Transfer, Measure and Extend Maintainability Metrics for Web Component based Applications to Achieve Higher Quality

Tobias Münch[a] and Rainer Roosmann[b]
University of Applied Sciences Osnabrück, Albrechtstraße 30, 49076 Osnabrück, Germany

Keywords: Maintainability, Measure, Web Components, Component-based-software-development, Web Applications.

Abstract: The last few years have seen an increased interest in web components composite W3C standard. These can be used without or with frameworks like Angular or React. Modern web applications get developed by the principles of component-based software development (CBSD). Therefore, a Web-Application is a composition of multiple web components, which are connected. In order to operate and continuously extend a web application successfully in the long term, the non-functional requirement of maintainability has crucial importance. This paper describes a model to collect, measure and compare maintainability in web components and web applications. These consist of object-oriented language (OOP) and a bound HTML-Fragment. Previous knowledge of maintainability gets extended to the interconnection between OOP and HTML-Fragments. Especially the coupling and cohesion between web components get analyzed. Through the developed model, the maintainability of web components can be specified in more detail. This allows web developers to analyze the quality of their web applications and reach a higher software-quality level.

1 INTRODUCTION

Modern web applications can be found in several domains like e-commerce, applications in the industry or even applications for consumers. JavaScript has reached the final frontier space by a web application for the SpaceX operating system (Schiemann, 2020). This system has to be maintainable in long a term lifecycle.

Web components, which are a composite standard of W3C from 2014, have currently an upswing by the alternative approach frameworkless development (Strazzullo, 2019). This approach focuses on minimizing external dependencies to increase maintainability and security. This paper aims to offer methods and tools to measure maintainability in web components and web applications.

Modern web applications are developed by the principles of component-based-software-development (CBSD) (Brown et al., 2002). Therefore, web applications consist of many composed web components. This principle matches the current development approach with frameworks.

The vast majority of the work in this area has focused on maintainability in object-oriented programming (OOP). For OOP, characteristics such as inheritance, coupling, cohesion and polymorphism have to be used in addition (Riaz et al., 2009). The well-known metric suites from Chidamber and Kemerer or Li can be used (Pressman, 2014). Heitlager et al. have connected these indexes to the previous ISO/IEC 9126 standard (Heitlager et al., 2007). These metrics can be transferred to web components because they are implemented partly in an OOP.

To date, no study has looked specifically at the maintainability of web components, especially for coupling and cohesion between them. Although, there is limited research investigating the interconnection between web components. This research leak leads to the questions:

RQ1: Which aspects of the maintainability index can be transferred to web components?

RQ2: How can maintainability be measured inside a web component, which consists of a Class and a HTML fragment?

RQ3: How can maintainability be measured in a web application, which is the composition of web components?

[a] https://orcid.org/0000-0001-9424-6201
[b] https://orcid.org/0000-0002-8625-2289

Münch, T. and Roosmann, R.
Transfer, Measure and Extend Maintainability Metrics for Web Component based Applications to Achieve Higher Quality.
DOI: 10.5220/0011511100003318
In *Proceedings of the 18th International Conference on Web Information Systems and Technologies (WEBIST 2022)*, pages 105-112
ISBN: 978-989-758-613-2; ISSN: 2184-3252

2 BASICS

In difference to native apps, **web applications** or web apps can be written once and be executed everywhere (Mikkonen and Taivalsaari, 2011; Biørn-Hansen et al., 2017; Heitkötter et al., 2012). A web application is a software application which is based on HTML5 standards and is developed by the principles of CBSD (Brown et al., 2002). Therefore, it consists of standard and custom web components (Strazzullo, 2019; Brown et al., 2002).

Web Components. can be described as custom, encapsulated and reusable components which can be used in web pages or applications. They are interpreted and executed in modern web browsers such as Google Chrome, Mozilla Firefox, Opera and Apple Safari. Web components are based on the web platform APIs Custom-Elements, Shadow-DOM, ES-Modules and HTML-Templates. Today, these APIs are included in the Web Hypertext Application Technology Working Group (WHATWG) standard (WHATWG, 2022).

The ISO/IEC international standard 25010 defines the **product quality model** (PQM) for applications, which is defined by the sub-categories functional stability, performance efficiency, compatibility, usability, reliability, security, portability and maintainability (Standard, 2022).

Maintainability. gets described by the PQM as "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" (Standard, 2022). It consists of the sub-characteristics: modularity, reusability, analysability, modifiability, and testability (Standard, 2022).

3 RELATED WORKS

Recent research shows that **web applications** can be developed client-side or server-side (Heitkötter et al., 2012). The traditional approach is server-side-rendering, where an application server renders a HTML document, which is executed in the web browser on a client (Mikkonen and Taivalsaari, 2011). These documents are less interactive than rich-components web apps. JavaScript and web components can be used to build a more interactive client-side document (Braga et al., 2012).

Web Components. have been widely explained in the last century. Krug and Gaedke have discussed the communication between different web components and proposed a communication bus in the paper "SmartComposition: Bringing Component-Based SoftwareEngineering to the Web" to simplify communication and thereby the composition of a web application (Krug and Gaedke, 2015).

In the paper "Self-contained web components through serverless computing", Ast and Gaedke have demonstrated how serverless computing and web components can be combined to create self-component serverless web components (Ast and Gaedke, 2017). They conclude that their approach can be applied to reduce time and costs while integrating web components.

Maintainability. has been systematically reviewed by Riaz et al., and they collected evidence on prediction and metrics (Riaz et al., 2009). Their study targets at software quality attributes of maintainability, as opposed to the process of software maintenance. The conclusion is that commonly used maintainability prediction models are based on algorithmic techniques with the sub-characteristics size, complexity and coupling.

Ghosheh and Black have proposed new design metrics measuring maintainability in heterogeneous web applications (Ghosheh et al., 2008). The Web Application Extension (WAE) for UML is used for these metrics to measure the design attributes: size, complexity, coupling and reusability (Ghosheh et al., 2008).

4 METHODOLOGY

This section outlines the specific methods used within our research. To describe the maintainability, sub-characteristics have to be measured with source code characteristics. Heitlager et al. have described the connection between them for for ISO/IEC 9126 (Heitlager et al., 2007). These results are transferable to ISO/IEC 25010.

4.1 Source Code Characteristics

Coupling. is defined by the metrics coupling between objects (*CBO*), afferent coupling (Ca) and efferent coupling (C_e) (Pressman, 2014; Anwer et al., 2017). The *CBO* defines "a count of the number of couples with other classes" for a class (Pressman, 2014). *CBO* has the theoretical basis that an object uses methods or variables from another object. The metric C_a measures how many other classes call a specific class (Anwer et al., 2017). C_e is defined as

vice versa as how many other classes are called from a specific class. Based on the results from Heitlager and ISO/IEC 25010, the coupling can be used for the system quality characteristic's modularity, reusability, analysability and testability.

Cohesion. is traditionally measured as the interrelatedness between portions of a program (Pressman, 2014). In the manner of OOP, it measures the relationship between methods and instance variables. Cohesion is measured by the term lack of cohesion in methods (LCOM). Chidamer et al. assumed a class with a n methods $M_1, M_2, ...M_n$ and each method uses a set of instance variables $I_1, I_2, ...I_n$. Let $P = I_i, I_j | I_i \cap I_j = 0$ and $Q = I_i, I_j | I_i \cap I_j \neq 0$ be defined. From there LCOM for a class is defined as

$$LCOM = |P| - |Q| \ \ if |P| > |Q|$$

$$LCOM = 0 \ \ otherwise$$

This metric is also known as LCOM1. It has been updated through several releases to the LCOM5 metric, which was introduced by Satwinder and Kahlon (Singh and Kahlon, 2011). LCOM5 is calculated by the following function:

$$LCOM5 = \frac{\frac{1}{a}\sum_{j=1}^{a}\mu(A_j) - m}{(1-m)}$$

a = number of attributes or instance variables

$\mu(A_j)$ = number of methods that access attribute A_j

m = number of methods in the class

$\mu(A_j)$ is summed over all the attributes $j = 1 - n$

Satwinder and Kahlon described the result range as "LCOM5 is normalized in the range 0 to 1 under the assumption that each attribute of a class is referenced by at least one method" (Singh and Kahlon, 2011). Based on the results of Heitlager et al., the coupling metric links to the system quality characteristic's reusability and analysability.

Complexity. can be measured by different methods, and it shows how complex the source code structure is (Coleman et al., 1994). The cyclomatic complexity (CC) is also known as the McCabe metric. CC can be applied to functions, modules, methods or classes of a program. Besides CC, the Cognitive Complexity (CogC) exists, which calculates how difficult a unit of code is to understand. The complexity metric links to the system quality characteristic's modifiability and testability. Heitlager et al. introduced this interconnection.

Size. of a software project is traditionally measured in Lines of Code (LoC) (Coleman et al., 1994). These are a part of the overall source code lines (SLOC), which can be divided into LoC, comment lines, and blank lines (Bhatt et al., 2012). Based on the results of Heitlager et al., the size metric links to the system quality characteristic's analysability and testability.

Code Duplication. measures how much code exists duplicated inside a software system codebase. Rieger et al. proposed the metrics a) LCC - the amount of copied code in the source files, b) LIC - lines of code copied file-internally and c) LEC - lines of code copied file-externally (Rieger et al., 2004). These metrics are based on the LOC of the software codebase. A higher LCC indicates a lower maintainability (Rieger et al., 2004; Sjøberg et al., 2012). Based on the results of Heitlager et al., the code duplication metric links to the system quality characteristic's modularity, analysability and modifiability.

Test Coverage. is also known as code coverage measures to which degree a source code is executed by a test suite. In practical code coverage tools, the typical code coverage metrics are statement, branch and path coverage, which are oriented at the control flow (Elbaum et al., 2001). It is measured how much per cent of the coverage type is covered (Elbaum et al., 2001). Based on the results of Heitlager et al., the test coverage metric links to the system quality characteristic's reusability, analysability, modifiability and testability.

4.2 Web Component

The measurement of complexity, size, code duplication and test coverage described above applies to web components. CC and CogC can be used to measure the complexity of source code written in JavaScript or TypeScript. All files written in JavaScript, TypeScript or HTML are evaluated to measure the metric size. The SLOC metric measures the lines of JavaScript or TypeScript and HTML files. Lines of comments are recognized in both languages. LOC only applies to JavaScript or TypeScript. The LCC, LIC and LEC metrics can be used in HTML Markup and JavaScript or Typescript to measure the characteristic code-duplication. Test-Coverage applies to the source code in JavaScript or Typescript. According to current research, test coverage is not transferable to HTML fragments.

Source code characteristics of coupling and cohesion only can be applied inside a web component if it contains multiple classes. Then CBO, C_a and C_e are

used to measure coupling between these classes, like in other oop languages. In the same way, LCOM can be used to measure cohesion.

4.3 Web Application

A web application consists of web components. Therefore the metrics size, code duplication and test coverage can be aggregated to apply to a web application.

A web application is a composition of web components, which in turn can contain other web components. These components are connected by HTML fragments or are initialized dynamically and inserted in the document object model (DOM). At present, there is no standard procedure or methodology for measuring the coupling and cohesion between web components in the DOM or HTML fragments. To measure these new types of coupling, input properties *CP* and used event-listener *CL* are used. The *CP* is defined analogical to the C_e because the DOM sets a property from another component. *CL* registers an event listener to a class instance, which is calling the method afterwards, so this can be described as a combination of C_a and C_e. Additionally, the existing metrics *CBO*, C_a and C_e measure coupling in the JavaScript or Typescript part of a web component.

The metric *CP* differs into static HTML bindings in markup like constant values CP_{const} and dynamic usage in JavaScript or TypeScript code CP_{dyn}. For example, implemented event listener often uses properties of the targeted web component. *CP* can be compared to C_a. All accesses on properties are computed into the metric *CP*. The event-listeners, which are bind by HTML markup CL_{const}or JavaScript CL_{dyn}, are counted into *CL*. Therefore, our metrics are

$$CP = CP_{dyn} + CP_{const}$$

$$CL = CL_{dyn} + CL_{const}$$

In order to compare two component coupling metrics, these metrics have to be relative. Therefore, n is introduced as the amount of other used components and this leads to:

$$CP_{rel} = \frac{CP}{n} \qquad CL_{rel} = \frac{CL}{n}$$

The following code snippet illustrates a used web component, which is accessed by some JavaScript code. This example is used to explain these new metrics.

```
<html>
 <body>
   <my-input type="text" onclick="sayHello()">
   </my-input>
   <script>
   function sayHello() {console.log("Hello")}
    const myInput = document
                    .getElementById("input");
    myInput.addEventListener('keyup',
         (event) => {
          console.log(event.target.value)
    });
   </script>
  </body>
</html>
```

In the web component my-input the property type is used ($CP_{const} += 1$) and an event-listener is registered to the 'click'-event ($CL_{const} += 1$). The function sayHello prints out the target value ($CP_{dyn} += 1$). Additionally, an event listener is registered to the 'keyup'-event in JavaScript ($CL_{dyn} += 1$). In the anonymous function of the listener, the value of myinput is used ($CP_{dyn} += 1$). These measurements lead to the result:

$$CP = 2 + 1 = 3$$

$$CL = 1 + 1 = 2$$

From this the following relative values CP_{rel} and CL_{rel} can be derived:

$$CP_{rel} = 3/1 = 3$$

$$CL_{rel} = 2/1 = 2$$

Like the other coupling metrics like *CBO CP* and *CL* states to how strong web components are coupled. If $CP = 0$ and $CL = 0$, then the web components are not coupled. The higher the value, the stronger the coupling.

4.4 Tools

In order to measure the code characteristics, SonarQube and embold are used. It analyses the source code characteristics complexity, size, code duplication and test coverage.

In order to have two measurement suites, embold is selected (https://embold.io/). It has similar features then SonarQube and is focused on DevOps.

Both tools can interpret test coverage, which is generated, for example, by the test framework jest. In our paper, we used free tooling for public repositories, which is common for open-source projects.

5 RESULTS

5.1 Case Study

European Regional Development Fund (ERDF) has funded Vet:ProVieh to reduce the use of antibiotics

in Lower Saxony. The project is an industry 4.0 software for veterinaries in the field of animal husbandry (https://github.com/Vet-ProVieh/). Through Vet:ProVieh, veterinary care is connected with antibiotics monitoring and action planning.

The Vet:ProVieh System is developed as a microservice architecture. A veterinary interacts with the system through a component-based PWA, which is structured by the domain-driven design (DDD) from Eric Evans. It follows the principles of frameworkless development and consists of many web components. These must have a high degree of maintainability to reduce such long-term maintenance costs. Especially the sub-characteristics modularity, reusability and testability are relevant for a growing software system.

The Progressive Web App (PWA) of Vet:ProVieh is structured in several packages. It consists of the professional domains of veterinary care (careplans), action planning (measures) and antibiotics monitoring (drugTreatments). The shared domain provides inputs, buttons, speech recognition for inputs and other user elements across all professional disciplines. Besides, the base packages vetprovieh-shared, vetprovieh-list, vetprovieh-select, formt-validation and vetprovieh-pager provide functionalities such as paging, interactive lists and select-inputs, annotations and form validation.

5.2 Measurement Results

5.2.1 Web Component

To measure maintainability in web components, we focus on the components vetprovieh-list and vetprovieh-detail (see RQ2). Vetprovieh-list was developed to create an interactive user interface for picking or selecting entries. How an entry will be rendered can be defined outside the component with a template. Vetprovieh-detail is a form frame to edit an object and send it afterwards to a RESTful-Webservice. It includes form validation, callbacks and a generic template. For the results, we recognize only source code in the lib folders by our measurement environment.

The measurement results for the vetprovieh-list are based on the introduced source-code characteristics (see table 1). It has been measured a CC of 102 and a CogC of 36. The component has an overall of 810 source code lines, which consists of 431 LOC and 283 comment lines (CL). There are 131 statements, 67 functions and six classes in seven files with zero code duplications. Test coverage of 88.5% has been achieved, which can be divided into a line coverage of 93.1% and a condition coverage of 79.6%.

Table 1: Comparison of measurement results between SonarQube and embold for vetprovieh-list.

Metric	SonarQube	Embold
SLOC	810	802
LOC	431	428
CL	283	283
CC	102	31
CogC	36	-
Test-Coverage	93.1%	93.1%
LCC	0	0

The second web component, vetprovieh-detail, has a complexity of CC 129 and CogC 56 (see table 2). It consists of 888 source code lines divided into 516 LOC and 267 comment lines. This component implements 203 statements, 88 functions and five classes in six files. No code duplications have occurred. Test coverage of 80.4% has been achieved, which can be divided into a line coverage of 88.2% and a condition coverage of 75.9%.

Table 2: Comparison of measurement results between SonarQube and embold for vetprovieh-detail.

Metric	SonarQube	Embold
SLOC	888	879
LOC	516	513
CL	267	274
CC	129	45
CogC	56	-
Test-Coverage	85.9%	85.9%
LCC	0	0

The metrics for coupling and cohesion can not be automatically measured with the selected tools. SonarQube has removed the metric LCOM4 because of many false-positive results (son, 2022). Embold should support *CBO* and LCOM for TypeScript and JavaScript according to the documentation, but in our case study, no results are measured (Embold, 2022). In an explorative study, Open-Source Projects on GitHub and NPM are examined to find different tools to measure these metrics. We found some repositories like cats from Thom Wright, but they are not usable yet (ThomWright, 2022).

5.2.2 Web Application

We examine the introduced web application Vet:ProVieh on the described metrics. Complexity, size and test coverage are measured in the same way as web components. The results are illustrated in table 3. Vetprovieh-app has a complexity of CC 1731 and CogC 606. It consists of 17096 source code lines divided into 10423 LOC and 3115 comment lines. This component implements 2619 statements,

1194 functions and 173 classes in 310 files. A small amount code-duplications have occurred, and there is refactoring needed. Test coverage of 40.3% has been achieved, which can be divided into a line coverage of 43.0% and a condition coverage of 29.4%.

Table 3: Comparison of measurement results between SonarQube and embold for the web application Vet:ProVieh.

Metric	SonarQube	Embold
SLOC	17096	19154
LOC	10846	12705
CL	4223	4584
CC	1731	752
CogC	606	-
Test-Coverage	39.8%	39.8%
LCC	544	1683

Like the results of web components, metrics for coupling and cohesion can not be automatically measured with the selected tools. Additionally, the complexity measurement from Kaur et al. can not be automatically calculated.

5.2.3 New Coupling Metric

To have results for our introduced metrics *CP* and *CL*, we have measured some web components without an automatic code analysis. We choose a part of the domain ‘measures‘, which is illustrated in Figure 1. Only interactions between custom web components are taken into account. The ‘vp-measure‘ is connected with ‘vetprovieh-notification‘, ‘measure-pdf-button‘ and ‘vp-objectives‘. In the source code, there is no event listener between these components, therefore $CL = 0$. There are several property usages, and the result is

$$CP = CP_{const} + CP_{dyn} = 1 + 3 = 4$$

These values can be described in a more detailed way. The connection to ‘vetprovieh-notification‘ can be described with the values $CP_{const} = 1$ and $CP_{dyn} = 0$, ‘measure-pdf-button‘ with $CP_{const} = 0$ and $CP_{dyn} = 1$ and ‘vp-objectives‘ with $CP_{const} = 0$ and $CP_{dyn} = 2$.

The seconds examined web component is ‘vp-objectives‘, which is connected to ‘vp-objective-item‘, ‘select-button‘ and ‘objective-modal‘. Coupling with ‘vp-objective-item‘ can be described with $CP_{const} = 0$, $CP_{dyn} = 8$, $CL_{dyn} = 1$ and $CL_{const} = 0$. The connection to ‘select-button‘ is shown up as $CP_{const} = 5$, $CP_{dyn} = 2$, $CL_{dyn} = 0$ and $CL_{const} = 0$. The usage of ‘objective-modal‘ can be described as $CP_{const} = 0$, $CP_{dyn} = 2$, $CL_{dyn} = 1$ and $CL_{const} = 0$. These details can be aggregated to:

$$CP = CP_{dyn} + CP_{const} = 11 + 5 = 16$$

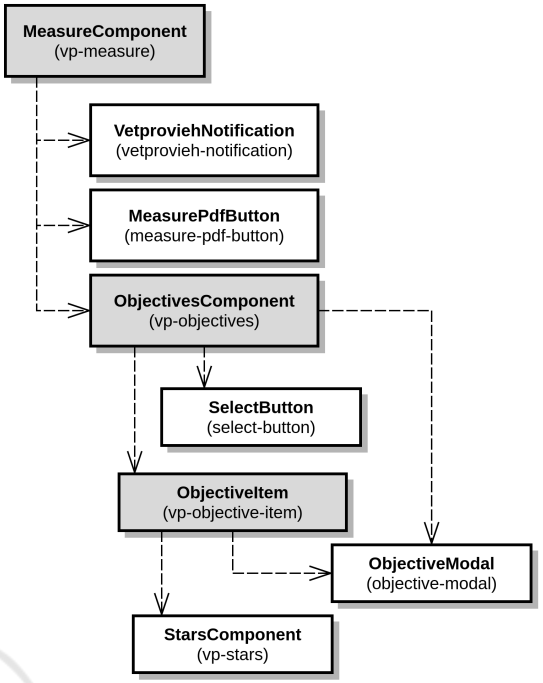

Figure 1: Evaluted web components of Vet:Provieh PWA.

$$CL = CL_{dyn} + CL_{const} = 2 + 0 = 2$$

The third examined web component is ‘vp-objective-item‘ with its interconnection to ‘vp-stars‘ and ‘objective-modal‘. The connection to ‘vp-stars‘ can be described with $CP_{const} = 1$, $CP_{dyn} = 1$, $CL_{dyn} = 1$ and $CL_{const} = 0$. The coupling to the component ‘objective-modal‘ is measured as $CP_{const} = 1$, $CP_{dyn} = 2$, $CL_{dyn} = 1$ and $CL_{const} = 0$. These detail results can be summed up to:

$$CP = CP_{dyn} + CP_{const} = 3 + 1 = 4$$
$$CL = CL_{dyn} + CL_{const} = 2 + 0 = 2$$

In order to compare these values, we have to compare them realtive to the amount of used components. On the component ‘vp-measure‘ has used three other components and so $CP_{rel} = CP/n = 4/3$. ‘objective-item‘ uses two other components and so $CP_{rel} = CP/n = 4/2 = 2$ and $CL_{rel} = CL/n = 2/2 = 1$. ‘vp-objectives‘ uses three other components and so $CP_{rel} = CP/n = 16/3 = 5,33$ and $CL_{rel} = CL/n = 2/3 = 0,66$ It can be concluded that ‘vp-objectives‘ is closer coupled to its sub-components then ‘vp-measure‘ and ‘objective-item‘.‘

5.3 Discussion

Our test measurement data of Vet:Provieh and its web components provide convincing evidence that

the code characteristics complexity, size, code-duplications and test coverage can be applied to the OOP part of web components. Additionally, the used tools SonarQube and embold have measured quite similar results for these characteristics. Contrary to our expectations, SonarQube and embold could not measure coupling and cohesion between web components. Therefore, the "RQ1 - Which aspects of the maintainability index can be transferred to web components?" can be answered. So the characteristic's complexity, size, code-duplications and test coverage can be applied to web components written in JavaScript or TypeScript. Size also includes HTML documents.

There are significant differences in the results for source code size between SonarQube and embold. These differences exist because embold calculates CC and LOC in another way than Sonarqube. Embold uses the original McCabe algorithm, and SonarQube uses a self-implemented algorithm for JavaScript measure CC. Their algorithm counts function, if statements, conditional expressions, loops, switch case statements and throw catch statements.

To answer the "RQ2 - How can maintainability be measured inside a web component, which consists of a class and a HTML fragment?" it can be concluded that metrics for coupling, cohesion, size, complexity, code duplication and test coverage can be used for web components. Still, only size, complexity, code-duplication and test coverage are currently automatically measurable. We calculated the results on our own by using a proposed algorithm to measure coupling between web components. This algorithm can be used to measure the connection between classes and their coupling inside a markup language like HTML. We would encourage researchers to examine the composition between markup language and object-oriented language in more detail to provide a static code analysis to measure our proposed metrics.

We can transfer the methods from the first two research questions to web applications to measure maintainability. Therefore, we can connect the results of RQ1 and RQ2 to answer the "RQ3 - How can maintainability be measured in a web application, which is the composition of web components?". The relationship between composed web components can be measured by the results of RQ2 and be aggregated to a result for a web application like our results have shown. Our results cannot check the sub-aspect CP_{dyn} because it was not used in the Vet:ProVieh project. Our case study has a small scope and needs to be verified through empirical research. In addition, the results of RQ1 can be transferred in precisely the same way.

We can not measure coupling and cohesion automatically inside web applications with the selected tools. After explorative research for other instruments or GitHub repositories, none has been found.

6 CONCLUSIONS

The main conclusion that can be drawn is that well-described traditional metrics complexity, size, code-duplications and test coverage can be used for the OOP part of web components and web applications. These can describe the quality characteristic maintainability of the ISO/IEC 25010 well. We used tools like SonarQube and embold to collect data for these metrics.

The code characteristic coupling has been extended to measure the connection between web components inside the DOM because web application links their components in this way. We propose a new metric, how to collect data for this extension, to describe it in a more detailed way. The usage of this metric has to be proven in multiple open source projects because our experiment has only a small scope. To apply our metrics to a greater area, they have to be evaluated automatically.

Current tools like SonarQube can not automatically measure coupling and cohesion between or inside web components. This lack of measurement and the algorithms below is a possible research gap in web development.

Therefore, the measurement of maintainability in web applications is not yet up to the standard as in other high-level languages such as Java or C#.

7 FUTURE WORK

Future research should consider the potential of measuring the complexity of composing web components inside a web component more carefully, for example, to measure the cohesion between web components inside a web application. Especially we can not recognize dynamic DOM manipulations trivially inside static code analysis.

Furthermore, we are looking forward to define how our proposed metrics can be aggregated in web applications to have an overall score.

Additionally, the metric suites from Chidamber and Kemerer are not implemented for TypeScript or JavaScript, which are implemented in an OOP style. Therefore additional research and implementation are needed to transfer the metric suite to web applications and its components. This includes further questions

about how static and dynamic code analysis can be used to measure coupling and cohesion in web applications and web components.

This is necessary because, with such tools, web developers can increase the maintainability of web applications, so long-term and high-reliability applications can be developed for the industry.

In Addition, this automation enables an empirical study to validate our proposed metrics in a more detailed way. For this study, several open source projects have to be selected.

REFERENCES

(2022). [SONAR-4853] Remove support of LCOM4 - SonarSource. https://jira.sonarsource.com/browse/SONAR-4853. [Online; accessed 5. May 2022].

Anwer, S., Adbellatif, A., Alshayeb, M., and Anjum, M. S. (2017). Effect of coupling on software faults: An empirical study. In *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*, pages 211–215. IEEE.

Ast, M. and Gaedke, M. (2017). Self-contained web components through serverless computing. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 28–33.

Bhatt, K., Tarey, V., Patel, P., Mits, K. B., and Ujjain, D. (2012). Analysis of source lines of code (sloc) metric. *International Journal of Emerging Technology and Advanced Engineering*, 2(5):150–154.

Biørn-Hansen, A., Majchrzak, T. A., and Grønli, T. M. (2017). Progressive web apps: The possiblewebnative unifier for mobile development. *WEBIST 2017 - Proceedings of the 13th International Conference on Web Information Systems and Technologies*, (Webist):344–351.

Braga, J. C., Damaceno, R. J. P., Leme, R. T., and Dotta, S. (2012). Accessibility study of rich web interface components. In *ACHI 2012, The Fifth International Conference on Advances in Computer-Human Interactions*, pages 75–79.

Brown, A., Johnston, S., and Kelly, K. (2002). Using service-oriented architecture and component-based development to build web service applications. *Rational Software Corporation*, 6:1–16.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.

Elbaum, S., Gable, D., and Rothermel, G. (2001). The impact of software evolution on code coverage information. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 170–179. IEEE.

Embold (2022). Metrics overview - embold help-center. https://docs.embold.io/de/metrics/. Accessed: 2022-04-23.

Ghosheh, E., Black, S., and Qaddour, J. (2008). Design metrics for web application maintainability measurement. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 778–784. IEEE.

Heitkötter, H., Hanschke, S., and Majchrzak, T. A. (2012). Evaluating cross-platform development approaches for mobile applications. In *International Conference on Web Information Systems and Technologies*, pages 120–138. Springer.

Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE.

Krug, M. and Gaedke, M. (2015). Smartcomposition: bringing component-based software engineering to the web. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, pages 1–4.

Mikkonen, T. and Taivalsaari, A. (2011). Apps vs. Open Web: The Battle of the Decade. http://www.w3.org/TR/offlinewebapps/.

Pressman, R. S. (2014). *Software engineering: a practitioner's approach*. McGraw-Hill Education.

Riaz, M., Mendes, E., and Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. In *2009 3rd international symposium on empirical software engineering and measurement*, pages 367–377. IEEE.

Rieger, M., Ducasse, S., and Lanza, M. (2004). Insights into system-wide code duplication. In *11th Working Conference on Reverse Engineering*, pages 100–109. IEEE.

Schiemann, D. (2020). JavaScript Reaches the Final Frontier: Space. https://www.infoq.com/news/2020/06/javascript-spacex-dragon.

Singh, S. and Kahlon, K. S. (2011). Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–10.

Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., and Dybå, T. (2012). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156.

Standard, I. I. (2022). Systems and software engineering — systems and software quality requirements and evaluation (square) - system and software quality models - iso/iec 25010:2011(e), vol. 2011. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en. Accessed: 2022-04-01.

Strazzullo, F. (2019). Frameworkless front-end development.

ThomWright (2022). cats. https://github.com/ThomWright/cats. [Online; accessed 5. May 2022].

WHATWG (2022). Dom - living standard - last updated 22 march 2022. https://dom.spec.whatwg.org/. Accessed: 2022-03-27.