preprocessors such as TypeScript (typescriptlang.org 2017) have been developed that provide strong typing and other features such as interfaces to the JavaScript (Fink & Flatow 2014a, para. 6). (Freeman 2011, p. 77)

In JavaScript, like in other programming languages, functions can be defined to accept parameters and return a value. Function parameters and return values are also loosely typed in JavaScript implying that the caller of the function may insert expressions of any

type into the parameters. Caller may also decide to omit any parameters when calling a function. In that case, the missing parameters will be *undefined* which is a reserved word in JavaScript for values that have not been defined. Caller may also present more parameters than required to the function. Then, the additional parameters are ignored. (Freeman 2011, pp. 75-77)

JavaScript objects, unlike in strongly typed languages, are not instances of any specific class. They are dynamic in nature and can be manipulated at will before and after their initialization. Objects have properties which can be any types allowed to regular variables including primitive types, other objects and functions. Functions as object properties are called methods. If methods want to access other properties of the object it can be done through *this* keyword. (Freeman 2011, pp. 79-81)

Function declares a private scope in JavaScript. Variable that is declared inside a function will only be visible inside the function and is called a local variable. Variables that are declared outside functions are called global variables and they may be used also inside any functions. Global variables can also be used by any other scripts that are loaded into the same HTML page after the global variable declaration. (Freeman 2011, p. 77)

As stated before, JavaScript can be utilized in manipulating DOM. For example, a simple event handler can be attached to a button element click to change the text inside an HTML element with the JavaScript code shown on Figure 2.1.

```
<html>
    <head>
    </head>

    <body>
        <p id="text">Default Text</p>
        <button onClick="changeText();">Change text</button>
    </body>
    <script>
        function changeText() {
            var textElement = document.getElementById('text');
            textElement.textContent = 'New text';
        };
    </script>

</html>
```

Figure 2.1 JavaScript event handler

2.2.4 CSS

Cascading Style Sheets (CSS) are used to modify the presentation of an HTML document. Styles are key-value pairs where key is a CSS property name and value is a value for the property. (Freeman 2011, pp. 39-40) There are three methods of applying CSS styles to an HTML document. *Inline styles*, *embedded styles* and *external styles*. Inline styles are

using the global *style* attribute on HTML elements. Style attribute accepts CSS style key-value pairs as its value and formats the presentation of the element appropriately. Embedded styles are defined inside *style* tags in the *head* section of an HTML page. External styles utilize styles that are defined in a separate *CSS file* and loaded to the HTML page in the *head* section using the *link* tag. External style sheets are the preferred way of utilizing the CSS on web applications because that ensures the separation of presentation and content. (Collison 2006, pp. 5-7)

Sometimes there may be situations when element styling needs to be changed in response to user interaction. This is possible through JavaScript DOM manipulation. Element style can be changed by assigning a certain *class* or *id* to an element or directly accessing the style object of the element through DOM (W3Schools.com 2017).

Web applications are nowadays used by multiple different devices with different screen sizes. CSS media queries can be utilized to provide responsive web applications that conform to different screen sizes. Media queries can be used for example, to identify user's screen size and load a different style sheet accordingly. (developer.mozilla.org 2017a)

There are also existing frameworks such as Bootstrap that help styling frontend web applications (getbootstrap.com 2017). These frameworks provide themes that can be used for example, by defining certain *class* attribute values to elements. Styling frameworks can also provide functionality in addition to styling for example, styled dialogs that respond to user interaction. Frameworks can also aid in developing a responsive web application since they often have inbuilt responsiveness to different screen sizes.

How CSS can be utilized with style tags to for example, draw borders around a p-element and change the element background color can be seen on Figure 2.2.

```
<html>
    <head>
    </head>

    <body>
        <p>Text element</p>
    </body>
    <style>
        p {
            border: 1px solid black;
            background-color: green;
        }
    </style>

</html>
```

Figure 2.2 CSS styles on p element

2.3 Web frontend frameworks

When developing a modern single page application, it is often required to utilize a frontend framework. Frontend frameworks introduce design patterns such as MVC to the frontend development that have been used in traditional web application server side structure or desktop applications. One of the main reasons the frontend frameworks are used is that they provide a defined structure and separate the frontend code into maintainable and testable parts. In addition to providing code structure the web frontend frameworks can also contain many built-in features required in a single page application such as client-side routing, communication with the server and HTML templating. (Fink & Flatow 2014b, para. 4)

Many web frontend frameworks are opinionated in a way that they guide developers to a certain application structure that the framework supports. This may limit the developer's freedom on some cases but also offers specific building blocks for creating a well-structured application. Many popular frameworks also usually offer documentation and support through active developer community. (Osmani 2012)

2.4 Web frontend development tools

Traditionally, web frontend development has not required any special tools in addition to a text editor for writing the code. Due to the static nature of HTML and CSS and the interpretation of the JavaScript, the frontend development has quick feedback loops after developer has made changes to the source code. However, the increasing amount of complexity in the frontend code and growing amount of source files including the 3rd party library dependencies has caused the need for frontend code ”compilation” such as *minification* and *bundling* of the source files to reduce the load on browsers when initializing the application (docs.webplatform.org 2017). The utilization of JavaScript preprocessors also requires the frontend application to be compiled to JavaScript before browsers can run the application (Fink & Flatow 2014a, para. 6).

Some tools that have been developed to answer to these requirements are called JavaScript task runners such as Grunt and Gulp. These tools allow the developers to automate tasks for example, for compiling and deploying frontend applications and running automated tests by defining the tasks in JavaScript and running them from command line. (Ambler & Cloud 2015a, para. 1)

A need for simple and effective module loading for frontend applications arises as the amount of dependencies increase. It is possible to include all dependencies by utilizing the HTML script tags but it becomes increasingly tedious as the web application size grows (Bos 2015; Ambler & Cloud 2015b, para. 1). CommonJS used by Node.js and implemented for frontend use by the JavaScript library Browserify and Asynchronous Module Definition (AMD) implemented by the JavaScript library RequireJS provide a

way of loading dependencies by referencing them from other JavaScript files. The dependencies form a tree that ensures that every dependency is loaded in correct order. (Ambler & Cloud 2015b, para. 1)

Node package manager (NPM) has been used in Node.js backend environments for managing dependencies to external libraries. However, it can also be utilized in web frontend applications for dependency management and for running scripts similar to task runners. Together with EcmaScript6 (ES6) modules, that are not yet supported by browsers but available through transpiling to CommonJS or AMD, NPM packages provide a simple way of handling web frontend application dependencies. As stated before, ES6 modules are not yet supported by browsers but they can be seen as the future standard for handling frontend JavaScript modules. (Bos 2015; Brown 2016)

In addition to Browserify and RequireJS, there is also a module bundler called Webpack that supports loading of CommonJS, AMD and ES6 modules. (webpack.org 2017)

2.5 Web frontend UI components

There are different types of components and libraries used in frontend web development. For example, they might offer helper functions for data manipulation, supply routing or AJAX functionality or provide interactive widgets to use in web applications UI. In the context of this thesis we are more interested in the UI related components. Frontend UI components utilize HTML, JavaScript and CSS to provide functionality and styling.

One JavaScript library that provides multiple widgets that can be used in frontend web development projects is called jQuery UI (jqueryui.com 2017). It is widely used and its widgets provide simple configurations and methods to manipulate the widgets to be used in an application. jQuery UI tabs widget will be used as an example for a web frontend UI component. Image of the tabs widget can be seen on Figure 2.3.

Figure 2.3 jQuery UI tabs widget

UI components usually require a HTML element or elements to act as a "container" for the widget. The tabs widget requires a *div* element and inside it an *unordered list (ul)* HTML element with a *list item (li)* element for each tab to provide the tab navigation. Additionally, the actual content of each tab is inserted into their respective div elements.

The tabs widget can be attached to the DOM by calling the *tabs* method, provided by the jQuery UI library, to the container div DOM element. Calling the *tabs* method renders the tabs widget on the provided markup and will make the widget visible to the user. The tabs widget has a default CSS styling provided by the jQuery UI library. However, developers can also provide their own stylesheet to override the defaults as needed. (Wellman 2009, para. 2-3; jqueryui.com 2017)

The HTML markup and JavaScript required by the tabs widget can be seen on Figure 2.4.

```
<body>
<div id="tabs">
  <ul>
    <li><a href="#tabs-1">Tab 1</a></li>
    <li><a href="#tabs-2">Tab 2</a></li>
    <li><a href="#tabs-3">Tab 3</a></li>
  </ul>
  <div id="tabs-1">
    <p>Tab 1 content.</p>
    </div>
  <div id="tabs-2">
    <p>Tab 2 content.</p>
  </div>
  <div id="tabs-3">
    <p>Tab 3 content.</p>
  </div>
</div>
</body>
<script>
  $( function() {
    $( "#tabs" ).tabs();
  } );
</script>
```

Figure 2.4 jQuery UI tabs HTML markup and JavaScript

In addition to default functionality, the UI components can usually be provided with configurations that affect the way the component behaves. In the case of the tabs component the extra configurations can be provided as a JavaScript object to the jQuery UI tabs method. For example, developer may provide *active* option with the *id* attribute value of the tab content element to provide the default active tab or *event* option with name of the mouse event as value used to activate a tab for example click or double click. The configurations can be passed to the widget when initialized but also after initialization. (Wellman 2009, para. 4; jqueryui.com 2017)

UI components may also provide methods that perform actions on the component. For example, developer can disable all the tabs of the tabs widget by calling the *disable* method on the widget. Some of the methods may provide an alternative to using the configuration options, for example, tabs can also be disabled by the *disabled* option provided

with a value of an array of tab indexes that are to be disabled. (Wellman 2009, para. 6; jqueryui.com 2017)

UI components can send events on specific situations in response to user interaction. Developer can assign callback functions to these events to react to the situation that caused the event. For example, the tabs widget provides an *activate* event which fires when a tab is activated. The event object and other event related information may be given to the provided callback method as parameters. (Wellman 2009, para. 5; jqueryui.com 2017)

As stated in Section 2.4, it is possible for modern web applications to include their dependencies, including the UI components, as NPM packages. The NPM packages can then be used in the JavaScript files by utilizing the CommonJS or AMD.

3. SOFTWARE COMPONENT QUALITY EVALUATION

Software component quality and its evaluation has been researched in the context of component-based software development (CBSD) and component-based software engineering (CBSE). CBSD promotes software development by constructing the system by integrating existing software components in a well-defined architecture rather than implementing everything from scratch (Kaur et al. 2009). CBSD utilizes software components available in the market called commercial off the shelf (COTS) components which are often black boxed implying that the component source code is not available to the developers (Almeida & Calistru 2011). Thus, the developers must rely on documentation and testing to evaluate the component suitability to a project. Various quality models introduced by research and listed by Almeida & Calistru (2011) can be utilized in the black box assessment.

According to the work of Almeida & Calistru (2011) the research concerning the quality evaluation of components has been derived from the traditional software quality evaluation and is based mainly on *ISO 9126* (now replaced by *ISO 25010:2011*), which is discussed later in this chapter. The research has been focusing mostly on the demands the CBSD architecture sets on the components and the component selection process. Though, the point of view on the research is more on the side of the component consumer the results can also be utilized by the component developers to improve the component quality in the eyes of the component consumers.

There is a notable lack of research in the field of quality evaluation of web libraries and components, especially on the frontend side. Thus, in the context of this thesis the CBSD ideas are attempted to be applied to the web frontend development where often 3rd party libraries are utilized.

3.1 General software quality evaluation

According to the definition by IEEE (1990) software quality is:

1. *The degree to which a system, component, or process meets specified requirements.*
2. *The degree to which a system, component, or process meets customer or user needs or expectations.*

This definition leaves the essence of quality still quite abstract because the customer and user needs and expectations vary depending on the software project. Thus, there is also a

need for a quality model that attempts to describe the characteristics that form the software quality.

Firesmith (2005) presents software quality model as a hierarchical model that divides the quality into four levels with following descriptions:

- Quality factors (characteristics): high-level characteristics or attributes of a system that capture major aspects of its quality (e.g., performance or usability).
- Quality sub factors (sub-characteristics): major components of a quality factor or another quality sub factor that capture a subordinate aspect of the quality of a system (e.g., throughput or learnability).
- Quality criteria (attributes): specific descriptions of a system that provide evidence either for or against the existence of a specific quality factor or sub factor.
- Quality measures (metrics): gauges that quantify a quality criterion and thus make it measurable, objective, and unambiguous.

A quality evaluation framework ISO 9126 quality model introduces six quality factors or characteristics that can be used to evaluate software quality. The characteristics are *functionality, reliability, usability, efficiency, maintainability and portability* (O'Regan 2014). ISO 9126 has been replaced by ISO/IEC 25010:2011 which introduces product quality model with eight quality characteristics. These characteristics by ISO/IEC (2011) and their definitions can be seen on Table 3.1.

Table 3.1 ISO/IEC 25010:2011 Quality characteristics

Characteristic	Description
Functional suitability	Degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.
Performance efficiency	Performance relative to the amount of resources used under stated conditions.
Compatibility	Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment.
Usability	Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.
Reliability	Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
Security	Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
Maintainability	Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.
Portability	Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.

The characteristics introduced in Table 3.1 are divided into sub-characteristics that further describe the main characteristic. The characteristics are not measurable by themselves as Firesmith's (2005) quality model definition suggests, and related attributes as well as the measurements must be defined to measure the software quality. ISO/IEC (2011) defines the attributes as *inherent property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means*" and measurement as *"set of operations having the object of determining a value of a measure"*.

An example of a quality measurement presented in quality model by Alvaro et al. (2006) is shown in Table 3.2.

Table 3.2 Quality measurement example

Characteristic	Sub-characteristic	Attribute	Measurement
Performance efficiency	Time behaviour	Throughput	Amount of outputs produced with success / period of time

3.2 Overview of the CBSE and CBSD software component quality research

As discussed earlier in this chapter, the component quality evaluation has been researched in the context of CBSD and CBSE. Almeida & Calistru (2011) provide introduction to the CBSE concept in their paper and present some benefits and problems that the CBSE brings. According to their presentation the CBSE utilizes both the COTS components and components that have been developed inside the company.

Almeida & Calistru (2011) present that CBSE offers benefits in reduced time to market and reduced development costs. They also state that using components in different systems increases the component quality because they are tested continuously in different conditions. Furthermore, they present that standard domain will be formed when components are exposed to different environments within the domain. When the components are utilized in this standard domain they will become well-defined and can be reused more easily. Almeida & Calistru (2011) also present the similarity of CBSE to industrial product manufacturing where standard components or parts can be used in multiple products. Thus, standardized software components could also be used to produce multiple large systems efficiently.

However, Almeida & Calistru (2011) also point out difficulties concerning the CBSE approach. Component version compatibility can be difficult to manage because component versions may change independently of the system they are used in. Components may also be available in multiple different versions. Almeida & Calistru (2011) also present

that components need to be understood in order to utilize them in an application in contrast to a piece of software that the developers have programmed themselves. Component selection is also not a trivial task if there are multiple components that can fulfill the requirements. Developers have to weigh the components in accordance to the functional and non-functional requirements of the developed system. Component verification and validation is often difficult because the components are usually black-box and it has to be done mainly by examining the component specification or API. Almeida & Calistru (2011) also discuss that regression testing on component-based system may cause concerns if component testing information is not available. Thus, developers have to use their own judgement to determine the test cases that need to be run in order to complete a successful regression test run. Component-based system maintenance may also need extra attention if there are changes in component versions or availability.

Though, there are existing models to evaluate software quality such as ISO 9126, they are very general and not easily applied into software component quality evaluation (Bertoa et al. 2005). Almeida & Calistru (2011) list in their study various quality models developed to be used with CBSD. They present a comparison of the models in their paper and conclude with pointing out that models by Bertoa & Vallecillo (2002) and Alvaro et al. (2006) are the most cited and used ones. These two models are introduced later in this chapter. An earlier comparison committed by Kalaimagal & Srinivasan (2008) presents model by Alvaro et al. as the most consistent and suited for component quality evaluation. However, Kalaimagal & Srinivasan (2008) also point out that the proposed quality models including Alvaro's present too many quality attributes to be measured which may lead to confusion. Thus, it is suggested as future work that a comprehensive model with minimum amount of quality attributes would be presented to help standardizing the component quality evaluation (Kalaimagal & Srinivasan 2008).

3.2.1 Model by Bertoa & Vallecillo

The motivation for Bertoa & Vallecillo (2002) to develop their component quality model was that the existing quality models were too general to be applied to the specific aspects of software components. They also felt that the software engineering community had only focused on the functionalities of the components and ignored the quality aspects. Their quality model attempts to describe relevant quality attributes and measures to the software components using the ISO 9126 quality model as the base.

Model by Bertoa & Vallecillo (2002) divides the defined quality characteristics and attributes into two categories based on if they can be evaluated run time or by observing the component during its life cycle. Some of the characteristics presented in ISO 9126 were removed and some new ones were added. Additionally, Bertoa & Vallecillo also changed the definition of some characteristics to be more suitable when evaluating software component quality. The component quality characteristics are presented in Table 3.3.

Table 3.3 Component quality characteristics by Bertoa & Vallecillo

Characteristics	**Sub-characteristics (Runtime)**	**Sub-characteristics (Life-cycle)**
Functionality	Accuracy Security	Suitability Interoperability Compliance Compatibility
Reliability	Recoverability	Maturity
Usability		Learnability Understandability Operability Complexity
Efficiency	Time behavior Resource behavior	
Maintainability		Changeability Testability

The major changes regarding ISO 9126 according to Bertoa & Vallecillo (2002) are:

- Sub-characteristic *Compatibility* has been added to *Functionality* to assess if newer versions of the same component are compatible with older versions.
- *Usability* has been redefined to describe the component's ability to be utilized by developers when developing software instead of being related to the actual software end user usability.
- Sub-characteristic *Complexity* has been added to *Usability* to measure the difficulty to use the component or integrate it into a system.
- *Portability* characteristic is removed because it is assumed to be intrinsic to components.
- *Fault tolerance, stability and analyzability* sub-characteristics are removed because they are seen as not applicable to components.

Bertoa & Vallecillo (2002) define five measurement types that can be used to measure component attributes:

- **Integer:** Integer value that describes the number of measurable units for an attribute.
- **Presence**: Boolean value that indicates if attribute is present and string value that describes how the attribute is implemented.
- **Time**: Integer value that describes time with string value that describes units.
- **Level**: Integer value that describes effort or ability between 0-4 (Very low, Low, Medium, High, Very high).
- **Ratio**: Describes percentages by integer value between 0-100.

Additionally, the model by Bertoa & Vallecillo (2002) utilizes indices which are derived measures based on other existing metrics types. For example, the complexity ratio of a component is calculated by dividing the number of configurable parameters by the number of interfaces.

As mentioned earlier, the quality attributes are divided into two categories. The quality attributes that can be evaluated runtime are shown in Table 3.4.

Table 3.4 Component runtime quality attributes on Bertoa's & Vallecillo's model

Sub-characteristic	Attribute	Type
Accuracy	1. Precision	Ratio
	2. Computational Accuracy	Ratio
Security	3. Data Encryption	Presence
	4. Controllability	Presence
	5. Auditability	Presence
Recoverability	6. Serializable	Presence
	7. Persistent	Presence
	8. Transactional	Presence
	9. Error Handling	Presence
Time behavior	10. Response time	Time
	11. Throughput	Integer
	12. Capacity	Integer
Resource behavior	13. Memory utilization	Integer
	14. Disk utilization	Integer

The quality attributes that can be evaluated during component life cycle are presented in Table 3.5.

Table 3.5 Component life-cycle quality attributes on Bertoa's & Vallecillo's model

Sub-characteristic	Attribute	Type
Suitability	1. Coverage	Ratio
	2. Excess	Ratio
	3. Service Implementation Coverage	Ratio
Interoperability	4. Data Compatibility	Presence
Compliance	5. Standardization	Presence
	6. Certification	Presence
Compatibility	7. Backwards Compatibility	Presence
Maturity	8. Volatility	Time
	9. Evolvability	Integer
	10. Failure removal	Integer
Learnability	11. Time to use	Time
	12. Time to configure	Time
	13. Time to admin	Time
	14. Time to expertise	Time
Understandability	15. User Documentation	Level
	16. Help System	Level

Sub-characteristic	Attribute	Type
	17. Computer Documentation	Presence
	18. Training	Presence
	19. Demonstration Coverage	Ratio
Operability	20. Effort for operating	Level
	21. Tailorability	Level
	22. Administrability	Level
Complexity	23. Provided Interfaces	Integer
	24. Required Interfaces	Integer
	25. Complexity Ratio	Index
Changeability	26. Customizability	Integer
	27. Customizability Ratio	Index
	28. Change Control Capability	Level
Testability	29. Start-up self-test	Presence
	30. Tests suite provided	Presence

Bertoa & Vallecillo (2002) present descriptions for the attribute measurement in their paper and they are utilized in Section 4.3 where the measures for web frontend component quality model are defined.

3.2.2 Model by Alvaro et al.

The motivation for Alvaro et al. (2006) to present their component quality model was to attend to the problems that utilizing low-quality components can cause to the software quality. The aim of the model is to improve the reliability of the components available in the market and thus, support the emergence of a mature software component market.

Alvaro et al. (2006) also conducted a preliminary study on two separate component markets to assess the viability of the presented quality model. The study found out that the model covered all the information that was available at the component markets they conducted their study in. Additionally, the study reached a conclusion that there is still an existing gap between the information provided by the component providers and the information required by the quality model.

Like Bertoa & Vallecillo (2002), Alvaro et al. (2006) present their model based on ISO 9126. Alvaro et al. also utilize the earlier work by Bertoa & Vallecillo (2002) as well as other research. The researchers introduce all six main quality characteristics that can be found in ISO 9126 but also include a new characteristic: marketability. The sub-characteristics are also divided to runtime and life-cycle categories. Component quality characteristics by Alvaro et al. (2006) are presented in Table 3.6.

Table 3.6 Component quality characteristics by Alvaro et al.

Characteristics	**Sub-characteristics (Runtime)**	**Sub-characteristics (Life-cycle)**
Functionality	Accuracy Security	Suitability Interoperability Compliance Self-contained
Reliability	Fault tolerance Recoverability	Maturity
Usability	Configurability	Understandability Learnability Operability
Efficiency	Time behavior Resource behavior Scalability	
Maintainability	Stability	Changeability Testability
Portability	Deployability	Replaceability Adaptability Reusability
Marketability	Development time Cost Time to market Targeted market Affordability	

The most notable changes compared to ISO 9126 in model according to Alvaro et al. (2006) are:

- Sub-characteristic *self-contained* has been added to *functionality* because the researchers find it important that component performs its functionality within itself without dependencies to other components.
- Sub-characteristic *configurability* has been added to *usability* because it is essential for developers to be able to determine how easy it is to configure the component to be used in a certain context.
- Sub-characteristic *scalability* has been added to *efficiency* because it is important for developers to know if the component can support the data volumes needed by the application.
- *Analyzability* sub-characteristic has been removed from *maintainability* because according to research the analyzability characteristics are not present in components.
- Sub-characteristic *reusability* has been added to *portability* because reusability is one of the main motivations for component based development.

- The inclusion of *marketability* characteristics motivated by providing information to developers to assess the credibility of the component provider.
- Redefinition of *usability* characteristic as component's ability to be utilized by developers similar to model by Bertoa & Vallecillo (2002).
- Sub-characteristic *installability* has been renamed to *deployability* in portability to better reflect the component context.

In their model Alvaro et al. (2006) define three measure types to be used when measuring quality attributes. They have same definitions as in model by Bertoa & Vallecillo (2002) described in chapter 3.2.1. The measure types are *presence*, *integer* and *ratio*.

Alvaro et al. (2006) provide listings for their quality attributes and the measure types used to determine values for the attributes. The attributes for runtime evaluation are shown in Table 3.7.

Table 3.7 Component runtime quality attributes on Alvaro's model

Sub-characteristic	**Attribute**	**Type**
Accuracy	1. Correctness	Ratio
Security	2. Data Encryption 3. Controllability 4. Auditability	Presence Ratio Presence
Recoverability	5. Error Handling	Presence
Fault Tolerance	6. Mechanism available 7. Mechanism efficiency	Presence Ratio
Configurability	8. Effort to configure	Integer
Time behavior	9. Response time 10. Latency and processing capacity	Integer Integer
Resource behavior	11. Memory usage 12. Disk usage	Integer Integer
Scalability	13. Processing capacity	Ratio
Stability	14. Modifiability	Ratio
Deployability	15. Complexity level	Integer

The quality attributes that need to be measured during component life cycle are presented in Table 3.8.

Table 3.8 Component life-cycle quality attributes on Alvaro's model

Sub-characteristic	**Attribute**	**Type**
Suitability	1. Coverage	Ratio
	2. Completeness	Ratio
	3. Pre-conditioned and Post-conditioned	Presence
	4. Proofs of pre-conditions and post-conditions	Presence
Interoperability	5. Data Compatibility	Presence
Compliance	6. Standardization	Presence
	7. Certification	Presence
Self-contained	8. Dependability	Ratio
Maturity	9. Volatility	Integer
	10. Failure removal	Integer
Understandability	11. Documentation available	Presence
	12. Documentation quality	Ratio
Learnability	13. Time and effort to (use, configure, admin and expertise) the component.	Integer
Operability	14. Complexity level	Ratio
	15. Provided Interfaces	Integer
	16. Required Interfaces	Integer
	17. Effort for operating	Presence
Changeability	18. Extensibility	Ratio
	19. Customizability	Presence
Testability	20. Test suite provided	Presence
	21. Extensive component test cases	Presence
	22. Component tests in a specific environment	Presence
	23. Proofs the components	Presence

Alvaro et al. present some insight on how to measure the attributes on their model (2005, 2006) but these are used later on Section 4.3 when the web frontend component quality model is presented.

4. DERIVING A MODEL FOR WEB FRONTEND COMPONENT QUALITY EVALUATION

The previous chapter discussed general component quality and component quality models as they have been researched in the context of CBSD and CBSE. The research has been focusing on black box COTS components. The aim of this chapter is to derive a quality model that can be used to evaluate web frontend component quality by utilizing the ideas and results from CBSD quality research. Thus, the CBSD quality research and the quality models it has produced, as well as the web development context information provided in the earlier chapters of this thesis, are used as a base for the model.

The aim for the model is to be simple to use and provide relevant information in the context of web frontend components. Therefore, some initial requirements are set for the model:

1. Focus on quality attributes that are important for web components.
2. Model is to be kept simple by avoiding too many quality attributes.
3. Select quality attributes that can be measured with adequate precision.

4.1 Web frontend component characteristics

First, to be able to construct a quality model for web front end components, an identification of frontend component specific characteristics is required. These characteristics will be extracted from the earlier chapters of this thesis where web frontend development is discussed.

The simple consumption of web frontend components is important. As stated in Section 2.4, there are multiple methods for utilizing JavaScript libraries: AMD, CommonJS, ES6 modules and through HTML script tags. It is important for web frontend components to support these different kinds of utilization methods to allow wide range of developers to access the component.

Simple installation and dependency management is a characteristic for modern web frontend components. This is achieved by utilizing for example, NPM as introduced in Section 2.4. Component distribution through a package manager will reduce the need for developers to manually handle the component files. Package managers also store the installed components and their dependencies in a specific folder in the project which separates project specific code from the code required by the components.

Web frontend components can be used in a wide variety of projects and therefore the ability to customize their appearance is important. Web-based components may support

the modification of their appearance through CSS as described in Section 2.5 because their presentation consists of HTML. Developers utilizing the component have to take into account that the styling of complex components may also affect their functionality.

Section 2.3 provided a brief introduction to frontend frameworks. Usually when web frontend components are used with a framework, developers wrap them inside framework supported module implementations for example, React components or AngularJS directives. However, some web frontend components may provide different implementations in specific framework formats that work out of the box, which eliminates some of the work required by the developers to adopt the component in their project.

Web frontend UI components were discussed in Section 2.5. There it was pointed out that many components support configurations, methods and events that can be utilized by the developers to customize the component and interact with the component to suit the needs of the project.

Developers have to pay attention to browser compatibilities of web frontend UI components, too. Some components may utilize technologies not supported by certain browser types and versions or may function unexpectedly in different browsers.

4.2 Web frontend component quality model

The high-level quality characteristics are selected as a first step for developing a frontend component quality model. The following listing shows the union of the characteristics provided by the models introduced in Section 3.2. The listing also shows the reasoning for including or eliminating certain characteristic from the web frontend component quality model.

Functionality is included in the model because it contains many relevant sub-characteristics for web frontend components concerning compatibility and features required by web applications.

Reliability is included in the model since components are usually used as a part of a larger system or interaction sequence where fault in a single component may interrupt or corrupt the whole process.

Usability is included in the model because it can be considered as the most important quality characteristic of a web frontend component. If a component cannot be understood and utilized by the developers, it will not be used regardless of the functionalities it might have.

Efficiency is included in the model because components may need to comply to heavy data load on certain use cases and suboptimal efficiency solutions in components can cause problems in the overall application level.

Maintainability is included in the model because it contains some relevant sub-characteristics for web frontend components regarding component testing. Otherwise web frontend components themselves do not usually require maintenance other than version updates.

Portability is included in the model because it is important that web frontend components can be deployed to different kinds of environments utilizing varying frontend frameworks.

Marketability is eliminated from the model because it does not provide any characteristics that would directly affect the web frontend component quality. The marketability describes more the component development process and the component affordability rather than the actual component quality.

Next, the relevant sub-characteristics for each characteristic will be selected with the same procedure as previously from the union of sub-characteristics provided for each characteristic by the models introduced in Section 3.2.

4.2.1 Functionality

Accuracy is eliminated from the model because its measurement is a statistical analysis that will require a great amount of effort and preparation. The measurement would be too heavy process for a model that is intended to be simple. It can also be argued that operational accuracy is at least partly covered by the testability sub-characteristic that is included in the model.

Security is eliminated from the model because security in web applications is handled in the backend. Frontend security measures such as validations can always be ignored by malicious users.

Suitability is included in the model because it tries to evaluate the suitability of the component for specific requirements which is essential for developers when selecting a web frontend component to a project.

Interoperability is eliminated from the model because it can be assumed that every frontend component supports JSON format which is standard for modern web applications.

Compliance is eliminated from the model because even if component is complying to certain standards or is certificated, they do not measure the component quality by themselves. It can be assumed that standards and certificates are the result of good component quality which can be covered by other characteristics.

Self-contained is included in the model because it was specifically introduced by Alvaro's model as a sub-characteristic that applies to components. Self-contained can be seen as important characteristic to web frontend components because it measures the component dependency to other components and functionality which inevitably also affects the complexity of use of the component.

Compatibility is included in the model because it measures the backwards compatibility of the different versions of components. This is important characteristic considering web applications because web applications usually consist of multiple components and their interactions. Therefore, issues with compatibility may cause a requirement for extensive changes to the application.

4.2.2 Reliability

Fault tolerance is included in the model because sometimes it is important that a component can function even sub optimally if it receives invalid values.

Recoverability is included in the model because if a component has a state, it is important that it can be serialized to be able to load the component with an existing state. Furthermore, error handling is important for web frontend components for example in the case of providing invalid configurations or invalid method calls.

Maturity is eliminated from the model because it does not provide any direct quality information about the current situation of the component. This sub-characteristic provides more an indication of the possible quality and can be used when selecting components but has little use when trying to improve component quality.

4.2.3 Usability

Configurability is included in the model because for web frontend components, it is important that they can be configured run time to cover different situations.

Understandability is included in the model because developers need to understand the component to be able to use it effectively.

Learnability is eliminated from the model because it is difficult to measure component learnability without conducting an extensive research. In order to keep the model simple, this sub-characteristic is eliminated.

Operability is eliminated from the model because it partly overlaps with the self-contained sub-characteristic from functionality. It also has difficult-to-measure attributes.

Complexity is eliminated from the model because it overlaps with the self-contained sub-characteristic from functionality.

4.2.4 Efficiency

Time behavior is eliminated from the model because while it is important that component operates efficiently regarding response time, its planning and measurement does not fit well into the simplicity requirements of the model. Time behavior also partly overlaps with the scalability attribute that is included in the model.

Resource behavior is included in the model because inefficient memory utilization of a component may slow the web browser down and interfere with the whole system. Additionally, the disk space that the component requires affects the application loading times because browser has to load the component files.

Scalability is included in the model because in web development, components may be utilized in different kinds of projects with varying data intensity. Therefore, it is important that a web component can operate at trivial and very high data volumes.

4.2.5 Maintainability

Stability is eliminated from the model because it only measures the ability of a component to support modifications. This is not a relevant use case to web frontend components because they are modified by configuring them run time to suit the needs of the project, not by modifying their source code.

Changeability is eliminated from the model because of the web component characteristics as mentioned in stability. Configurability sub-characteristic from usability is the similar characteristic that will be used in the model to measure the component run time configurability.

Testability is included in the model because it is important that the component provides some proof that it can be tested and that it has been tested. It can also indicate that the component can be safely utilized in production because its core functionality has been proven to be working.

4.2.6 Portability

Deployability is included in the model because simple deployment is important for web frontend components.

Replaceability is eliminated from the model because it fully overlaps with compatibility sub-characteristic from functionality.

Adaptability is eliminated from the model because web frontend components are not usually transferred from environments to another. It could be argued that transferring

component from frontend framework to another can be seen as transfer to another environment but it is a rare occurrence and not directly related to the web frontend component because the web frontend component is usually wrapped in framework specific component code.

Reusability is included in the model because it contains relevant attributes to web frontend components. For example, modularity, which affects the fact if the component parts can be utilized individually or always as a whole entity and if the component allows developers to only load the required parts of the whole component.

4.2.7 Summary of selected characteristics

The characteristics and their sub-characteristics that are selected to the web frontend component quality model are shown in Table 4.1. The *deployability* sub-characteristic in *portability* is renamed back to its original name *installability* because it is descriptive in web frontend component context. In addition to changing name, installability is also transferred to life-cycle sub-characteristics because the component run-time deployment procedures are handled with the *configurability* sub-characteristic in *usability*.

Browser compatibility is added as a new runtime sub-characteristic to *functionality* because it is important for web application components to function properly with different browsers. Good browser compatibility is essential for a web frontend component to be usable in a wide range of applications.

The *compatibility* sub-characteristic in functionality is renamed to *backwards compatibility* to make a clear distinction from the browser compatibility. The backwards compatibility is also transferred to runtime sub-characteristic because web frontend components are JavaScript-based and due to its interpretative nature, the compatibility errors can only be detected when running the application.

Table 4.1 Web frontend component quality model characteristics

Characteristics	Sub-characteristics (Runtime)	Sub-characteristics (Life-cycle)
Functionality	Browser compatibility Backwards compatibility	Suitability Self-contained
Reliability	Fault tolerance Recoverability	
Usability	Configurability	Understandability
Efficiency	Resource behavior Scalability	
Maintainability		Testability
Portability		Installability Reusability

It can be seen in Table 4.1 that in the web frontend quality model only usability and functionality characteristics have sub-characteristics on both the runtime and life-cycle categories while originally every characteristic, apart from efficiency, has sub-characteristics in both categories. This is not necessarily a problem if the nature of each of these characteristics is briefly contemplated:

Reliability: How a component reacts to error situations can be evaluated with precision only when the application is running. Therefore, no life-cycle sub-characteristics are required.

Efficiency: Directly linked to the times when the application is running because that is when the efficiency can be evaluated. Therefore, no life-cycle sub-characteristics are required.

Maintainability: Changes to the code of the application are generally done outside of the running application. Therefore, no runtime sub-characteristics are required.

Portability: When application is installed on an environment it is not running. Therefore, no runtime sub-characteristics are required.

4.3 Attributes and measures for the selected quality characteristics

To proceed with the web frontend component quality model, it is necessary to select attributes for the quality sub-characteristics in the model. The measures for evaluating the attributes are also to be presented. The ideas introduced in the models in Section 3.2 are utilized when developing the measures for the web frontend component quality model. For example, the measure types similar to the presented models are utilized with the exception that ratio is presented as a decimal number between 0 and 1 instead of an integer between 0 and 100. Additionally, the attribute definitions for the quality sub-characteristics that are presented by Alvaro et al. (2005) and Bertoa & Vallecillo (2002) are utilized when selecting relevant attributes and measures for the web frontend component quality model sub-characteristics.

4.3.1 Functionality

Functionality consists of four sub-characteristics. This subsection presents their attributes and the measurements for them.

Browser compatibility

For the needs of this model, browser compatibility is divided into two attributes: *cross-browser functionality* and *browser support*. Cross-browser functionality tries to measure

the ability of the component to provide consistent functionality when used with different browsers and their different versions. On the other hand, browser support measures the ability of the component to be utilized on a sufficient level in an application that is run on a certain browser version. The measures for browser compatibility attributes are presented in Table 4.2.

Table 4.2 Browser compatibility measures

Attribute	Type	Measure
Cross-browser functionality	Ratio	First, to evaluate the cross-browser functionality of a component, the evaluator selects the browsers and their specific versions to be used in the evaluation. The component features to be evaluated are also selected. One browser and its specific version is selected as a reference point for correct functionality of the features of the component. The features of the component in other browsers and their versions are compared to the component features in the reference point browser. The result is provided as a ratio of how many of the component features in other browsers match the reference point.
Browser support	Ratio	This attribute provides two alternative methods of measurement: 1. If component provides documentation or testing documentation that states supported browser versions they may be utilized. 2. Component is run manually as part of an application on different browser versions and results are observed. The result is provided as a ratio of how many required browser versions are supported.

It is notable that the attributes presented here for browser compatibility may be extremely time consuming to measure for all features of the component if the component does not provide relevant documentation. Therefore, the individual responsible for the measurement has to use discretion when selecting the most important features of the component that will be tested for browser compatibility.

Backwards compatibility

The sub-characteristic backwards compatibility is divided into two attributes: *data compatibility* and *functional compatibility*. Data compatibility is utilized with components that can serialize their state for persistent storage. It is measuring the ability of the component to support previous data formats if they have been stored to the backend by the application. Functional compatibility is used to measure the compatibility of the component provided methods and configurations in a way that existing implementations utilizing the component would not stop operating correctly after component version updates. The measures for backwards compatibility attributes are presented in Table 4.3.

Table 4.3 Backwards compatibility measures

Attribute	Type	Measure
Data compatibility	Presence	This attribute is measured by verifying if the attribute is present on the component. This may be achieved by finding such statement from the component documentation or by manually verifying the data compatibility when updating the component version.
Functional compatibility	Presence	This attribute is measured with the same procedure as the data compatibility.

Ideally the statements needed by the backwards compatibility attributes are satisfied by the component documentation. Otherwise, the individual responsible for the measurement has to go through multiple versions of the component to get an indication if the attributes are present on the component.

Suitability

In this model, component suitability is divided into three attributes: *coverage*, *excess* and *completeness* (Bertoa & Vallecillo 2002; Alvaro et al. 2005). Coverage measures how many of the required functionalities are implemented by the component. Excess measures how many of the implemented functionalities of the component are not required by the application. Completeness measures how many of the specified functionalities are implemented by the component. The measures for suitability attributes are presented in Table 4.4.

Table 4.4 Suitability measures

Attribute	Type	Measure
Coverage	Ratio	Measured by dividing the number of modules in the component that implement some of the functionality required by the application with the number of modules required by the application.
Excess	Ratio	Measured by dividing the number of modules in the component not utilized by the application with the number of modules provided by the component.
Completeness	Ratio	Measured by dividing the number of modules provided by the component with the number of modules specified by the component documentation.

In the context of web frontend components, the term *module* used in the suitability measures is referring to a relevant JavaScript object or function that provides services from the component or library that the application can utilize. The individual responsible for the measurement has to use her judgement to identify these modules usually with the aid of component documentation. Nevertheless, as can be seen from the descriptions in Table 4.4 the suitability measurements are relatively easy to execute if the requirements for the application are clear and the relevant documentation is available.

Self-contained

Self-contained is measured by *dependencies* attribute originally named dependability (Alvaro et al. 2005). It tries to evaluate if the component requires some external implementations to provide its own services. Originally in Alvaro's model the dependability is intended to measure the ratio of functionalities provided by the component but implemented by its dependency to the total functionalities provided by the component. For the needs of this model, the dependencies attribute is interpreted as an attribute more similar to the *required interfaces* in the *operability* sub-characteristic presented by Alvaro et al. (2006). The measures for self-contained attributes are presented in Table 4.5.

Table 4.5 Self-contained measures

Attribute	Type	Measure
Dependencies	Integer	Measured by calculating the number of required modules by the component.

The modules calculated in the measurement for dependencies are the parameters that are implemented outside of the component that are required by the modules that the component provides to operate correctly.

4.3.2 Reliability

Reliability consists of two sub-characteristics. This subsection presents their attributes and the measurements for them.

Fault tolerance

Fault tolerance is divided into two attributes: *mechanism available* and *mechanism efficiency* (Alvaro et al. 2005). Mechanism available evaluates if the modules provided by the component have implemented fault tolerance mechanisms. Mechanism efficiency tries to evaluate how effectively the mechanisms work in preventing the effects of errors. The measures are modified from the suggestions by Alvaro et al. (2005). The measures for fault tolerance attributes are presented in Table 4.6.

Table 4.6 Fault tolerance measures

Attribute	Type	Measure
Mechanism available	Ratio	Measured by dividing the number of modules provided by the component that implement a fault tolerance mechanism with the total number of modules provided by the component.
Mechanism efficiency	Level	Measured on a general level for the whole component with a value between 0-4.

In JavaScript-based web frontend components, the fault tolerance mechanisms may for example, handle missing or incorrect type parameters for object constructors or methods by utilizing default values and informing the developer with a console message. Because JavaScript is interpreted, most of the errors are noticed runtime, for example, at a point when code tries to execute a missing callback function. For fault tolerance, it is important that the component handles errors and exceptions that are not preventing it from operating so that the whole application is not crashing.

Recoverability

Recoverability is divided into two attributes: *serializable* and *transactional* (Bertoa & Vallecillo 2002). Bertoa's model originally also has error handling attribute but it is similar to fault tolerance sub-characteristic, so it will not be included in this model. Additionally, the persistent attribute is not used in this model, because it is not relevant to web frontend components. This is caused by the fact that the web frontend does not support storage other than cookies and local storage of the browser that are not persistent.

Serializable attribute evaluates if the component state is serializable in a way that it can be initialized again directly in that state. Transactional attribute measures if the component has a mechanism for transactions. The measures for recoverability attributes are presented in Table 4.7.

Table 4.7 Recoverability measures

Attribute	Type	Measure
Serializable	Presence	Measured by evaluating if the component has a mechanism for serializing its state for example, to a JSON format that can be loaded later on the component.
Transactional	Presence	Measured by evaluating if the component implements transactions in a way that allows to monitor and modify the history of actions done to the component.

4.3.3 Usability

Usability consists of two sub-characteristics. This sucsection presents their attributes and the measurements for them.

Configurability

The configurability measures provided by the model by Alvaro et al. (2005) are not sufficient and detailed enough and thus, will not be used in the web frontend component quality model. The information provided about web frontend UI components introduced in Section 2.5 will be utilized to present the attributes. Consequently, for the purposes of this model configurability is divided into three attributes: *functional configurability, event*

configurability and *appearance configurability*. Functional configurability tries to evaluate the level that the component allows developers to configure its functionality by modifying the component configurations on component initialization or after the component is initialized. Event configurability measures the level of events that are available by the component to allow the rest of the application to react to the component actions. Appearance configurability attempts to measure the developers' possibilities to affect the appearance of the component. The measures for configurability attributes are presented in Table 4.8.

Table 4.8 Configurability measures

Attribute	Type	Measure
Functional configurability	Level	Measured with a value between 0-4. The component modules are analyzed to see if they allow initialization parameters or provide a way to modify the configurations after the module is initialized.
Event configurability	Level	Measured with a value between 0-4. The component modules are analyzed to see if they provide relevant events that the application can apply callbacks to.
Appearance configurability	Level	Measured with a value between 0-4. The effort for modifying the appearance of the component is evaluated. For example, are there configurations for modifying the component appearance with themes or do the developers need to implement component appearance modification with custom CSS. Additionally, is there documentation available that would help with modifying the component appearance

For configurability, the component documentation is invaluable in providing the information that is required by the individual responsible of the measurement.

Understandability

Understandability is divided into three attributes: *documentation coverage, documentation quality* and *demonstration coverage*. There are more attributes introduced by Bertoa & Vallecillo (2002) such as *help system*, *computer documentation* and *training* but they are discarded as these are not common with web frontend components. Documentation coverage measures the coverage of functionalities that are presented in the documentation available to the user. Documentation quality attempts to evaluate the usefulness and clarity of the documentation and additionally, the aspects of the computer documentation in a sense that is the documentation available for example, in HTML format. Demonstration coverage tries to measure the coverage of functionalities that are shown in tutorials and demonstrations of the component. The measures for understandability attributes are presented in Table 4.9.

Table 4.9 Understandability measures

Attribute	Type	Measure
Documentation coverage	Ratio	Measured by dividing the number of functionalities documented with the total number of functionalities in the component. Functionalities may include installation, configuration options, methods, events and appearance configurations.
Documentation quality	Level	Measured with a value between 0-4. Measured qualities may include for example, documentation readability and code examples.
Demonstration coverage	Ratio	Measured by dividing the number of functionalities shown in demonstrations with the total number of functionalities in the component. The total number of functionalities may be extracted from the documentation available. Functionalities may include installation, configuration options, methods, events and appearance configurations.

Documentation coverage may be difficult to evaluate absolutely if the component is not known to the individual responsible for the measurement. However, the general guidelines for functionalities presented in the measure can be used to identify the relevant documentation items.

4.3.4 Efficiency

Efficiency consists of two sub-characteristics. This subsection presents their attributes and the measurements for them.

Resource behavior

Resource behavior is divided into two attributes: *memory utilization* and *disk utilization* (Bertoa & Vallecillo 2002; Alvaro et al. 2005). Memory utilization measures the memory needed by the component to operate. Disk utilization measures the space the component files take from the disk. The measures for resource behavior attributes are presented in Table 4.10.

Table 4.10 Resource behavior measures

Attribute	Type	Measure
Memory utilization	Integer	Measured by evaluating the memory utilization of the component when used in a minimal application.
Disk utilization	Integer	Measured by calculating the total disk space required by the files in the component.

Scalability

Scalability is measured by *processing capacity* attribute (Alvaro et al. 2005) that tries to evaluate the ability of the component to handle large data volumes. In this model, the processing capacity is interpreted similar to the latency attribute of time behavior sub-characteristic in model by Alvaro et al. (2005). The measures for scalability attributes are presented in Table 4.11.

Table 4.11 Scalability measures

Attribute	Type	Measure
Processing capacity	Integer	Measured by testing the component scalability when exposed to different amounts of data by dividing the number of data units processed with the time required to process the data. Exact measurement needs to be specified component specifically.

Scalability measurement is affected greatly by the component implementation. For example, if the component accepts static data in array format as its input the measurement is easier to conduct than if the component data is fetched from API or if the data is generated by some other measures.

4.3.5 Maintainability

Maintainability consists of one sub-characteristic. This subsection presents its attributes and the measurements for it.

Testability

Testability is divided into two attributes: *test suite provided* and *tests in a specific environment*. Bertoa & Vallecillo (2002) and Alvaro et al. (2005) provide more attributes in their models such as proofs of the component and start-up self-test but these are not specifically relevant for web frontend components. Test suite provided measures if the component package contains tests that can be ran by the developers who are utilizing the component. Tests in a specific environment evaluates if the component test documentation contains information about if the component has been tested in different environments for example, with different browsers or with different JavaScript frameworks. The measures for testability attributes are presented in Table 4.12.

Table 4.12 Testability measures

Attribute	Type	Measure
Test suite provided	Presence	Measured by verifying if the component package includes a runnable test suite.
Tests in a specific environment	Presence	Measured by verifying if the component testing documentation contains information about component tests done on different browsers or JavaScript frameworks.

4.3.6 Portability

Portability consists of two sub-characteristics. This subsection presents their attributes and the measurements for them.

Installability

Installability is divided into four attributes: *framework support, loading type coverage, installation simplicity* and *application to DOM*. Model by Alvaro et al. (2005) does not provide any relevant information on how to evaluate installability and thus, these attributes are utilized based on web frontend information presented earlier in this thesis. Framework support evaluates if the component is available as framework supported components, for example, as React component or AngularJS directive. Loading type coverage tries to evaluate if the component supports different types of module loading including script tag, AMD and CommonJS. Installation simplicity attempts to measure the effort required to install and update the component package. Application to DOM evaluates the options and complexity the component presents for applying the component to DOM in HTML containers. The measures for installability attributes are presented in Table 4.13.

Table 4.13 Installability measures

Attribute	Type	Measure
Framework support	Ratio	Measured by dividing the number of supported framework specific versions of the component required by developers by the required number of framework supports of the component.
Loading type coverage	Ratio	Measured as the ratio of module loading types supported by the component to the number of required loading types.
Installation simplicity	Level	Measured with a value between 0-4. Measured by evaluating the effort required to install and update the component package on the project.
Application to DOM	Level	Measured with a value between 0-4. Measured by evaluating the simplicity of the required HTML markup for the component to be applied to DOM.

Component installation and update complexity depends heavily on the package installation design. For example, if it is NPM based it is possibly very simple. On the other hand, if no package manager support is provided the files have to be moved manually.

Reusability

Reusability is divided into three attributes: *modularity, coupling* and *architecture compatibility*. Alvaro et al. (2006) also present other attributes for example, domain abstraction level and cohesion but they are discarded as they do not bring any additional value to the model. Modularity tries to measure the ratio of modules to different functionalities in the component. Coupling evaluates the dependencies between the different modules in the component for example, can they be utilized separately. Architecture compatibility evaluates if the component is dependent on specific architecture types or forces the developers to use certain architecture in the application when utilizing the component. The measures for reusability attributes are presented in Table 4.14.

Table 4.14 Reusability measures

Attribute	Type	Measure
Modularity	Ratio	Measured as the ratio of modules provided by the component to the functionalities provided by the component.
Coupling	Ratio	Measured as the ratio of how many modules provided by the component require at least one another module provided by the component to function properly.
Architecture compatibility	Level	Measured with a value between 0-4. Measured by analyzing how the component usage affects the supporting code structure implemented in the application because of the component.

4.3.7 Summary of attributes and measures

The attributes and measures presented in the subsections above provide the lowest level of the web frontend component quality model. Total of 30 attributes were selected to the model. When the model is utilized to measure web frontend component quality a value is evaluated for each attribute through its respective measure.

It can be argued that the ratio, presence and integer type measures are objective because they are based on numeric and logical evaluation. On the other hand, the level measure type is more subjective because the evaluator evaluates the level based on her expertise. Out of the 30 attributes 8 are of level type.

4.4 Summary of the web frontend component quality model

The quality measurement in accordance to the model presented in this chapter requires quite comprehensive expertise of software development from the person responsible of

the evaluation. Many of the measures require the evaluator to identify certain structures such as modules from the web frontend component which may not always be simple especially if the evaluator tries to evaluate the quality of a component she is not familiar with.

However, the aim of the model in the context of this thesis is to evaluate components that are familiar to the evaluator in order to find means to increase their quality in the eyes of developers that could utilize them in their projects. The model may also have additional uses as a reference when developing a new component in identifying important points that need to be considered to make the component appealing.

Some of the attributes that the model presents depend on the requirements of the application where the component will be used, for example, the cross-browser functionality and suitability coverage. If the evaluator is trying to improve the quality of her own developed component the perceived target project requirements have to be generated if no reference project is available.

Many of the attributes in the model try to find data for their evaluation from the component documentation. This implies that a high-quality component also provides its documentation in high quality. Of course, to contain information of, for example, browser compatibility, it has to be actually verified before it can be written into the documentation. It could be argued that according to the presented web frontend component quality model, high quality documentation is a way of expressing the overall component quality to potential developers.

Finally, the requirements set for the model at the start of this chapter are briefly evaluated. The first requirement was to focus on quality attributes that are important for web components. This requirement was attempted to be met by introducing attributes specific to web applications such as browser compatibility and loading type coverage.

The second requirement was to keep the model simple by avoiding too many quality attributes. This requirement was attempted to be met by reducing the amount of quality attributes by eliminating irrelevant or redundant sub-characteristics presented by the CBSD quality models. The current number of 30 attributes is still relatively high but also lower than the 44 and 37 attributes that are introduced by models by Bertoa & Vallecillo (2002) and Alvaro et al. (2006), respectively. The number of attributes was also increased by the fact that the web frontend component quality model introduced several attributes that are web component specific.

The third requirement was to select quality attributes that can be measured with adequate precision. This is evaluated properly after the web frontend component model has been utilized in Chapter 5.5. Now, it can be stated that the model tried to define all the attributes in a way that contains a measure that can be used to evaluate the attribute with sufficient precision.

The full web frontend component model derived in this chapter is in Appendix A.

5. QUALITY EVALUATION WITH THE WEB FRONTEND COMPONENT QUALITY MODEL

In this chapter, the web frontend component quality model is utilized to evaluate the report editor component extracted from the IoT-Ticket product developed by Wapice Ltd. The report editor component package can be extracted for the needs of other projects through a build script utilizing Grunt task runner. The report editor component has been utilized in a single project so far and it is still quite primitive which offers a relevant opportunity for testing the web frontend component quality model to provide quality improvement suggestions. Thus, the model is utilized to evaluate the quality of the report editor component from the developers' point of view.

As a component, the report editor is larger and more complex than most of the web frontend components. It can be seen as almost its own frontend application. However, it can still be utilized as a component in an application as an interactive designer of data visualizing reports. Therefore, the model can be utilized to evaluate the quality of the report editor.

5.1 Overview of the IoT-Ticket platform

IoT-Ticket (iot-ticket.com 2017) is a complete internet of things (IoT) platform which covers data acquisition, dashboard, reporting and analytics features. Data acquisition can be done by utilizing electronics that send data to IoT-Ticket big data server or by software that connects to the IoT-Ticket API. Wapice also provides WRM247+ device that can be used to acquire data for the IoT-Ticket.

Users can create interactive dashboards and reports based on the data that has been acquired by the devices by using web based user interface of the IoT-Ticket. Dashboards are designed with interface designer and dataflow editor. Reports are designed with report editor and dataflow editor.

The interface designer is a graphical tool for designing dashboards and it offers numerous widgets including charts, tables, gauges and many others that can be used to visualize the data acquired by the devices connected to the IoT-Ticket. The widgets can be positioned on the dashboard as desired by the user. Data collected by the devices is available in the interface designer though data tags which can be dragged and dropped on the widgets to connect their data to be visualized by the widget. Image of a dashboard that has been designed with the interface designer can be seen in Figure 5.1.

Figure 5.1 IoT-Ticket dashboard

The dataflow editor is a graphical programming editor that allows users to design complex logic and control and modify the data that has been acquired by the devices. The dataflow editor contains for example, mathematical and logical operations and timers. The dataflow editor also controls events that can be used to trigger certain operations for example to reset a counter visualized by a gauge component after a certain time.

The dataflow is edited by connecting dataflow blocks to each other. Each dataflow block has different kinds of inputs and outputs that can be connected to other blocks to affect the functionality of individual blocks and also the whole dataflow. As mentioned earlier the dataflow blocks can represent mathematical or logical operations but each widget that is present in the interface designer is also used as part of the dataflow with their own blocks. Therefore, the data displayed by the widgets is not limited to the raw data from the data tags but the data can also be modified by the functionality provided by the dataflow editor and its dataflow blocks.

The dataflow editor provides also an execution mode selection. The dashboards can be executed in client mode or server mode. Client mode implies that the dashboard dataflow is only evaluated when it is opened on a client. Server mode implies that the dashboard dataflow is ran continuously on the server even when no clients have it opened. Server mode enables that the events are triggered even when the dashboard is not open on a client for example, sending a report email through a report dataflow block. Image of the dataflow editor is available on Figure 5.2.

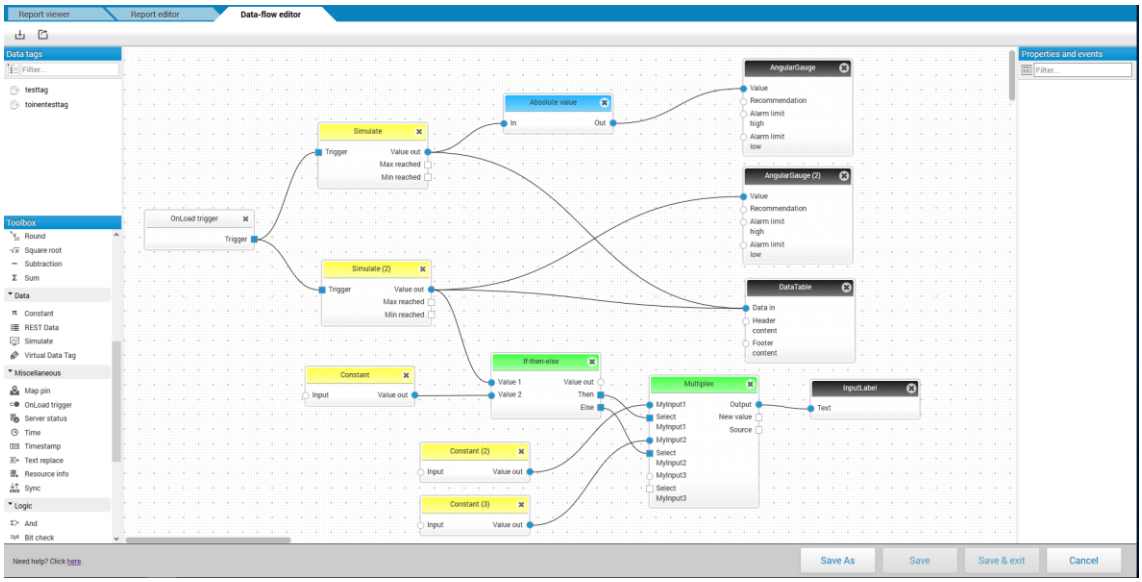

Figure 5.2 IoT-Ticket dataflow editor

The report editor works similarly to the interface designer but it also contains common features from word processing software. It allows users to design a multi-page document with headers and footers and such as in any word processing software but it also includes the IoT-Ticket widgets and dataflow processing features.

The report editor consists of four views: report viewer, report editor, dataflow editor and view for report pdf generation. Report viewer allows the user to view the report document but not make any changes to it similar to a print preview view on a word processing software. Report editor allows users to modify the report by dragging widgets to it and writing text content. The dataflow editor provides the users the ability to modify the logic that provides the data for the widgets. The view for pdf generation is not accessed by regular users but the server is using that to generate pdf documents of the reports. The pdf generation view is a simplified version of the report viewer that only renders the report pages one after another in a way that a pdf document can be constructed. Image of the report editor is presented in Figure 5.3.

The difference of reports to the dashboards is that the timers and some of the interactive widgets are not available to reports because report is a static document that represents the situation at the moment it was created. The reports can be triggered to be sent with email from the dashboards utilizing the dataflow editor triggers and, of course a report can be viewed by opening it manually from the IoT-Ticket user interface.

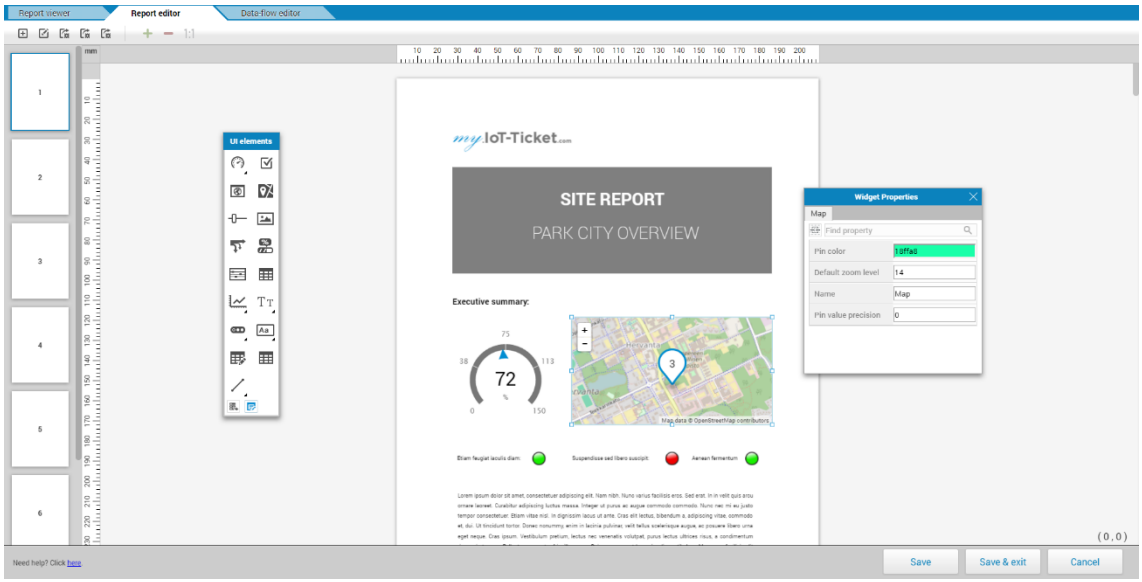

Figure 5.3 IoT-Ticket report editor

IoT-Ticket also offers web based analytics tool for analyzing the data stored by the devices to the IoT-Ticket server. The analytics tool can be used to execute different kinds of analysis for example, correlation and abnormal value detection on large amounts of data over long time ranges. The results of the analysis can be visualized to the user with graphs and matrices.

5.2 The report editor component

The report editor component package consists of several files and folders that contain the JavaScript files and other resources required by the report editor. The main JavaScript file of the component (report.min.js) provides separate Backbone.js views for report viewer (ReportView), report editor (ReporterView), dataflow editor (ReflowView) and pdf view (StandAloneReportView). The component package also contains view models for these four views called ReportViewModel, ReporterViewModel, ReflowViewModel and StandAloneReportViewModel.

The report is loaded to the report views by using a ReportModel that is also available in the package. Report model is initialized by parsing a JSON string that can be stored in a database. The report JSON has information of the dataflow and all the widgets that are related to the report. Once the report has been loaded to the view, it can be rendered to a HTML page.

Currently, the report editor component demands quite a lot from the developers utilizing it. For example, the control logic that renders the views that together form the full report editor has to be implemented by the developers utilizing the component. Additionally, the view model dependencies to the application state have to be also implemented separately.

The interactivity with the editor has also to be implemented by the developers by implementing functions for the component dependencies called state models or subscribing to the events provided by the NavigationEvents module included in the report editor package.

5.3 Report editor component quality evaluation

Next, the report editor component will be evaluated by utilizing the web frontend component quality model introduced in Chapter 4. Each of the characteristics will be evaluated separately by sub-characteristic and by attribute using the respective measure for each attribute. A result will be evaluated for each attribute. Later, in the next chapter improvement suggestions will be provided about how to improve the current quality of the report editor component based on these results.

The report editor component will be ran by utilizing a piece of test code that is included in the report editor component package for the evaluation that needs to inspect the report editor run time. The test code renders the report editor in an HTML page with an empty report and provides routing between the views.

5.3.1 Functionality

Functionality consists of four sub-characteristics that are evaluated according to the web frontend component quality model.

Browser compatibility

First sub-characteristic that is evaluated is browser compatibility. Browser compatibility consists of 1. cross-browser functionality and 2. browser support.

1. Cross-browser functionality is evaluated for the Chrome 57.0.2987.133, Firefox 52.0.2, Internet Explorer 11.839.10586.0 and Edge 25.10586.672.0 browsers. Chrome will be used as reference and other browsers are compared to those results. Because this evaluation is extremely time consuming only some of the most common features affecting the UI are selected for the cross-browser functionality evaluation. The features that will be tested are:

1. Inspecting the layout of the report viewer, report editor and report data flow editor for a report that contains a horizontal guide positioned at 50 mm from the top of the report page and a default height, full page width data table widget with top position at the guide. Additionally, the data table widget has a number sequence block with values 0-10 as its input.

2. Creating a new report page with data table widget as in feature 1 and an additional empty report page. Then, changing the order of the report pages. Observing the results in report viewer and report editor.
3. Opening the data table properties dialog and changing the font size to 16px and setting the text align property to ”Center”. Observing the effects on data table widget on report viewer and report editor.

The results for the features testing are presented in Table 5.1.

Table 5.1 Cross-browser functionality features testing observation results

Feature/ Browser	Chrome (Reference)	Firefox	Internet Explorer	Edge
1	-	**Report viewer**: Some of the data table 1px width borders seem wider. **Report editor**: Some of the data table 1px width borders seem wider. Chrome is showing the horizontal ruler 210mm position marker but FF is not. **Data-flow editor**: No differences	**Report viewer**: The report area borders are missing shadows that are present in Chrome. **Report editor**: Chrome is showing the horizontal ruler 210mm position marker but IE is not. Vertical ruler is hovering over the report area bottom. **Data-flow editor**: No differences	**Report viewer**: The report area borders are missing shadows that are present in Chrome. **Report editor**: Chrome is showing the horizontal ruler 210mm position marker but IE is not. **Data-flow editor**: No differences
2	-	**Report viewer**: No differences **Report editor**: No differences	**Report viewer**: No differences **Report editor**: No differences	**Report viewer**: No differences **Report editor**: No differences
3	-	**Report viewer**: No differences **Report editor**: No differences	**Report viewer**: No differences **Report editor**: No differences	**Report viewer**: No differences **Report editor**: No differences

2. Browser support information is not included in the report editor package documentation. Therefore, browser support result is evaluated at the same time as the cross-browser functionality using the same features. The requirements for this evaluation are set to the browser versions that are used on the cross-browser functionality evaluation implying the Chrome, Firefox, Internet Explorer and Edge.

Browser compatibility results:

1. **Cross-browser functionality**: three features were tested with four browsers where one browser was a reference point. Total of nine tests were done and three

of them resulted in finding noticeable differences in appearance or functionality. Thus, the calculated ratio for browser compatibility based on this evaluation is 6/9 = 0.67

2. **Browser support**: All browsers tested provided the functionality required by the tested features. Thus, the calculated ratio for browser support is 12/12 = 1

Backwards compatibility

This functionality sub-characteristic consists of two attributes: 1. data compatibility and 2. functional compatibility.

1. Data compatibility will be evaluated first. Documentation analysis confirms that the data compatibility information is not found on the report editor documentation. However, from the IoT-Ticket development process it is known that the data compatibility is an important feature because version updates do not update the saved reports and dashboards JSON data that are stored in a database. Thus, for the existing reports and dashboards to continue to work after updates the data has to be compatible with newer versions of the IoT-ticket than they were saved with.

Data compatibility is known to be present in the report editor for the most part. Some of the updates are known to take effect only on newly added widgets but old ones are still functioning as before. However, the information is not included in the documentation and the result for the presence measure has to be false.

2. Functional compatibility is evaluated next. Functional compatibility information is not present on the report editor documentation. Therefore, the result for the presence measure is false.

Backwards compatibility results:

1. **Data compatibility**: Result is not found in documentation. Thus, presence evaluates to false.
2. **Functional compatibility**: Result is not found in documentation. Thus, presence evaluates to false.

Suitability

This functionality sub-characteristic consists of three attributes: 1. coverage, 2. excess and 3. completeness. The suitability evaluation is done against the reference project that has utilized the report editor package. This evaluation is not particularly relevant because the report editor package evolved to suit the needs of this project. For the sake of completeness, the measurements are reported here.

1. Coverage is measured first. The application requires 13 modules implemented by the report editor package. These modules implement all the functionality required by the application and therefore the number of required modules is also 13.

2. Excess is measured next. Every module in the report editor package is utilized by the application and therefore the number of modules not utilized by the application is 0. The total number of modules in the package is 13.

3. Completeness is measured last. Every module that is specified in the component documentation is also present in the package.

Suitability results:

1. **Coverage**: Ratio evaluates to 13/13 = 1.
2. **Excess**: Ratio evaluates to 0/13 = 0.
3. **Completeness**: Ratio evaluates to 13/13 = 1.

Self-contained

Self-contained consists of a single attribute: 1. dependencies.

1. Dependencies is evaluated based on the component documentation. Documentation specifies that the report editor package view components require four modules (ReportStateModel, ResouceStateModel, UtilityStateModel and router) to function properly.

Self-contained results:

1. **Dependencies**: Integer value result is 4.

5.3.2 Reliability

Reliability consists of two sub-characteristics that are evaluated according to the web frontend component quality model.

Fault tolerance

Fault tolerance consists of two attributes: 1. Mechanism available and 2. mechanism efficiency.

1. Mechanism available is measured first. All the modules present in the report editor package are evaluated based on if their constructor functions or utilized methods are implementing a fault tolerance mechanism. The results are presented in Table 5.2.

Table 5.2 Mechanism available measurement results

Module	Constructor	Methods
ReportView	Model given as parameter is not verified to be defined. Thus, mechanism is not implemented.	**loadGraph**: if the report id provided is not matching the id of the report returned by the view model or the report id is not defined the report is attempted to be fetched from the default API. Thus, mechanism is implemented.
ReporterView	Result is identical to as above.	**loadGraph**: Result is identical to as above.
ReflowView	Result is identical to as above.	**loadGraph**: Result is identical to as above.
StandAloneReportView	Result is identical to as above.	**setReportData**: the report data given as parameter is not verified to be defined. Thus, mechanism is not implemented. **cacheMeasurementPoints**: No parameters. No mechanism is required. **render**: No parameters. No mechanism is required.
ReportViewModel	Models given as parameters are not verified to be defined. Thus, mechanism is not implemented.	No methods to be called by the user.
ReporterViewModel	Result is identical to as above.	No methods to be called by the user.
ReflowViewModel	Result is identical to as above.	No methods to be called by the user.
StandAloneReportViewModel	Result is identical to as above.	No methods to be called by the user.
ReportModel	Missing parameters are initialized with default values. Thus, mechanism is implemented.	**parse**: verifies if the parameter is defined. Also verifies if the parameter object contains properties before accessing them. Thus, mechanism is implemented.
ReportFlowModel	Missing parameters are initialized with default values. Thus, mechanism is implemented.	No methods to be called by the user.
NavigationEvents	No constructor. No mechanism is required.	**trigger**: Implemented by Backbone.js events. No mechanism is required.
ResourcesCollection	Constructor implemented by Backbone.js collection. No mechanism is required.	**get**: Implemented by Backbone.js collection. No mechanism is required. **fetch**: Implemented by Backbone.js collection. No mechanism is required.
Uuid	No constructor. No mechanism is required.	**generate**: No parameters. No mechanism is required.

2. Mechanism efficiency is evaluated next. This attribute will only be measured based on implemented mechanisms and the missing mechanisms are ignored because they are notified by the mechanism available attribute. The usage of default values and verification that parameter object properties exist before accessing them is fault tolerance increasing behavior. However, the report editor views are not verifying that the report id passed as parameter is defined before trying to fetch the report from default API which could be done to improve fault tolerance. Overall, the mechanisms utilized are adequate and the level from 0-4 will be evaluated to value 3.

Fault tolerance results:

1. **Mechanism available**: Measurement divided the report editor modules to two categories: constructor and methods. The total number of the modules evaluated is therefore 26. The number of these modules that implemented a mechanism is 11. Thus, the result for the ratio is 11/26 = 0.42.
2. **Mechanism efficiency**: The attribute is measured on a scale from 0 to 4 to level 3.

Recoverability

Recoverability consists of two attributes: 1. serializable and 2. transactional.

1. Serializable is measured by evaluating if the component has a mechanism for serializing its state. The report editor stores its state runtime in the ReportModel object. The report model provides a method for getting its contents in JSON format. The JSON object of the report state can thus be stored persistently. The report model can be created again with the JSON object when the saved report is wanted to be opened on the report editor again. Based on the evaluation the presence is measured as true.

2. Transactional is evaluated next by measuring if the component implements transactions that would allow the monitoring and modifying the history of actions done to the component. The report editor provides an undo and redo feature for certain commands that are executed on the report editor such as creating a report page or repositioning a widget on a report page. The undo/redo feature is not fully transactional however, since it does not store every action for example, the widget property modifications. It is also not possible to monitor the list of the done actions or store the commands persistently. The commands are also bound to the current instance of the report editor view implying that if the user switches between report editor view and report flow editor view the command history is lost. Thus, the presence value is measured as false.

Recoverability results:

1. **Serializable**: Presence evaluates to true.
2. **Transactional**: Presence evaluates to false.

5.3.3 Usability

Usability consists of two sub-characteristics that are evaluated according to the web frontend component quality model.

Configurability

Configurability consists of three attributes: 1. functional configurability, 2. event configurability and 3. appearance configurability.

1. Functional configurability is measured by evaluating the initialization parameters and configurations of the component modules. Currently the report editor package modules offer configuration through the state models which are the view model dependencies that are implemented by the application specific code (ReportStateModel, ResouceStateModel and UtilityStateModel). For example, the “save report” button handler must be implemented by the ReportStateModel.

While the state models offer a way to configure some of the functionality of the report editor this method is not optimal. As the state model name implies they are not originally intended for functional configuration but implementing the state handling of the report editor. At its current state, it might be confusing for the developers to identify the correct methods for functional configuration from the state handling methods though the documentation describes their use through an example project description.

Additionally, the package modules are not offering too many configuration options. The configurations only include saving of the report, closing of the report editor and setting unsaved changes state functionality while many other features could be configurable as well.

The component API interaction is also lacking configurability. The report editor component is connecting to API when it is fetching some of the resources it needs to operate correctly. This API interaction and the API URLs are not configurable by the developers apart from the root part of the URL.

Overall, the report editor offers the minimum amount of required functional configurability that allows the operation of the editor in an application. Thus, the configurability is evaluated as level 1.

2. Event configurability is evaluated next by measuring the events provided by the component modules. The modules that are valid for this evaluation are the ReportView, ReporterView, ReflowView and StandAloneReportView because they are the UI modules of the report editor package.

Not any of the mentioned views offer events that the application code could react to with callback functions. The events communication can be handled through the NavigationEvents module available in the report editor package but the events it contains are not documented apart from the navigation events between the report views. The NavigationEvents module also contains all the internal events that are utilized by the report editor which implies that there are many events that are not meaningful to the application.

Additionally, the event listening through the NavigationEvents is depending on the Backbone.js events functionality instead of plain JavaScript.

The event configurability of the report editor component can be implemented with an unconventional way through the NavigationEvents object available in the package. Consequently, the event configurability is evaluated as level 1 because like functional configurability it only offers the minimum required features.

3. Appearance configurability is measured by evaluating the effort required to modify the visual appearance of the report editor views. Like the event configurability, the appearance configurability is also valid only for the UI modules of the package.

The report editor package does not offer any other way of modifying its appearance than providing a CSS file with custom styles. There is also no documentation of the customization available implying that the developers have to use their own expertise in applying the styles to desired elements of the report editor views. The level measurement for appearance configurability is evaluated as 1 because the component does not support appearance configurability in any other way than the regular CSS modification.

Configurability results:

1. **Functional configurability**: The attribute is measured on a scale from 0 to 4 to level 1.
2. **Event configurability**: The attribute is measured on a scale from 0 to 4 to level 1.
3. **Appearance configurability**: The attribute is measured on a scale from 0 to 4 to level 1.

Understandability

Understandability consists of three attributes: 1. documentation coverage, 2. documentation quality and 3. demonstration coverage.

1. Documentation coverage is evaluated by inspecting the component documentation and the actual implementation of the component. The functionalities that are expected to be found from the documentation and their results are presented in Table 5.3.

Table 5.3 Documentation coverage measurement results

Functionality	Documentation entry
Installation and updating	The report editor package building is explained. The documentation also explains that the installation is done by copying the report editor package to the target application project and including the relevant files from the package to a HTML page. The updating of the package is also described in the documentation. Thus, installation and updating information **is present** in the documentation.
Configuration options	The configurations that are available (saving report, closing the editor, setting unsaved changes state) are described in the documentation. The editor initialization with a saved report is also explained on a general level implying that the actual code required to initialize the editor is not described. The setting of the authorization token for the API is also explained in the documentation if it is required by the application. Thus, the configuration options **are present** in the documentation.
Methods	The rendering methods of the report editor views are explained. However, the documentation is missing descriptions for each module available in the component for their provided methods. Thus, the method descriptions **are not fully present** in the documentation.
Events	The navigation events needed by the router are explained in the documentation. Some other events can also be utilized that are not found on the documentation for example, the widget selection event that is used to set selected widget ids to the state and reset scaling event that is used to reset the zooming when switching between report editor views. Thus, the events **are not fully present** in the documentation.
Appearance configuration	The report editor appearance configuration **is not present** in the documentation.
Dependencies	Navigation between report editor views is explained in the documentation. Additionally, the state models and most of their methods are presented in the documentation. However, some of the latest additions in methods are missing from the state models that need to be implemented. The API interactions are also found from the documentation. Thus, the dependencies **are not fully present** in the documentation.

2. Documentation quality is evaluated next. The report editor component consists of many modules and it requires many dependencies to operate correctly. The relations of the package modules and their dependencies are explained by images similar to UML deployment diagrams. Additionally, an actual usage of the package in an application is described in the documentation.

There are many code examples of the report editor package usage including the report views construction and attaching event listeners to navigation events. On the other hand, the documentation is lacking examples of initializing the report model which is critical for the implementation of the loading of a saved report unless the default report API is utilized.

The document structure is logical. It starts with introducing the package contents and the modules it includes. Then it proceeds to present the utilization of the modules including the dependencies they require. However, the documentation could make a clearer distinction between describing the dependencies and other usage of the report editor modules. At the current state of the documentation there is not a clear list of what is required of the application that is using the report editor component. The documentation also introduces an example utilization case of the report editor in a project.

The documentation is only available as a word document and therefore it might be more difficult to utilize as a developer than for example, a HTML-based documentation.

Overall, the documentation has a logical structure and presents most of the information required by the developers utilizing the component. It also utilizes deployment diagrams to describe the relations between the component modules and the application specific code. Additionally, the documentation utilizes code examples. However, it is also lacking some important examples and the documentation format could be more accessible for the developers. Therefore, the documentation quality is evaluated as level 2.

3. Demonstration coverage is evaluated against the test code that is available in the report editor package because it can be utilized by developers that want to use the report editor package in an application. The results for this evaluation are presented in Table 5.4.

Table 5.4 Demonstration coverage measurement results

Functionality	Demonstration description
Installation	Demonstration shows the inclusion of the relevant report editor package files in a HTML page that is used as a container for the report editor component. Thus, installation **is present** in the demonstration.
Configuration options	The configurations that are available (saving report, closing the editor, setting unsaved changes state) are not shown on the demonstration. Loading a saved report is not shown on demonstration. The demonstration does not fetch any data to be shown on the report from the API. Thus, the configuration options **are not present** in the demonstration.
Methods	The rendering methods of the report editor views and the resources collection methods are shown in the demonstration. However, all the report model, uuid and StandAloneReportView methods are not shown. Thus, the methods **are not fully present** in the demonstration.
Events	The navigation events required to provide navigation between the report editor views are shown on the demonstration. Other events are not presented on the demonstration for example, the widget selection event or reset scaling event. Thus, events **are not fully present** in the demonstration.
Appearance configuration	Appearance configurations **are not present** on the demonstration.

Functionality	Demonstration description
Dependencies	Navigation between report editor views is shown in the demonstration. Additionally, the state models and most of their methods are presented. Some of the most recently added methods are missing from the state models in the demonstration. Thus, the dependencies **are not fully present** in the demonstration.

Understandability results:

1. **Documentation coverage**: Two of the six main functionalities are present in the documentation. The result for the ratio is 2/6 = 0.33.
2. **Documentation quality**: The attribute is measured on a scale from 0 to 4 to level 2.
3. **Demonstration coverage**: One of the six main functionalities are present in the demonstration. The result for the ratio is 1/6 = 0.17.

5.3.4 Efficiency

Efficiency consists of two sub-characteristics that are evaluated according to the web frontend component quality model.

Resource behavior

Resource behavior consists of two attributes: 1. memory utilization and 2. disk utilization.

1. Memory utilization is evaluated by utilizing memory-stats-js JavaScript library (memory-stats-js 2017). When the report editor is loaded with minimum application that is only containing the HTML page as container for the editor, the memory utilization is evaluated to approximately 28 MB.

2. Disk utilization is evaluated by inspecting the disk space that the report editor package is occupying. The disk space is evaluated to 41.3 MB.

Resource behavior results:

1. **Memory utilization:** Integer value is evaluated to 28 MB.
2. **Disk utilization:** Integer value is evaluated to 41.3 MB.

Scalability

Scalability consists of one attribute: 1. processing capacity.

1. Processing capacity is evaluated by measuring the report generation time for reports with different data-flow loads. The report generation time is important for the report editor component because its main function is to calculate the report data-flow and generate

and present the report to the user. The measurement is done by creating reports that process data generated by *simulate* data-flow blocks connected into *input label* widgets. The time it takes for the editor to send the *graph processed* event that indicates that the flow calculation and page rendering is completed is measured for each different number of data-flow blocks by utilizing the *StandAloneReportView* that is used for the pdf file generation of reports. The *console.time()* method in JavaScript is utilized to measure the time by starting the time right before calling the *render* method of StandAloneReportView and registering a listener to the graph processed event which stops the time when the event is received. The simulate blocks are using the default configuration that generates a value between 0 and 100. The input label widgets are also using default configurations. All the widgets are inserted into one report page. The results for this evaluation are presented on Table 5.5.

Table 5.5 Processing capacity measurement results

Number of Simulate and Input Label blocks	Measurement 1	Measurement 2	Measurement 3	Mean
0 + 0	3328.89ms	3344.07ms	3333.76ms	3335,57ms
10 + 10	3482.18ms	3472.30ms	3456.51ms	3470.33ms
50 + 50	4034.54ms	3976.38ms	4005.49ms	4005.47ms
100 + 100	4919.99ms	4966.61ms	4930.28ms	4938.96ms
200 + 200	6245.07ms	6955.32ms	6384.49ms	6528.29ms

Simple linear regression model for report generation time with number of dataflow blocks as explanatory variable produces a result (graphpad.com 2017):

Report generation time (ms) = 8.067*number of dataflow blocks + 3293 (ms)

The result has R^2 of 0.9983 which represents a very good fit. The model has a constant of 3293ms which represents the static time needed to generate a report regardless of the number of dataflow blocks.

Scalability results:

1. **Processing capacity:** According to the linear model when ignoring the constant, the processing capacity of report editor component is 1 dataflow block/8.067ms = 0.124 dataflow blocks/ms or 124 dataflow blocks/s.

5.3.5 Maintainability

Maintainability consists of one sub-characteristic that is evaluated according to the web frontend component quality model.

Testability

Testability consists of two attributes: 1. test suite provided and 2. tests in a specific environment.

1. Test suite provided is evaluated first. The report editor package does not include a runnable test suite. There are unit tests, integration tests and end-to-end tests available for the report editor but they are not included in the package. Therefore, the presence attribute is evaluated to false.

2. Tests in a specific environment is measured by verifying if the documentation contains relevant information about the tests that have been executed on the report editor component. The documentation does not contain information of the tests and if they are done on different browsers or if the component is tested in applications utilizing different JavaScript frameworks. Thus, the presence attribute is evaluated to false.

Testability results:

1. **Test suite provided**: Presence evaluates to false.
2. **Tests in a specific environment**: Presence evaluates to false.

5.3.6 Portability

Portability consists of two sub-characteristics that are evaluated according to the web frontend component quality model.

Installability

Installability consists of four attributes: 1. framework support, 2. loading type coverage, 3. installation simplicity and 4. application to DOM.

1. Framework support is measured by evaluating if the component provides a framework supported version of itself that is known to be required by applications. Currently, the report editor does not provide any such versions of itself. The application where the report editor has been utilized is developed with React, so providing a React component of the report editor would have made the development easier.

2. Loading type coverage is measured by evaluating which loading types are supported by the report editor package. The tested types are script tag, AMD and CommonJS. The report editor package is utilizing only the script tag as it assigns itself into the window object. Thus, AMD and CommonJS are not supported.

3. Installation simplicity is measured by evaluating the effort required to install and update the component. The installation itself is not difficult because the developers have to only include the report editor package script file and CSS file to a HTML page and the

report editor is available in the application. The update is done utilizing the same method where the report editor package files are overwritten with the updated files. However, the installation and update is not automated for example, with NPM so the developers have to do it manually as described before. Therefore, the installation simplicity is evaluated as level 3 because of the lack of installation and update automation.

4. Application to DOM is measured by evaluating how simple HTML markup is required by the component. The report editor component requires a lot compared to regular JavaScript UI components because the application has to provide a separate HTML page to act as a container for the report editor. This is because the report editor views append themselves directly to the body element of the HTML page they are on. To include the report editor inside an application view, an iframe HTML element or similar approach has to be utilized which makes the communication with the component quite complex. Thus, the application to DOM is evaluated as level 1.

Installability results:

1. **Framework support**: No framework specific version of the component is supported though a React component could be utilized. Thus, the result of the ratio is 0/1 = 0.
2. **Loading type coverage**: One of three loading types are supported by the component. Thus, the result of the ratio is 1/3 = 0.33.
3. **Installation simplicity**: The attribute is measured on a scale from 0 to 4 to level 3.
4. **Application to DOM**: The attribute is measured on a scale from 0 to 4 to level 1.

Reusablity

Reusability consists of three attributes: 1. modularity, 2. coupling and 3. architecture compatibility.

1. Modularity is measured by evaluating the ratio of modules found in the component to the functionalities provided by the component. The functionalities provided by the report editor component are roughly report previewing, report editing, report data flow editing, report pdf view, navigation events and uuid generation, which makes a total of six functionalities. The total number of modules provided by the report editor package is 13. Consequently the result of the ratio is 13/6 = 2.2.

2. Coupling is measured by calculating how many modules provided by the component require another module of the component to function properly. The number of those modules is four because all the report views require a view model implementation.

3. Architecture compatibility is measured by evaluating how great effect the usage of the component has on the architecture of the whole application. The report editor has some

implications to the architecture because currently it is required to use iframe HTML element to use the editor in an application. The communication between the application and the iframe containing the report editor component has to be implemented by utilizing the JavaScript's window object. For example, the application has to set callback functions as properties to the window object for the report editor to be able to access them inside the iframe.

The report editor package does not support AMD or CommonJS which implies that the report editor has to be included with script tags or by taking additional measures to get the package to be compatible with AMD or CommonJS. The report editor views also require to be appended to the body element of the HTML page they are on, implying that they force the application to provide their own HTML page for the component.

Overall, the effects on the architecture are quite heavy and thus, the level is evaluated to 2.

Reusability results:

1. **Modularity**: The number of functionalities provided by the component is six and the number of modules is 13. Thus, the result of the ratio is 13/6 = 2.2.
2. **Coupling**: The number of modules that require another module is four and the number of modules is 13. Thus, the result of the ratio is 4/13 = 0.31.
3. **Architecture compatibility**: The attribute is measured on a scale from 0 to 4 to level 2.

5.3.7 Summary of the quality evaluation

The results for the report editor component evaluation using the web frontend UI component quality model are presented in Table 5.6.

Table 5.6 Report editor quality evaluation results

Characteristic	Sub-characteristic	Attribute	Type	Evaluation result
Functionality	Browser compatibility	Cross-browser functionality	Ratio	0.67
		Browser support	Ratio	1
	Backwards compatibility	Data compatibility	Presence	False
		Functional compatibility	Presence	False
	Suitability	Coverage	Ratio	1
		Excess	Ratio	0
		Completeness	Ratio	1
	Self-contained	Dependencies	Integer	4
Reliability	Fault tolerance	Mechanism available	Ratio	0.42
		Mechanism efficiency	Level	3
	Recoverability	Serializable	Presence	True

Characteristic	Sub-characteristic	Attribute	Type	Evaluation result
		Transactional	Presence	False
Usability	Configurability	Functional configurability	Level	1
		Event configurability	Level	1
		Appearance configurability	Level	1
	Understandability	Documentation coverage	Ratio	0.33
		Documentation quality	Level	2
		Demonstration coverage	Ratio	0.17
Efficiency	Resource behavior	Memory utilization	Integer	28 MB
		Disk utilization	Integer	41.3 MB
	Scalability	Processing capacity	Integer	124 data-flow blocks/s
Maintainability	Testability	Test suite provided	Presence	False
		Tests in a specific environment	Presence	False
Portability	Installability	Framework support	Ratio	0
		Loading type coverage	Ratio	0.33
		Installation simplicity	Level	3
		Application to DOM	Level	1
	Reusablity	Modularity	Ratio	2.2
		Coupling	Ratio	0.31
		Architecture compatibility	Level	2

5.4 Quality evaluation implications and improvement suggestions

Next, the results of the quality evaluation that was done in Section 5.3 are contemplated and improvement suggestions are made. This section is based on the quality evaluation results that are shown in Table 5.6 but the reasoning that led to these results presented in Section 5.3 is utilized as well. Each sub-characteristic of the characteristics presented in the model are discussed.

5.4.1 Functionality

The *browser compatibility* of the report editor is overall at adequate level with *cross-browser functionality* at ratio 0.67 and *browser support* at ratio 1. Thus, the editor itself is functioning with all tested browsers but there are minor cosmetic differences between different browsers. The cross-browser functionality ratio 0.67 could be interpreted as fairly low but as stated before the differences were cosmetic. However, based on this

evaluation it can be suggested that also more of the most important features would be tested for cross-browser functionality.

Backwards compatibility was evaluated based on the information that is available in the component documentation regardless of what is internally known about the component. The information about *data compatibility* and *functional compatibility* was not found and therefore they were both evaluated as false. Based on this evaluation it is suggested that information about data compatibility and functional compatibility is added to the component documentation.

Suitability evaluation passed with high scores because the application where the report editor had already been utilized, was used as the reference application for the measurement. Thus, the reference application directly affected the report editor package contents during the package development. Therefore, the ratio results for *coverage, excess* and *completeness* were 1, 0 and 1 respectively. This evaluation did not add any value to this analysis but overall suitability may be useful for evaluating component suitability for another application. Especially, if component has excess features and thus, excess documentation, its complexity may increase because developers might not find the features they need.

Self-contained characteristic was evaluated using the *dependencies* attribute. The result found was that the report editor overall depends on four external modules that need to be provided by the application developers in order to properly utilize the report editor. Currently, these dependencies are mostly utilized for providing application specific functionality such as report saving handler that could be achieved by providing more options directly from the report editor component interface. The state handling that the dependencies provide can be wrapped inside the report editor component to abstract them from the application developers. Thus, it is suggested that the component dependencies are dropped to reduce the dependencies and the complexity it implies to the component.

5.4.2 Reliability

Fault tolerance was found overall adequately efficient when it is utilized in the component. The ratio for *mechanism available* was 0.42 which is fairly low. The *mechanism efficiency* was evaluated at level 3 which is a good result. It is understandable that the fault tolerance has not received major attention because the component has been utilized by developers that have been participating in its development and are therefore familiar with it and are not likely to call methods incorrectly. However, it is suggested that more attention is paid to the fault tolerance coverage of the modules and methods that the component provides.

Recoverability of the component was evaluated based on *serializable* and *transactional* attributes. The component was found to be serializable but the transactionality is not fully

implemented. The transactionality could be improved by preserving the command history between the switching of the report editor views and adding more actions such as property changes to be commands. The commands could also be considered to be stored persistently to provide command history across multiple sessions of editing a report.

5.4.3 Usability

The component *configurability* was evaluated utilizing the *functional configurability, event configurability* and *appearance configurability*. All the attributes were evaluated to level 1 which implies that the configurability could be improved. Thus, it is suggested that the component would provide more configuration options and provide an alternative, more simple method for supplying the configuration options than the current state model implementation. For example, the report editor could accept a configuration object as a parameter when the report editor is initialized and the configurations could be modified later by a method call.

The component API interaction could be improved by allowing the developers to pass the resources the report editor needs when initializing the component or to allow the developers to fully customize the API URLs the report editor is using. The API configurations could also support modifying the HTTP header fields to make the user authentication method configuration easier.

Currently, the event configurability is too complex because developers have to utilize the NavigationEvents module and the state models to implement event handling. The situation could be improved by providing an event interface from the component which would allow developers to attach callback functions to the events that are triggered from the component such as report saving and closing of the editor.

The appearance configurability currently supports only external CSS styles. This could be improved by providing themes and appearance configuration that could be modified with configuration options.

Understandability was evaluated by considering the *documentation coverage, documentation quality* and *demonstration coverage.* The evaluation found room for improvement from all the three attributes. Documentation coverage ratio was evaluated as 0.33 when main functionality descriptions such as installation and configuration options were searched. Thus, the documentation could be revised by adding more of the relevant information to the developers utilizing the component.

Documentation quality was found to be level 2 which could be improved. The improvement suggestions are to provide HTML-based documentation that would make it easier for developers to traverse between the documentation sections. Additionally, documentation could show more examples of some of the most critical functionalities.

Demonstration coverage ratio was evaluated to 0.17 by examining the test code that is present in the report editor package. This implies that the demonstration is not adequate and needs to be improved to be useful. The current demonstration does not fully show other main functionalities than installation. For example, report editor interactions such as saving of the report and loading a saved report to the editor are not presented. Thus, demonstration could include more complete use cases of the report editor component.

5.4.4 Efficiency

Resource behavior was evaluated by measuring the *memory utilization* and *disk utilization* of the report editor package. The memory utilization was evaluated at 28 MB and disk utilization at 41.3 MB. The resource behavior results are difficult to analyze because they would be needed to be compared to components with similar functionality. However, the disk utilization value appears to be relatively large and it could be verified that the package does not contain any extra resources that are not required by the report editor.

Scalability was evaluated by measuring the *processing capacity.* The report generation time seems to increase linearly as the number of dataflow blocks increases which gives an indication that the component is scalable. However, the static time to generate a report regardless of the amount of dataflow blocks is over 3 seconds which could possibly be reduced by changing the graph processed event to be sent right away when report is processed and rendered. At the moment, safety time is utilized to make sure that everything is rendered.

5.4.5 Maintainability

Testability was evaluated based on *test suite provided* and *tests in a specific environment* attributes. Both presence values evaluated to false. Thus, it is suggested that test suite would be provided with the component or test results would be presented along with the documentation. Documentation could also provide information about tests done utilizing different browsers.

5.4.6 Portability

Installability was evaluated by inspecting *framework support, loading type coverage, installation simplicity* and *application to DOM*. Framework support ratio was evaluated to 0 because the report editor does not provide a React component implementation that could have been used in a project that utilized the report editor component. Thus, it could be considered to provide framework supported versions of the report editor component if those are required by applications.

Loading type coverage ratio was evaluated as 0.33 implying that the component does not support CommonJS and AMD. Adding support to also these loading types is suggested.

Support can be added by utilizing universal module definition (UMD) format to wrap the report editor component (github.com/umdjs/umd, 2017).

Installation simplicity was evaluated as level 3 which indicates that the installation is fairly simple. However, to further increase the simplicity of the installation it could be automated by distributing the component as NPM package or similar.

Application to DOM was evaluated as level 1 so it has room for improvement. The complexity of the report editor application to DOM could be reduced by finding alternative to the iframe element or wrapping the iframe element inside the report editor component so the communication with the iframe could be abstracted from the developers utilizing the report editor package.

Reusability evaluation was done by measuring the *modularity, coupling* and *architecture compatibility* of the component. The modularity evaluation produced interesting results because the ratio evaluated to 2.2. This implies that the report editor package is too modularized compared to the functionalities it provides. This situation could be remedied by reducing the number of modules in the package by providing a more abstract report editor component that utilizes internally some of the modules that are now available in the package but do not provide any features to the developers.

Coupling ratio was measured as 0.31. Preferably, the result should be 0 which would imply that the component modules are not depending on each other. Coupling could be reduced to 0 by redesigning the package structure as suggested in the modularity improvement suggestion.

Architecture compatibility was evaluated to level 2 which is the average result. The points for improvement are similar to the ones in application to DOM. The communication with the iframe and the implementation of the HTML page are influencing the supporting architecture the report editor package requires. Additionally, the lack of CommonJS and AMD loading affects the way the application has to load the package. Thus, there should be efforts for abstracting the HTML and iframe handling inside the component and add support for CommonJS and AMD loading.

5.5 Web frontend component quality model evaluation and future work

The web frontend component quality model is now evaluated according to its performance earlier in this chapter. First, it is stated that the evaluation produced a relatively large number of improvement suggestions which was one of the goals of this thesis. Improvement suggestions were made to the actual component implementation as well as to the component documentation which implies that the model is able to produce suggestions to improve the overall quality of the component.

The results provided by the model were presented in Table 5.6 on page 61. However, the actual improvement suggestions were not solely based on the measured results. The reasoning that led to those results seem to be of equal or even greater importance when determining the improvement suggestions based on the evaluation. Of course, the results table provides an overview of the quality status regarding each attribute, sub-characteristic and characteristic and can be used to indicate which characteristics need the most attention.

The initial requirements that were set for the web frontend component quality model included selecting quality attributes that can be measured with adequate precision. After the evaluation, it can be stated that it was possible to measure a value for each attribute with the measures presented in the model. However, some of the measurements were fairly shallow considering the overall component functionality. For example, measuring the cross-browser functionality or memory utilization can be done more thoroughly if it is seen important and the evaluator has sufficient time to execute the evaluation. Additionally, the report editor as a component is very large, implying that it should be significantly easier to conduct an evaluation for a smaller component.

As has been stated before, the model requires quite strong expertise from the evaluator. The array of quality attributes that are measured is wide and the evaluator has to identify specific structures from the component. For example, for modularity the total number of functionalities provided by the component has to be identified. The expertise of the evaluator could thus, affect the outcome of the evaluation on some attributes if the evaluator is not able to identify everything that is required by the measurement. This could potentially lead to differing results between two different quality evaluations of the same component and thus affect the objectivity of the evaluation.

Therefore, future work could include measuring the same component by different evaluators and comparing the results to measure the objectivity of the model. Future work could also include evaluating smaller components to see how well the model finds quality problems from them. It would also be interesting to find out if all the attributes seem relevant for smaller components. The model could also be tested as how well it provides guidelines for developing a completely new component.

It could also be considered to execute an evaluation on a simple 3rd party component to see how well the model could be utilized when selecting a component for an application like the original CBSD models have been utilized. Most of the attributes could possibly be measured based on component documentation and a simple demo application.

The component subject discussed in this thesis could also be utilized to spark discussion inside the company to develop a web frontend component repository to be utilized across web application projects. The repository could then act as a way to spread expertise through well-structured multi-purpose components and additionally reduce development

time and costs when components could be utilized instead of developing own components for each application.

Naturally, future work also includes implementing the improvement suggestions provided by the quality evaluation to the report editor package based on the demand of the report editor package in future projects.

6. CONCLUSIONS

The aim of this thesis was to develop a web frontend component quality model. The model was developed based on web application component characteristics and existing research on software component quality models. The web frontend component quality model divides the quality hierarchically to four levels that are quality characteristics, quality sub-characteristics, quality attributes and quality measures. The model utilizes four types of quality measures that are presence, level, ratio and integer.

The relevant quality sub-characteristics and attributes were selected from the software component quality models introduced by the research and some were also added based on web frontend component characteristics. Thus, a quality model was formed that consists of six quality characteristics, 13 quality sub-characteristics and 30 quality attributes and measures to assess them.

The web frontend component quality model aims to evaluate the overall component quality implying that the evaluation focuses on both component implementation and documentation. The six quality characteristics presented in the model also cover all the ISO 9126 software quality evaluation standard characteristics which speaks for the overall quality aspect of the model.

The web frontend component quality model was tested and evaluated by assessing the quality of the report editor component that was extracted from the IoT-Ticket product developed by Wapice Ltd. The result of the quality evaluation according to the model provided an array of results that can be used as a view to the overall quality of the component. The quality evaluation also inspired improvement suggestions to the report editor component implementation and documentation.

Among the improvement suggestions to the report editor component were, for example, to increase the component configurability by adding a support to a configuration object and to implement events interface for the component. It was also suggested that the component would support alternative loading types for example CommonJS and AMD. The component documentation could also be improved by providing a HTML version of the documentation and including more of the relevant general information, for example, about the version compatibility and browser support.

All in all, the model was tested against a real component that has been utilized in the software industry in production environment. Thus, the web frontend quality model is able to measure the quality of web frontend components utilized by software companies.

REFERENCES

Almeida, F.L.F; Calistru, C.M. (2011). Assessing Quality Issues in Component Based Software Development. International Journal of Advanced Research in Computer Science. Volume 2, Issue 2, March 2011, pp. 212-218.

Alvaro, A.; Santana de Almeida, E.; Meira, S. L. (2005). Quality Attributes for a Component Quality Model. In the 10th International Workshop on Component Oriented Programming (WCOP) in conjunction with the 19th ACM European Conference on Object Oriented Programming (ECCOP), Glasgow, Scotland, 2005.

Alvaro, A.; Santana de Almeida, E.; Meira, S. L. (2006). A Software Component Quality Model: A Preliminary Evaluation. Conference on Software Engineering and Advanced Applications, August 2006. SEAA '06. 32nd EUROMICRO.

Ambler, T.; Cloud, N. (2015a). Chapter 2 – Grunt, JavaScript Frameworks for Modern Web Dev. Apress. 502p. Available http://library.books24x7.com/toc.aspx?bookid=101452 Accessed on 5.2.2017.

Ambler, T.; Cloud, N. (2015b). Chapter 5 – RequireJS, JavaScript Frameworks for Modern Web Dev. Apress. 502p. Available http://library.books24x7.com/toc.aspx?bookid=101452 Accessed on 5.2.2017.

Bertoa, M. F.; Vallecillo, A. (2002). Quality Attributes for COTS Components. In the Proceedings of the 6th International ECOOP Workshop on Quantitative Approaches in Object Oriented Software Engineering (QAOOSE). Available http://www.lcc.uma.es/~av/Publicaciones/02/bertoa-QAOOSE.pdf Accessed on 26.2.2017

Bertoa, M. F.; Troya, J. M., Vallecillo, A. (2005). Measuring the usability of software components. Journal of Systems and Software, Volume 79, Issue 3, March 2006, pp. 427–439.

Bos, W. (2015). An Intro To Using npm and ES6 Modules for Front End Development http://wesbos.com/javascript-modules/ Accessed on 5.2.2017.

Brooks, D.R. (2007) An Introduction to HTML and JavaScript for Scientists and Engineers, Springer. 191p.

Brown, M. (2016). Understanding JavaScript Modules: Bundling & Transpiling. https://www.sitepoint.com/javascript-modules-bundling-transpiling/ Accessed on 5.2.2017.

Collison, S. (2006). Beginning CSS Web Development, Apress. 413p.

developer.mozilla.org (2017a). CSS Media queries. https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries Accessed on 4.2.2017.

developer.mozilla.org (2017b). HTTP access control (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS Accessed on 26.5.2017.

docs.microsoft.com (2017a). Routing in ASP.NET Web API. https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api Accessed 26.5.2017.

docs.microsoft.com (2017b). ASP.NET MVC Views Overview. https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/asp-net-mvc-views-overview-cs Accessed 26.5.2017.

docs.microsoft.com (2017c). Introduction to ASP.NET Web Programming Using the Razor Syntax. https://docs.microsoft.com/en-us/aspnet/web-pages/overview/getting-started/introducing-razor-syntax-c Accessed 26.5.2017.

docs.webplatform.org (2017). Minification. https://docs.webplatform.org/wiki/concepts/programming/javascript/minification Accessed on 5.2.2017.

entityframeworktutorial.net (2017). What is Entity Framework? http://www.entityframeworktutorial.net/what-is-entityframework.aspx Accessed on 2.6.2017.

Fink, G.; Flatow, I. (2014a). Chapter 1 - Introducing Single Page Applications, Pro Single Page Application Development: Using Backbone.js and ASP.NET. Apress. 324p. Available http://library.books24x7.com/toc.aspx?bookid=64620 Accessed on 5.2.2017.

Fink, G.; Flatow, I. (2014b). Chapter 4 - SPA Concepts and Architecture, Pro Single Page Application Development: Using Backbone.js and ASP.NET. Apress. 324p. Available http://library.books24x7.com/toc.aspx?bookid=64620 Accessed on 5.2.2017.

Firesmith, D. (2005). Achieving Quality Requirements with Reused Software Components: Challenges to Successful Reuse, Second International Workshop on Models and Processes for the Evaluation of off-the-shelf components (MPEC'05), May 2005, pp. 16-29. Available: http://resources.sei.cmu.edu/asset_files/Presentation/2005_017_001_22301.pdf Accessed on 24.2.2017

Freeman, A. (2011). The Definitive Guide to HTML5, Apress. 1041p.

getbootstrap.com (2017). Bootstrap framework. http://getbootstrap.com/ Accessed on 4.2.2017

github.com/umdjs/umd (2017). Universal module definition. https://github.com/umdjs/umd Accessed on 19.5.2017.

graphpad.com (2017). Online linear regression calculation. http://www.graphpad.com/quickcalcs/linear1/ Accessed on 4.6.2017.

The Institute of Electrical and Electronics Engineers, Inc. (1990). IEEE Standard Glossary of Software Engineering Terminology. Available: http://ieeexplore.ieee.org/document/159342/ Accessed on 19.2.2017.

The International Organization for Standardization, The International Electrotechnical Commission (ISO/IEC). (2011). ISO/IEC 25010:2011. Available: https://www.iso.org/obp/ui/#iso:std:35733:en Accessed on 19.2.2017.

iot-ticket.com (2017). IoT-Ticket. https://www.iot-ticket.com/ Accessed on 31.3.2017.

jqueryui.com (2017). jQuery UI library. https://jqueryui.com/ Accessed on 10.2.2017.

Kalaimagal, S.; Srinivasan R. (2008). A Retrospective on Software Component Quality Models. ACM SIGSOFT Software Engineering Notes, Volume. 33, no. 6, 2008, pp. 2-8. Available http://dl.acm.org.libproxy.tut.fi/citation.cfm?id=1449611 Accessed on 26.2.2017

Kaur, I; Sandhu, P. S.; Singh, H.; Saini, V. (2009) Analytical Study of Component Based Software Engineering. World Academy of Science, Engineering and Technology. Volume. 50, 2009, pp. 437-442.

memory-stats.js (2017). JavaScript memory monitor. https://github.com/paulirish/memory-stats.js/tree/master Accessed on 29.4.2017

msdn.microsoft.com (2017). ASP.NET Routing. https://msdn.microsoft.com/en-us/library/cc668201.aspx Accessed on 26.5.2017.

docs.oracle.com (2017). A Relational Database Overview. https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html Accessed on 26.5.2017.

O'Regan, G. (2014). Chapter 1 – Introduction, Introduction to Software Quality. Springer. 369p. Available: http://library.books24x7.com/toc.aspx?bookid=77004 Accessed on 19.2.2017.

Osmani, A. (2012). Journey Through the JavaScript MVC Jungle. https://www.smashingmagazine.com/2012/07/journey-through-the-javascript-mvc-jungle/ Accessed on 5.2.2017.

typescriptlang.org (2017). TypeScript. https://www.typescriptlang.org Accessed on 4.2.2017

W3.org. (2017). HTML specification. https://www.w3.org/TR/2011/WD-html5-20110525/ Accessed on 3.2.2017.

w3schools.com (2017). CSS manipulation with DOM. http://www.w3schools.com/js/js_htmldom_css.asp Accessed on 4.2.2017.

webpack.org (2017). Webpack. https://webpack.js.org/ Accessed on 5.2.2017.

Wellman, D. (2009). Chapter 2 – Tabs, jQuery UI 1.6: The User Interface Library for jQuery. Packt Publishing, 440p. http://library.books24x7.com/toc.aspx?bookid=30363 Accessed on 10.2.2017.

APPENDIX A: WEB FRONTEND COMPONENT QUALITY MODEL

Characteristic	Sub-characteristic	Attribute	Type	Measure
Functionality	Browser compatibility	Cross-browser functionality	Ratio	First, to evaluate the cross-browser functionality of a component, the evaluator selects the browsers and their specific versions to be used in the evaluation. The component features to be evaluated are also selected. One browser and its specific version is selected as a reference point for correct functionality of the features of the component. The features of the component in other browsers and their versions are compared to the component features in the reference point browser. The result is provided as a ratio of how many of the component features in other browsers match the reference point.
		Browser support	Ratio	This attribute provides two alternative methods of measurement: 1. If component provides documentation or testing documentation that states supported browser versions they may be utilized. 2. Component is ran manually as part of an application on different browser versions and results are observed. The result is provided as a ratio of how many required browser versions are supported.
	Backwards compatibility	Data compatibility	Presence	This attribute is measured by verifying if the attribute is present on the component. This may be achieved by finding such statement from the component documentation or by manually verifying the data compatibility when updating the component version.
		Functional compatibility	Presence	This attribute is measured with the same procedure as the data compatibility.
	Suitability	Coverage	Ratio	Measured by dividing the number of modules in the component that implement some of the functionality required by the application with the number of modules required by the application.
		Excess	Ratio	Measured by dividing the number of modules in the component not utilized by the application with the number of modules provided by the component.
		Completeness	Ratio	Measured by dividing the number of modules provided by the component with the number of modules specified by the component documentation.
	Self-contained	Dependencies	Integer	Measured by calculating the number of required modules by the component.

Characteristic	Sub-characteristic	Attribute	Type	Measure
Reliability	Fault tolerance	Mechanism available	Ratio	Measured by dividing the number of modules provided by the component that implement a fault tolerance mechanism with the total number of modules provided by the component.
		Mechanism efficiency	Level	Measured on a general level for the whole component with a value between 0-4.
	Recoverability	Serializable	Presence	Measured by evaluating if the component has a mechanism for serializing its state for example, to a JSON format that can be loaded later on the component.
		Transactional	Presence	Measured by evaluating if the component implements transactions in a way that allows to monitor and modify the history of actions done to the component.
Usability	Configurability	Functional configurability	Level	Measured with a value between 0-4. The component modules are analyzed to see if they allow initialization parameters or provide a way to modify the configurations after the module is initialized.
		Event configurability	Level	Measured with a value between 0-4. The component modules are analyzed to see if they provide relevant events that the application can apply callbacks to.
		Appearance configurability	Level	Measured with a value between 0-4. The effort for modifying the appearance of the component is evaluated. For example, are there configurations for modifying the component appearance with themes or do the developers need to implement component appearance modification with custom CSS. Additionally, is there documentation available that would help with modifying the component appearance
	Understandability	Documentation coverage	Ratio	Measured by dividing the number of functionalities documented with the total number of functionalities in the component. Functionalities may include installation, configuration options, methods, events and appearance configurations.
		Documentation quality	Level	Measured with a value between 0-4. Measured qualities may include for example, documentation readability and code examples.
		Demonstration coverage	Ratio	Measured by dividing the number of functionalities shown in demonstrations with the total number of functionalities in the component. The total number of functionalities may be extracted from the documentation available. Functionalities may include installation, configuration options, methods, events and appearance configurations.
Efficiency		Memory utilization	Integer	Measured by evaluating the memory utilization of the component when used in a minimal application.

Characteristic	Sub-characteristic	Attribute	Type	Measure
	Resource behavior	Disk utilization	Integer	Measured by calculating the total disk space required by the files in the component.
	Scalability	Processing capacity	Integer	Measured by testing the component scalability when exposed to different amounts of data by dividing the number of data units processed with the time required to process the data. Exact measurement needs to be specified component specifically.
Maintainability	Testability	Test suite provided	Presence	Measured by verifying if the component package includes a runnable test suite.
		Tests in a specific environment	Presence	Measured by verifying if the component testing documentation contains information about component tests done on different browsers or JavaScript frameworks.
Portability	Installability	Framework support	Ratio	Measured by dividing the number of supported framework specific versions of the component required by developers by the required number of framework supports of the component.
		Loading type coverage	Ratio	Measured as the ratio of module loading types supported by the component to the number of required loading types.
		Installation simplicity	Level	Measured with a value between 0-4. Measured by evaluating the effort required to install and update the component package on the project.
		Application to DOM	Level	Measured with a value between 0-4. Measured by evaluating the simplicity of the required HTML markup for the component to be applied to DOM.
	Reusablity	Modularity	Ratio	Measured as the ratio of modules provided by the component to the functionalities provided by the component
		Coupling	Ratio	Measured as the ratio of how many modules provided by the component require at least one another module provided by the component to function properly.
		Architecture compatibility	Level	Measured with a value between 0-4. Measured by analyzing how the component usage affects the supporting code structure implemented in the application because of the component.