Kevin Haslinger, BSc

Evolution of Software Measures - a Small-Scale Case Study Using Two Web Frameworks

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dr.techn. Roman Kern

Institute for Interactive Systems and Data Science
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Roman Kern

Graz, January 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

As the complexity of a software projects rises it can become difficult to add new features. Additionally to the maintainability, other quality attributes such as reliability and usability may suffer from the increased complexity. To prevent complexity from becoming an overwhelming issue we use principles of good programming and reside to well known software architectures. We often do so, by choosing to use specific frameworks. However, we can only subjectively judge whether or not the usage of a specific framework resulted in less perceived complexity and an improvement in other quality attributes.

In our work, we investigated the applicability of existing software measurements for measuring desired quality attributes and their applicability for framework comparison. We chose a set of quantitative software measurements which are aimed at specific quality attributes, namely maintainability and flexibility. Additionally, we used well established software measurements such as McCabes Cyclomatic Complexity [44] and Halsteads Metrics [32] to measure the complexity of a software.

By developing the same application using two different web frameworks, namely ReactJS and Laravel, over a set of predefined 'sprints', each containing a specific set of features, we were able to investigate the evolution of different software measurements. Our results show that some of the measurements are more applicable to the frameworks chosen than others. Especially measurements aimed at quantitative attributes of the code such as the coupling measures by Martin [43] and the Cyclomatic Complexity by McCabe [44] proved particularly useful as there is a clear connection between the results of the measurements and attributes of the code. However, there is still the need for additional work which focuses on defining the exact scale each of the measurements operates on, as well as need for the development of tools which can be used to seamlessly integrate software measurements into existing software projects.

Contents

Contents

1. Introduction

Modern software is built to last. Expectations on longevity lead to the need to develop maintainable systems. Extensive research by Rajiv D. Banker, Srikant M. et. al. [4] has shown that the complexity of a system has significant impact on its maintainability and therefore on the overall costs of the development process. We use modern software architectures to reduce the complexity of our systems. Al Sharif et. al. [2] have shown a relation between software architecture and complexity which justifies the efforts put into the architecture. Based on the principles of good programming and design patterns several frameworks have evolved. Their goal is to provide developers with an established and well designed software architecture and therefore increase productivity and maintainability. However, since there are a lot of different frameworks out there, each of them based on different principles, it is difficult to decide which framework fits best. Additionally, the question if these frameworks actually succeed in reducing complexity or increasing maintainability remains unanswered.

The solution is clear: we need a way to measure the attributes of the software we want to improve and then, through comparisons, conclude whether or not the measured attributes decreased or increased. This notion of software measurements is nothing new and dates back to two of the most well known software measurement works of Boehm et al. [11] and McCall et al. [45] which both came up with a list of software characteristics which break down to a comprehensive index of empirical measurements. The interest in such kind of measures is more than understandable. They allow us to estimate future effort and base our decisions on empirical data. This can help in the planning of business decisions and prevent fatal outcomes. Further, we can use them to measure the reliability of software; or even software modules. A practical application of software measures by Clark et al. [17] demonstrates that measuring software complexity can help with detecting potentially defect and risky modules in critical software like autonomously driving

cars. Additionally, using software quality measurement we can asses the impact of a software project on a company's success. Another, more or less criticized use of software measures is proposed and investigated by Coleman et al. [18] who used their Maintainability Index to compare two different software projects and support 'make or buy' decisions. More traditional applications suggest the usage of software complexity measurements to verify code coverage of existing testing suites [64].

When looking at framework comparisons, most of the time a comparison based on qualitative attributes can be found. Qualitative attributes are often measured subjectively, by evaluating a given piece of software and deriving values for measurements based on personal opinion. According to Fenton and Bieman [26, p. 66] this kind of subjective measurements can still be useful, since they provide us with information about the overall picture. When asking a team of developers to rate different modules of a software project based on their subjectively perceived difficulty of working with a given module, we can use the information gathered to identify modules which are most likely to be difficult to maintain. However, when using subjective measures for decision making, we must not forget about their flaws.

Subjective measures are aimed at qualitative attributes. The interpretation of those qualitative attributes and therefore the values of their measurement are solely based on the persons which conduct the measurement. Therefore, different groups of people will most likely lead to different results in measurements. This makes it hard to find a common consensus, since we would need to reproduce the exact same conditions multiple times to validate a given measurement model. Traditional measurements, such as the measurement of length, do not face this kind of problem.

Given a measuring tape, a person can easily measure the length of a wooden board. If we ask another person to measure the same wooden board, the results of both measurements will be the same. Unlike the subjective measures which are based on qualitative attributes, which are interpreted and rated based on personal opinion and experience, the measuring tape measures a quantitative attribute of the wooden board, which is independent to the person measuring it: the length. The values received through measurement of quantitative attributes do not change if conducted by different persons.

In this thesis out of work the evolution of traditional and quantitative software measurements during the development of a small scale software project built in two different web frameworks (namely ReactJS and Laravel) is observed. We investigate whether or not software measurements such as the Cyclomatic Complexity by McCabe [44], the Maintainability Index by Coleman et al. [18] and the coupling measures by Martin [43] evolve in a similar fashion for both frameworks. The goal of this thesis is to use a set of quantitative software measurements to compare two different web frameworks with each other. We investigate the applicability of quantitative software measurements for the chosen frameworks in general, but also their applicability for framework comparison. The resulting values and the meaningfulness of conclusions drawn from them will be described.

The rest of this thesis is structured as follows. Chapter 2 will present an overview of modern and traditional software measures. It will further cover definitions needed to understand measurement and software measurement in general. The mathematical foundations of measurement theory are presented and used to provide insight in the ideas behind the meaningfulness of measurements. This includes the discussion of several fundamental works on software measurement such as the work by Zuse et al. [66] and Fenton and Bieman [26] as well as an investigation on their differences. Chapter 3 contains information about related work where software measures were used for framework comparison. The fourth chapter covers information about the software used during the experiment such as the frameworks involved and software used to conduct the software measures. Chapter 5 describes the software developed and the development process as well as the different architectures expected impact on the software measurements. Chapter 6 covers the software measures used during this experiment in more detail. Chapter 7 presents the results of the software measures. Additionally, every result is investigated in more detail and interpreted. Possible conclusions and the applicability of the selected software measures for framework comparison is discussed. The thesis ends with a conclusion in chapter 8 and a guideline for future work.

2. Background

This section will discuss existing software measurement techniques. Software measurements have many applications. They provide us with important and empirical data which can be used to make informed decisions. Further they allow us to measure the quality of our software and therefore compare it to other software [21]. Other applications include the creation and validation of test suites based on code coverage [64, 48, 23]. One of the applications that has gotten a boost in attention in latest years and will be discussed later in this chapter is software defect prediction [17, 35, 52].

2.1. Software Measurement

This section will discuss the definition of software measurements by Zuse et al. [66, p. 25]. In general measurements map somehow observable attributes to the numerical domain. When we finally end up with some values for our measures we can then use statistics and mathematical operations to derive further knowledge or refine our data. We can then interpret the collected numbers in some meaningful way.

However, as Zuse et al. [66] point out that numbers alone are of little help. Without a reliable way to interpret and use the information gained from those numbers they have little value. We also somehow need to verify the connection between the attributes we wanted to capture and the numbers we measured. This together with the fact that complexity has always been a loosely defined term are the two biggest problems of measuring the complexity in software projects.

The next section will cover different applications of software measurements, while common problems with measurements and the definition of complexity will be discussed later in this chapter.

2.1.1. Applications of Software Measurement

In this section the work by Zuse et al. [66, p. 28] focused on defining software measures is summarized and supplemented with findings and recent works discussed later in this thesis. Zuse et al. describe 6 suitable applications of software measurements:

1. Cost estimation
2. Productivity Measures
3. Reliability models
4. Computational / Algorithmic complexity
5. Complexity measures
6. Quality models

Cost estimation

One of the more practical ways of using software measurements is to use them to predict the amount of work that is needed to create the software (the costs of developing the software). An example of such measurements would be function points. They have found wide application in the field of software measurements. Function points as the name suggests assign numerical values to functions. These values are calculated to represent the "effort" needed or put into specific functions. By calculating the function points for each function the software will need, one can estimate the effort needed and propose a contract based on empirical measurements instead of rough guesses. As most other software measurements, function points have been both criticized [37, 41] and praised alike [36].

Among other problems, the subjective nature of function point estimation is mentioned the most. Inexperience in applying the function point method can lead to vastly different estimations as Low, Richards et al. [41] show. However, Low, Richards et al. further show that even when dealing with experts the comparison of

their estimates can lead to 30% variation. Contrary to these findings an extensive study by [36] consisting of multiple function point counts of 111 different systems has shown that between pairs of estimators the variations between their estimates lies in the range of $\pm 10\%$.

Boehm et al. [12] present solutions for cost estimation of software products which they base on knowledge used in the field of economics. They describe the uncertainty of software projects as decreasing function over the software life cycle, whereas the uncertainty is the highest at the start of the project and steadily decreases as the software is built and requirements are refined. They argue that it is important to calculate the overall benefit of decisions during the earlier phases and that for software engineering in particular prototyping or in other words 'buying of information' [12] has shown to be successful at reducing uncertainty. For practical applications they proposed the COCOMO cost estimation model [12, 9] which was later revisited and expanded to COCOMO II. [9, 10]

Productivity Measures

In the literature revised the applications of cost estimation and productivity have been used synonymously. For this reason the work of Boehm [12] discussed in the last section can be seen as part of productivity measures. However, a direct example would be the work by Gordon, Halstead et al. [29] where the software measures defined by Halsteads Software Science [32] were used to calculate the effort needed to comprehend or build the software that is being measured. Additionally to the effort required to write the software, we can consider the effort required to keep the software up to date. A well known and again often praised and criticized method to measure the maintainability of a software system is the so called Maintainability Index proposed by Coleman et al. [18].

The goal of the Maintainability Index is to capture software maintainability by combining different, well established software measurements into a single number. Coleman et al. argue this works because each of the three measures used captures different factors which are deterrent to software maintainability. In their work they suggest the use of widely accepted measures. In particular Coleman et al. use Halsteads [32] measure for effort, McCabes extended Cyclomatic Complexity [44]

and Lines of Code. It has been found that the Maintainability Index captures many intuitive code quality attributes. An example would be that the Maintainability Index is likely to decrease if a module is rewritten or restructured [18]. A lower Maintainability Index is justified since all the developers who have worked with the previous version of this module now have to learn and understand the new structure.

The big benefit of the Maintainability Index is the ability to calculate it for multiple modules of the same software project. This allows to compare different modules of the system and can help to identify error prone or hard to maintain modules. Also, if the Maintainability Index is calculated for every major change in the system for each sub module, it is possible to identify modules which have a low tendency to change. These modules are likely to form a set of reusable components or may even be used as reusable libraries.

In their initial work Coleman et al. [18] support their software measure by looking at different use cases in practice. In particular their system was used to aid Hewlett-Packard.

The Maintainability Index of an overall system was calculated to gain an overview. As this provided little information overall, in the next step they calculated the Maintainability Index for individual sub modules. They found that the modules the developers described as 'cumbersome to work with' also had a low Maintainability Index. The Maintainability Index was further used to compare an in-house solution with a third party solution. Since the in-house solution had a lower Maintainability Index, HP decided to go with the third party solution. Coleman et al. argue that their Maintainability Index can be helpful when it comes to make-or-buy decisions. However, major concerns with their software measure have arisen over the years.

When using the Maintainability Index in a project, developers have no insight on how to directly affect it [65]. It is calculated by some magic formula without obvious ways on how to affect the measure. Kuipers et al. [40] acknowledged this problem and propose a modified model for the Maintainability Index which is aimed at providing a clear insight on how the measure is calculated and how it can be affected by changes in the code. A more recent attempt at providing a measurement system aimed at code-smells [28] can be found in [65]. Wu et al. [65] focused on

providing developers with a software measurement system which provided a clear mapping between the measurements taken and observable problems in the code and architecture.

Reliability Models

Zuse et al. [66, p. 28] describe Reliability models as

> [...] statistical models for predicting mean time to failure or expected failure interval. [66, p. 28]

These models are not exclusive to software development.

Computational and Algorithmic Complexity

The computational or algorithmic complexity of a program or piece of software tries to capture the efficiency with which a certain problem has been solved [66, p. 28]. When measuring complexity, researchers often aim at minimizing the complexity of the software measured. However, as shown by McCall et al. [45] there is an inverse relationship between efficiency and maintainability which means increasing one will have negative effects on the other. It is not surprising that combining both needs - high efficiency with low complexity - is a difficult task. Some work in this direction was done by Oliveira et al. [51] who tried to apply various kinds of software measurements to embedded systems.

Their experiment was aimed at finding a compromise between modern software designs and the requirements of embedded systems. The experiment confirmed the already mentioned inverse relationship between efficiency and complexity. Specifically Oliveira et al. [51] have shown that for their tested system McCabes Cyclomatic Complexity [44] was inversely proportional to the measured physical performance. Oliveira et al. conclude that using software measures in the context of embedded systems can help find the correct trade-off between performance and other important goals which correlate with the investigated quality measures such as time-to-market.

Complexity Measures

Measurements of this category focus on specific attributes of the software. The most prominent examples for such attributes would be the control structure of the program as well as simple measures like lines of code. These measures have gained significant attention, with the most well known and used examples being the complexity measure of McCabe [44] and Halsteads theory of Software Science [32]. Both works have been validated by countless empirical studies over the years and will be discussed in detail later in this chapter.

Quality Measures

Models to measure the quality of software often consist of a wide spectrum of measurements. Early examples for efforts put into measuring software quality are the heavily cited works of McCall [45] and Boehm [11]. Their works are the most well known works on software quality measures and both provide a comprehensive list of measurements. A more recent and also well known approach is presented in the Information System Success Model by Delone and McLean [22, 21]. Their attempts at measuring software quality focuses on more than just code quality attributes. Instead, they additionally measure the entire ecosystem of the software. Unlike most other measures discussed in this chapter, which mostly rely on the code or control structures of the program, these newer quality models include measures aimed at the effort needed to keep the software running as it should and at the people actually using the software.

The next section will cover Software Quality in more detail, while the later sections focus on complexity and how to measure it in the context of software projects.

2.2. Software Quality

A well known measurement for software quality has been developed by McCall, Richards et al. [45]. In their work they acknowledge that quality itself consists of multiple aspects which they call ‘quality factors’. They define specific criteria for each of those quality factors while the decision if those criteria are met are based on

precisely described measurements. Each criteria has its own set of measurements for which McCall, Richards et al. describe how to measure them.

The hierarchical structure of software quality has been adopted by different studies. After McCall, Richards et al. the 'Information System Success Model' by DeLone and McLean [21] gained huge attention in the field of software quality. With their extensive research they identified the reoccurring hierarchical nature of quality. They found that back in 1948 Shannon and Weaver [54] had already indirectly identified the hierarchy of software system quality. DeLone and McLean extend the existing model consisting of a technical level, semantic level and level of effectiveness to their own model consisting of six dimensions: System Quality, Information Quality, Use, User Satisfaction, Individual Impact and Organizational Impact. They successfully created a more accessible way to measure software quality, which was widely accepted and further researched. Ten years later DeLone and McLean published the next iteration of their Information System Success Model [22].

After having reviewed several papers critiquing, applying, and validating their proposed model they present a new model that also incorporates Service Quality and replaces Individual and Organizational Impact as Net Benefits. The new model is aimed at providing a more precise and applicable Information System Success Model for e-commerce systems. DeLone and McLeans Information System Success Model (D&M IS-Model) still sees use today. Gucegliogl u, A. Selcuk et al. [30, 31] adopted the D&M IS-Model to business processes. Taking the dimensions of the D&M IS-Model Gucegliogl u, A. Selcuk et al. use measurements applicable for business processes to evaluate their quality and benefits, as well as providing a framework to compare business processes. Among many other similarities between business processes and software solutions they draw attention to the complexity measurement. They use cyclomatic complexity [44] to measure the complexity of a business process where as the amount of decisions in the process defines the complexity of the process.

In conclusion software quality measurements focus on the whole ecosystem surrounding and affected by the system. These measurements inspect the system itself (Information Quality, System Quality), the people making sure the system is available and working as it should (Service Quality), the users which use the

software (User Satisfaction, Use, Intention to Use) and the company that benefits from the software in some way (Net Benefits). Complexity measurements on the other hand are quantitative measures solely based on the software itself.

2.3. Complexity

Before we go into how to measure complexity, we have to agree on a definition of what it is. Complexity has always been a loosely defined term to allow different interpretations based on the current field. Edmonds et al. [25] present an extensive definition of complexity aimed at a broad field of vastly different disciplines. The overall definition is given by

> Complexity is that property of a model which makes it difficult to formulate its overall behavior in a given language, even when given reasonably complete information about its atomic components and their inter-relations. [25]

Following this definition Edmonds et al. illustrate that when looking for complexity we are always looking at a model - or in other words a system - consisting of different components and their inter-relationships. Edmonds et al. then define complexity as the lack of knowledge about the overall system, even though we have sufficient information about its single components.

Zuse et al. [66, p. 34] dedicate a whole chapter of their book to verbal definitions of the term complexity. Among many others they cite a definition by Sullivan et al. [60] which provides us with a definition for the psychological part of complexity:

> In general usage, complexity denotes the degree of mental effort required for comprehension. [60]

After looking at many different definitions of complexity one can easily depict a common relation: Complexity increases the more components with relationships to other components are added to a system. This means that an increasing amount of components and/or relationships will lead to greater complexity. Whether we are dealing with a system consisting of a big number of trivial components with little relationships between each other or with a system consisting of a small number

of highly interrelated components; both cases lead to a perceivable increase in complexity. With these observations in mind we can now work on a definition for software complexity.

2.3.1. Software Complexity

A definition for software complexity aims to further specify complexity in the field of software development. In their work Curtis et al. [20] give a definition for software complexity based on their earlier research:

> Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software. [19]

This general definition incorporates the information of the previous section. The preceding definition described complexity as the difficulty to understand the overall behavior of a system, which is equivalent to Curtis idea behind expending resources while interacting with the system. This definition also puts emphasis on relationships between the software and other systems (which also includes other components). Curtis explains that complexity has no meaning in an isolated system as it only matters during the interaction between the software and another system *'[...] such as machines, people and other software packages'* [20]. However, this is again a broad and abstract definition of complexity. Curtis et al. come to the conclusion that a software measurement has to focus on a certain software characteristic and define empirical measurements suitable to capture this software characteristic. One of those characteristics that has gained a lot of attention is the control structure of the software.

A well revised example for such a control structure based complexity measurement is McCabes cyclomatic complexity [44].

2.3.2. Cyclomatic Complexity

McCabes cyclomatic complexity is based on graph theory. It uses the cyclomatic number defined by Berge [7] by looking at the control flow graph of programs. As defined by Berge [7] the cyclomatic number can be calculated as $v(G) = e - n + p$ where n are the vertices, e the edges and p the connected components. Since this mathematical approach is not always suitable McCabe has put effort into describing simpler models that lead to similar if not identical results for cyclomatic complexity. His first simplification does only require visual inspection of the control flow graph. Based on Eulers formula it is possible to determine the complexity of a plane control flow graph by counting its regions.

The second method proposed by McCabe is even more practical. This method does not require a control flow graph at all and instead suggests to count the amount of predicates or in other words statements like:

```
IF <predicate> THEN <action>
```

When considering compound statements such as

```
IF <predicate> AND <predicate> THEN <action>
```

McCabe [44] notes that

> [...] it has been found to be more convenient to count conditions instead of predicates when calculating complexity. [44]

Cyclomatic complexity has found various applications in software quality research since then. Mohamed et. al. [49] used cyclomatic complexity to assess computer science students programming skills.

Menzies et. al. [46] have shown that cyclomatic complexity can be used to derive Defect Detectors; data mining tools which find modules that have a high chance of being defective based on static code attributes. In their paper they also show that cyclomatic complexity leads to a better defect detection rate than weaker

measurements like lines of code.

Similarly Clark et al. [17] used cyclomatic complexity to assess reliability and maintainability of critical systems. Based on their measurements they managed to identify modules of an autonomous driving software which were most likely to require refactoring. However, another important aspect of cyclomatic complexity is its usage in testing software.

Based on McCabes cyclomatic complexity Watson et al. [64] proposed a method to assess the quality of testing; the structured testing criterion. The concept of the structured testing criterion is that we can choose a set of paths such that

> […] any additional path through the module's control flow graph can be expressed as a linear combination of paths that have been tested. [64]

Structured testing follows the idea to write test cases such that all paths of the control flow graph of a program are visited. To achieve this Watson et al. [64] postulate the Base Path Method.

A base path is a path in the control flow graph of the program which is linearly independent from all other paths in the program. The first base path can be chosen freely; however, Watson et al. recommend to choose the one that is the most relevant to the software at hand. From this base path new paths are generated by simply taking other paths on conditional branches. In the end you have a set of basis paths. All other paths of the control flow graph can be represented as a linear combination of these basis paths. The important connection between this method and cyclomatic complexity is that the minimum amount of paths needed for the base path method to cover all paths in the control flow graph is equal to the cyclomatic complexity number [64]. This means that given an amount of linearly independent paths equal to the cyclomatic complexity of the associated control flow graph, we can create test cases for those paths and reach high code coverage. Since all of these paths cover independent parts of the software we can test most of the program by using a small number of test cases.

An example for the Base Path Method is the toolset developed by Deng and Wang [23]. Deng and Wang constructed a graph by defining URLs as nodes and links between URLs as edges. Now, using the base path method discussed in the last paragraph they generated a set of paths along this graph while also determining the inputs these URLs might need. As an additional step they use the information gathered by the base path generation to then generate test cases based on the paths chosen and inputs needed.

Mirshokraie et al. [48] combined traditional mutation testing with cyclomatic complexity to overcome downsides of mutation testing. Similar to Clark et al. [17], Mirshokraie et al. used cyclomatic complexity to determine which paths are more error prone and lead to different behavior in the underlying application. Mutants for those paths cover the most error prone parts of the application and therefore provide test cases aimed at critical sections of the software.

2.3.3. Halsteads Complexity Measure

Another traditional complexity measure is the method presented by Halstead [32]. His work was a pioneer attempt to combine interdisciplinary knowledge into a concept of empirically measurable complexity for software. He used psychological knowledge about the human brain and human comprehension to derive software measurements which aim to estimate required mental effort, software size and predict error proneness. Back then and up to today various empirical studies have confirmed the strong correlation between the measurements presented by Halstead and mean amount of bugs in a program [27, 6, 35, 52].

Halstead has been involved in various research based on his now famous Software Science. In [29] Gordon and Halstead present a correlation between Halsteads measures and programming time (time required to write a program). This correlation also gained attention and was later empirically tested and supported by an experiment by Sheppard et al. [56] and still sees use today. A modern example is presented by Chang et al. [15] which apply Halsteads measures to programs created using the visual programming language Scratch. They measure Halsteads measures for Scratch programs and later validate the results using process data ultimately

confirming the validity of Halsteads methods for the Scratch programming language or more generally for visual programming tools.

2.4. Object Oriented Measures

Object oriented software development is a commonly agreed and widely used paradigm in modern software development projects. As R. Martin describes [43] this is due to our believe in the positive impact of object oriented design on our software project. We hope for better reusability, maintainability and robustness. However, as R. Martin points out just using object oriented design will not magically turn every software into a high quality product. It is not object oriented design itself which brings a lot of these benefits; it is best practices associated with object oriented design that do. To shine some light on this misconception R. Martin developed different 'Design Quality Metrics' specifically for object oriented software projects.

2.4.1. Robert Martin's Design Quality Metric

Martin's measure is based on an intuitive idea: Classes which are highly interdependent to other classes are hard to change. For example if there is class A and 10 other classes which depend on class A, it is very likely that we have to change some if not all 10 classes once we change class A. Additionally, if we wanted to use one of the 10 other classes in another system we would probably have to bring along class A (and potentially all the classes it depends on). Both of these examples make it obvious that classes which are independent are more flexible and more reusable.

Based on this intuitive idea Martin [43] came up with three different measurements aimed at quantifying the inter-dependability of classes. However, since there are often certain 'clusters' of classes which are highly inter-dependent between each other but otherwise independent to the outside, Martin added an additional layer. He calls these highly inter-dependent but otherwise independent clusters of classes 'categories' and defines his measures as follows:

Afferent Coupling also written as C_a is '*the number of classes outside this category that depend upon classes within this category*' [43]

Efferent Coupling also written as C_e is '*the number of classes inside this category that depend upon classes outside of this category*' [43]

Instability or simply I is defined as $\frac{C_e}{C_a+C_e}$. This takes a value between 0 and 1 where 0 means the class is absolutely stable and 1 means the class is absolutely instable

Martin distinguishes highly stable classes by the fact that classes which have a lot of dependents are hard to modify. In other words, classes with a high Afferent Coupling are unlikely to change since a change of those classes would likely lead to a lot of changes in the dependent classes. On the other hand classes with a high Efferent Coupling are labeled as highly instable. This is due to the fact that changes in any of the dependencies might cause changes in the current class.

Unlike the measures discussed in the earlier section which aim to quantify the psychological effort needed to work with a given piece of software, Martins coupling measurements are attributed to measure the flexibility and robustness of the system as well as its components reusability.

2.4.2. Object Oriented Measures by Chidamber and Kemerer

Chidamber and Kemerer [16] present a set of object oriented measures based on a solid theoretical and mathematical foundation. They use the work of Bunge [13, 14] to formally define important attributes of object oriented designs.

Desirable Attributes in Object Oriented Design

This section will cover basic definitions of Chidamber and Kemerer [16] which are then used to identify and formally describe desirable attributes in object oriented software.

Chidamber and Kemerer define an object X as

$$X = \langle x, p(x) \rangle \tag{2.1}$$

where x is simply an identifier f.e. the class name and $p(x)$ is the set of properties of X. The properties in this context consist of the class methods and instance variables, notionally:

$$p(x) = \{M_x\} \cup \{I_x\} \tag{2.2}$$

where $\{M_x\}$ are the methods of class X and $\{I_x\}$ are the instance variables of class X. Using this definition Chidamber and Kemerer classify important aspects of object oriented design as follows:

Coupling measures the interdependence of classes between each other. High coupling is considered an issue because it is hard to change classes with high coupling since changes to these classes are likely to cause changes in other classes as well. Chidamber and Kemerer use the ontological definition of coupling given by Vessey and Wever [63]:

> two objects are coupled if and only if at least one of them acts upon the other, X is said to act upon Y if the history of Y is affected by X , where history is defined as the chronologically ordered states that a thing traverses in time

So given two objects X and Y they are coupled if any method $M_1 \in \{M_y\}$ uses any other method $M_2 \in \{M_x\}$ or manipulates any instance variable $I \in \{I_x\}$ and vice versa.

Cohesion is generally considered to measure how 'strong', stable or independent a class is. Chidamber and Kemerers intuitive approach captures this attribute as the similarity between methods of the same class. Precisely they define the similarity σ between methods $M_1, M_2 \in \{M_x\}$ as '*the intersection of the sets of instance variables that are used by the methods*' [16]:

$$\sigma(M_1, M_2) = \{I_1\} \cap \{I_2\} \tag{2.3}$$

Complexity of an Object as defined by Chidamber and Kemerer also follows the notion of ontologies. Bunge [13, 14] defines complexity as the '*numerosity of its composition*' and Chidamber and Kemerer use this definition to characterize the complexity of an object as the amount of properties of an object X or precisely the cardinality of $p(x)$. So unlike the measures of McCabe [44] or

Halstead [32] which are based on the control flow graph and properties of the code respectively, Chidamber and Kemerers definition of complexity focuses on the ontology of objects.

Utilizing these definitions Chidamber and Kemerer present a list of object oriented measures.

Chidamber and Kemerer Metrics

This section focuses on some of the measures of the Chidamber and Kemerer Metrics Suite (CK-Metrics) which have received considerable attention in the literature reviewed [1, 62].

For a full and more detailed explanation the reader is advised to refer to the original work [16].

Weighted Methods per Class (WMC) aims to capture the psychological complexity of building and maintaining a specific class. Chidamber and Kemerer argue that the more methods a class has the higher the complexity. This also means that classes which have a lot of methods have a significant impact on the rest of the system since any class which inherits from such a 'method heavy' class also inherits all the methods. Further, classes with a lot of methods tend to be more application specific which usually means they can not be reused. The Weighted Methods per Class measure can be calculated as

$$WMC = \sum_{i=1}^{n} c_i \tag{2.4}$$

where c_i are the individual complexities of methods $M_1 \dots M_i \in \{M_x\}$. An important interpretation given by Chidamber and Kemerer is that '*If all method complexities are considered to be unity, then WMC = n, the number of methods.*'. Chidamber and Kemerer explicitly do not specify how to measure the complexity of the individual methods. In the literature reviewed the complexity of the methods was assumed to be uniformly 1 as suggested by the original paper. Even though one might argue that this specific decision is an oversight, empirical evaluations suggest that solely counting the amount of methods is a reasonable measure [5, 8]. Also as mentioned by Vliet [61]

when choosing the complexity of methods to be uniformly, this measure can already be applied before the actual implementation is finished.

Depth of Inheritance Tree (DIT) can be seen as a measure of the overall complexity of the design. The highest value this measure can take is equal to the depth of the inheritance tree in the design. If this value is high we are dealing with classes which inherit a big amount of properties from their ancestors. This makes the behavior of classes unpredictable and can therefore be seen as a design flaw. It is measured using the inheritance tree and counting the distance of the current node (class) to the root of the tree.

Lack of Cohesion in Methods (LCOM) for a class is calculated by looking at the similarity between the methods of a class. For a class X all similarities $\sigma(M_1, M_2)$ where $M_1, M_2 \in \{M_x\}$ are computed. LCOM is then determined by counting the amount of empty and non-empty similarity sets using the formula:

$$\begin{aligned} LCOM = & \quad \texttt{\# of empty similarity sets} \\ & - \texttt{\# of non-empty similarity sets} \end{aligned} \tag{2.5}$$

This measure acts as an indicator for the robustness and reusability of a class.

2.5. Meaningfulness of Software Measurements

This section will describe the theoretical foundations of measurement theory. The work of Zuse et al. [66] on defining software measurement as a theoretical and mathematical framework is discussed.

Using the definitions of Zuse et al. [66, p. 39-66] a definition of scales and their meaningful use is presented, followed by a discussion of the problems encountered when working with software measurements.

When taking a measurement for something we generally map an empirically observable thing to a numerical representation. We then apply mathematical operations like statistics to the results of our measurements to derive further knowledge. However, there are certain conditions which have to hold to make meaningful statements with those measurements. What does ‘Program 1 is more or equally complex than Program 2’ mean? Is such a statement even meaningful? Can we

apply the arithmetic mean to make statements about bigger parts of the program? What is it exactly that our measure tries to capture? Are the characteristics it measures the ones we are interested in? To answer these questions we must first agree on some formal definitions.

2.5.1. Relational Systems

In measurement theory as defined by [53] and presented by Zuse et al. [66, p. 40] we deal with relational systems.

Definition 2.5.1. *(Relational System)*: A relational system $\mathcal{A}$ is an ordered tuple $(A, R_1, \ldots, R_n, o^1, \ldots, o^m)$ where A is a non-empty set of objects, the $R_i, i = 1, \ldots, n$ are relations on A and the $o_j, j = 1, \ldots, m$ are closed binary operations.

For measurements Zuse et al. [66, p. 40] define two important relational systems: the empirical relational system and the formal relational system.

Definition 2.5.2. *(Empirical Relational System)*:

$$\mathcal{A} = (A, R_1, \ldots, R_n, o^1, \ldots, o^m) \tag{2.6}$$

A is a non empty set of empirical objects that are to be measured.
For software measurements we deal with program texts and flowgraphs.

R_i are k_i-ary empirical relations on A with $i = 1, \ldots, n$.
For example the empirical relation 'equal or more complex'.
The arity k_i defines how many empirical objects are part of
the empirical relation. Morasca [50] adds that
"*each empirical relation R_i has an arity k_i, so
$R_i \subseteq E^{k_i}$, i.e. R_i is a subset of the cartesian product
of the set of entities $E \times E \times \ldots \times E$ k_i times.*"

o^j are binary operations on the empirical objects A
that are to be measured (for example a concatenation of control flowgraphs)
with $j = 1, \ldots, m$

The empirical relational system represents the objects as we perceive them. When solely dealing with empirical relational systems we are most of the time already

able to make some generic statements. A person can subjectively decide whether submodule A is more complex than submodule B. However, if we wanted to measure the complexity of a module C which uses both submodule A and B we lack a meaningful way to combine the complexity of submodule A and B.

To solve this issue we take *measures/measurements* and use appropriate *scales*. However, to define measures and scales, we first have to define formal relational systems. Again we use a definition provided by Zuse et al. [66, p. 40]:

Definition 2.5.3. *(Formal Relational Systems)*:

$$\mathcal{B} = (B, S_1, \ldots, S_n, \bullet^1, \ldots, \bullet^m) \tag{2.7}$$

B	is a non empty set of formal objects, for example numbers or vectors.
S_i	$i = 1, \ldots, n$ are ki-ary relations on B such as 'greater' or 'equal or greater'.
$\bullet^j$	$j = 1, \ldots, m$ are closed binary operations on B such as the addition or multiplication

After mapping the objects of an empirical relational system to objects of a formal relational system we have a set of numbers or vectors. For these formal objects we have intuitive ideas on how to work with them: addition or calculating the arithmetic mean are easily realizable and seem feasible. However, even after this mapping we still have no guarantee that operations such as the arithmetic mean lead to meaningful results.

The meaningfulness of such operations and statements as discussed earlier depends on the scale we are dealing with. Next, the definitions of measures and scales by Zuse et al. [66, p. 40] are given.

Definition 2.5.4. *(Measure/Measurement):* A **measurement** μ is a mapping $\mu : A \to B$ which yields for every empirical object $a \in A$ a formal object (measurement value) $\mu(a) \in B$.

Definition 2.5.5. *(Scale)*: Let $\mathcal{A} = (A, R_1, \ldots, R_n, o^1, \ldots, o^m)$ be an empirical relational system, $\mathcal{B} = (B, S_1, \ldots, S_n, \bullet^1, \ldots, \bullet^m)$ a formal relational system and

μ a measure. The triple $(\mathcal{A}, \mathcal{B}, \mu)$ is a scale if and only if for all i, j and for all $a, b, a_i^1, \ldots, a_i^k \in A$ the following holds:

$$\begin{gathered} R_i(a_i^1, \ldots, a_i^k) \iff S_i(\mu(a_i^1), \ldots, \mu(a_i^k)) \\ \text{and} \\ \mu(a \, o_j \, b) = \mu(a) \bullet^j \mu(b) \end{gathered} \tag{2.8}$$

where $a_i^1, \ldots, a_i^k \in A$ are objects which are part of the relation R_i. It is important to note that '*if $\mathcal{B} = \mathbb{R}$ is the set of real numbers, the Triple $(\mathcal{A}, \mathcal{B}, \mu)$ is called a real scale*' [66].

This equation captures what more modern literature [26, p. 33] refers to as the *representation condition* of a measurement. The first part of the definition of a scale (2.8) by Zuse et al. [66, p. 40] - which is known as the representation condition - requires that a measure μ preserves all empirical relations and has an equivalent formal relation. According to Fenton and Bieman [26, p. 119] a measurement can be validated by showing that it fulfills the representation condition. That is because if a measurement preserves all empirical relations and has equivalent numerical (formal) relations, we can use the numerical representation to learn more about the empirical world [26, p. 33].

We have now defined the mapping of an empirical relational system to a formal relational system as a scale. The next section will provide a classification of scales and their implications on meaningfulness.

2.5.2. Scales

Scales provide a mapping between an empirical relational system and a formal relational system. With the definition of admissible transformations we are able to further classify scales. Presented below is the definition of admissible transformations by Zuse et al. [66, p. 43]:

Definition 2.5.6. *(Admissible Transformations):* Let $(\mathcal{A}, \mathcal{B}, \mu)$ be a real scale. A mapping $g : \mu(A) \to B$ is an admissible transformation if and only if $(\mathcal{A}, \mathcal{B}, g \, o \, \mu)$ is also a scale.

Zuse et al. [66, p. 43] use admissible transformations to define the meaningfulness of statements:

Definition 2.5.7. *(Meaningfulness)*: A statement is meaningful if and only if its truth value is invariant against all admissible transformations.

The following section will discuss the different scale types as defined by Stevens et al. [58]. The meaningfulness as defined above is shown for different statements involving different scales.

2.5.3. Types of Scales and the Meaningfulness of Statements

Stevens et al. [58] define five types of scales, each of them being classified by their admissible transformations. Every type of scale allows different mathematical operations.

Nominal Scale

The nominal scale is the simplest scale. A measurement which results in a nominal scale follows the simple rule to assign each object of a certain group to a certain label, where label might be any identifier for example a number. An example for this would be a software project using different programming languages. One could classify each file based on the language used, assigning each Javascript file to group one, each HTML file to group two and so on. It is important to note that for a nominal scale it is not necessary to use numerical values for the groups; we might as well use words. In case our software project consists of more than just one Javascript and HTML file it is possible to make simple meaningful statements. For example, when dealing with groups which have multiple members it is possible to determine the groups which have the most members.

An important property of this scale is that the descriptions of the groups are freely interchangeable and do not affect our results. This means that instead of calling the group containing Javascript files ‘group one’ we can call it ‘group Javascript’. Any function mapping an existing group name to another unique group name preserves

the validity of our results. More generally, any admissible transformation g which is a one-to-one mapping leads to a new scale which preserves the truth value of the previous scale.

Following the example where we classify files based on their programming language we can use the definition of meaningfulness of measurements by Zuse et al. [66, p. 43] to decide whether or not a specific statement is meaningful for the scale our measurement results in. In our example, after classifying each file we can come to the conclusion that most of the files in our software project are Javascript files which we classify as 'group one'. Our statement is: 'group one contains the most files of all groups measured'. When applying an admissible transformation for the nominal scale, which is any g which is a one-to-one mapping, we end up with another nominal scale where the truth value of our statement 'group one contains the most files of all groups measured' is still the same. By the definition of meaningful statements by Zuse et al. [66, p. 43] this statement is meaningful.

Ordinal Scale

When dealing with an ordinal scale we can relate objects between each other. This means given a set of objects we can arrange them in some meaningful way based on a property shared by all objects. A common example found in agile software development is the rating of tasks. When using SCRUM, a set of tasks is presented before each sprint. The developers are then asked to rank the tasks based on difficulty. This is done by comparing the tasks between each other and ordering them by saying 'X is more complex than Y'. This leads to a certain order in the tasks. We can now assign a number to each task which reflects the difficulty rating. As typical for an ordinal scale, these numerical values can be arbitrary, as long as they preserve the assigned logical order. Given three programs, we can rank them using the numbers one, two and three. However, we might as well use the numbers 6, 70 and 120 as long as the mapping preserves the logical order. In general, any function g which is a monotonically increasing function is an admissible transformation for ordinal scales. That is, it will transform the given scale into a new scale while preserving the order of our objects and in turn preserving the truth value of our previous scale.

Stevens et al. [58] note that ordinal scales are often misused because statistical operations such as the arithmetic mean are applied to them even though they are not meaningful in this context. That is because the length of the intervals between different objects might not be equal. When using our example from before, this means that the difference between the task ranked to be the most difficult and the task directly below it is probably higher than the difference in complexity between a task ranked in the middle part and the one directly below it. Calculating the arithmetic mean is subject to errors if we only consider the relative ranking order of objects. However, Stevens notes that the use of the arithmetic mean and similar statistics on ordinal scales, even though not suitable for them, lead to '*fruitful results*'. This is similar to the results found for various software measures. As discussed later in this section not all measures used for this thesis can be classified as interval or ratio scales, which are needed for meaningfully applying operations such as the arithmetic mean. However, various empirical studies have proven their applicability in practice and the usefulness of their results.

This theoretical discrepancy is one of the big issues for software measures. For most software measures presented in this thesis it is hard to find any proof of to which scale they belong to. Zuse et al. [66, p. 151-169] for example show that under specific conditions McCabes Cyclomatic Complexity [44] can be considered as a ratio scale. However, Zuse et al. [66, p. 142-145] also show that the measures of Halstead can not be used as a ratio scale. Calculating the arithmetic mean is not meaningful when not dealing with a ratio or interval scale. For Halstead, the only meaningful alternative would be to calculate the median, because it can be used when dealing with an ordinal scale [66, p. 142-145]. Any admissible transformation that preserves the order of the objects does not affect the median of a measurement on a scale. Since those are the only admissible transformations for an ordinal scale, calculating the median of such a scale and making statements about or based on it is meaningful.

However, the Maintainability Index by Coleman et al. [18] explicitly uses the average Halstead Effort in its formula. Nonetheless the Maintainability Index has found wide acceptance and is probably the most used software measurement today. It has been used to detect difficult to maintain and error prone modules [52] and the original authors [18] also suggest that it can be used to compare different software systems between each other.

This leads to the conclusion that most software measures found today have to be used with these limitations in mind. When interpreting the results of our software measurements we have to be careful and respect the fact that statistical operations such as the arithmetic mean may lead to wrong results.

Interval Scale

With interval scales we finally reach the point where we work with quantitative differences instead of qualitative ones. When using ordinal scales we order the objects based on qualitative attributes. Interval scales on the other hand introduce the notion of differences, intervals and distances between objects.

Interval scales allow the addition and subtraction of measures. They do not have a zero point, that means the absence of attributes measured with a interval scale is not represented. Their admissible transformations are those of the form $g(x) = a\mu(x) + b$ where $a > 0$ and b is any number, allowing us to change the origin of the measure and the unit we use. Interval scales allow us to make meaningful statements about the difference of our measures. For example it is meaningful to say that the difference between the temperature today and yesterday is twice as much as the difference between the two days before. However, statements such as 'today (20°C) is twice as hot as yesterday (10°C)' are not meaningful. This is the case because for interval scales, such as the scales used for temperature and time, multiplication and division are not meaningful. Using the definition of meaningfulness from earlier we see that if we convert our measurement of temperature from the Celsius scale to the Fahrenheit scale the truth value of our statement changes. Using an admissible transformation of the form $g(x) = a\mu(x) + b$, where $a = \frac{9}{5}$ and $b = 32$ for the conversion from Celsius to Fahrenheit the statement 'today (20°C, 68°F) is twice as hot as yesterday (10°C, 50°F)' changes its truth value. Therefore, following our definition of meaningfulness the statement 'today (20°C) is twice as hot as yesterday (10°C)' is not meaningful. The most important property of interval scales to note at this point is that the calculation of the arithmetic mean for measurements on an interval scale is meaningful.

Ratio Scale

Following the definition by Stevens et al. [58] a ratio scale contains all properties of the previous scales and additionally provides the ability to calculate ratios between values and a definite zero value. As the most permissive scale type it allows all statistical operations which are valid for previous scales and additional operations like division or calculating the coefficient of variation. A transformation of one ratio scale to the other can be done by multiplying the value of a ratio scale by a constant. That is, all admissible transformations are of the form $g(x) = a\mu(x)$ with $a > 0$.

Ratio scales have a definite zero value representing the absence of the attribute measured. All other values are starting from this zero point. The zero point is a theoretical construct; that is, when we consider the measure of length we can think of an object having zero length, even though such an object does not exist. For most measurements ratio scales are the desired type of scale. Examples for ratio scales include measures of length and mass.

2.5.4. Atomic Modifications and Partial Properties

In this section additional definitions by Zuse et al. [66] are presented and used to introduce subjective measures. The procedure of Zuse et al. [66, p. 46-57] to classify a software measure as ratio or ordinal scale requires the definition of so called atomic modifications, empirical subrelations and partial properties.

Atomic modifications are modifications to the control flowgraph like adding or removing a node, adding or repositioning an edge. Empirical subrelations are a weaker definition of an empirical relation. They only represent a subset of what the actual empirical relation represents. When comparing two programs between each other the empirical relation 'equal or more complex than' could have the empirical subrelation 'the control flowgraph is equal or more complex than'. '*It (the empirical subrelation) contains partial information about the empirical relation.*' [66, p. 51]

The sensitivity of a measure to an atomic modification is called a partial property. Precisely Zuse et al. [66, p. 52] define partial properties as

Definition 2.5.8. *(Partial Property of a Measure)* The sensitivity of a measure μ to an atomic modification is called a partial property of the measure. We say: A measure μ has the partial property $\geq e$ to an atomic modification X.

Therefore we are now dealing with specific atomic modifications and are looking at the changes in our measurements caused by these modifications. Partial properties define a clear mapping between an atomic modification and the change to the program captured by our measurement. With these definitions clarified, we have to reconsider that measurements are merely a mapping from something empirically observable to a numerical domain. As stated by Kriz [39], the result of the measurement alone is of little value. What matters is the empirical attribute we tried to capture and decisions we can make based on the changes in our measurements. This means a measure only becomes meaningful if we interpret the result and can use it to get more information about the empirical entity we tried to capture .

2.5.5. Subjective Measures

By defining the partial properties of a measure we can subjectively decide whether a software measure is meaningful to us or not. If we accept the idea that an increase in nodes of the control flow graph leads to an increase in complexity, a measure which captures this partial property is meaningful for measuring complexity for us subjectively. This notion of subjective measures is also described by Fenton and Bieman [26].

Of course the goal for any measurement would be that when performed under the same conditions multiple times, it leads to the same result. When talking about subjective measures, we are dealing with personal opinions when choosing the kind of software measurement we use and subjective views when interpreting the results. This leads to the problem that it is hard to find a common consensus. However, as noted by Fenton and Bieman [26, p. 68-70] this does by no means imply that subjective measures provide no benefits.

Subjective measures can still be interpreted and provide important information about the overall picture. For example if we are satisfied with the idea that an increase in nodes of the control flow graph as captured by McCabes cyclomatic complexity number [44] leads to an increase in complexity, we can use this measurement to gain additional information about our software project. When applied to all files we will receive cyclomatic complexity numbers for each file. These numbers can now be used to identify modules of high complexity. Even though this subjective view of complexity might not be the objectively agreed consensus, we can still use this measure effectively to our advantage. An example for the procedure described is presented by Clark et al. [17] who used McCabes cyclomatic complexity number to identify modules which, relatively speaking, have the highest chance of containing defects. This specific use of software measures is called 'software defect prediction' and one of the most active research topics [52, 35] in the field of software measurement.

The notion of comparing parts of a software project between each other on a file to file basis as done by software defect prediction models can be described as a 'safe' way of using software measurements. As long as we are only comparing parts of a software project between each other we are dealing with an ordinal scale. If we for example calculate the Maintainability Index [18] for every file in our software project, we could then order all files based on the results of our measurement. The files ranked at the bottom can then be investigated in detail to detect eventual defects. However, often we want to compare entire software projects or at least modules consisting of multiple files between each other. A 'go-to approach' for this scenario would be to calculate the arithmetic mean of all files to end up with a single number representing the Maintainability Index of the entire module/project. However, calculating the arithmetic mean for an ordinal scale is not meaningful, as it solely focuses on the ranking of objects in relation to others. The differences between objects are not clear and not guaranteed to be equal [58]. Calculating an arithmetic mean would impose that all the difference between our objects on the ordinal scale are equal. This assumption can lead to wrong results, but as Stevens et al. [58] point out, many times it still leads to '*fruitful results*'. It is important to note the imprecision of these kinds of assumptions. When interpreting results based on the arithmetic mean of a software measure, for which the scale is not clearly defined, one has to keep in mind these flaws and be careful with decisions made.

Restrictions for Statistical Operations

The scale we use for any particular software measure places restrictions on the kind of operations we can perform with the results of our measurement. Depending on the scale and the operations performed we end up with meaningful or meaningless results. If we do not intend to use any mathematical or statistical operations to accumulate or analyze our measurement results, we are not faced with any problems regarding the meaningfulness of derived statements. As an example if we wanted to compare multiple files between each other based on McCabes [44] cyclomatic complexity number we could do so and would receive an ordinal scale with all files placed at a specific ranking position. However, if we then wanted to compare entire software projects consisting of multiple files we would have to calculate the arithmetic mean of all files contained in the project to retrieve a single number representing the whole project.

In this case, the result would only be meaningful if we were dealing with an interval scale. This is the case because when dealing with an ordinal scale the relative distances between each object are unknown and could vastly differ between each other, whereas when dealing with an interval scale the differences are clear and can be compared. Therefore, if we were to use the arithmetic mean we would have to ensure that the measure we use results in at least an interval scale.

Zuse et al. [66, p. 57] use the notion of Extensive Structures as defined by Krantz et al. [38] to classify the type of scale specific software measurements result in [66, chapter 8]. An Extensive Structure places restrictions in the form of axioms on a relational system. A relational system which fulfills the axioms of an Extensive Structure provides the properties needed for a ratio scale [38].

The process of identifying the scale of a measurement is not goal of this thesis, therefore the reader is advised to view the original work by Zuse et al. [66] which proposes a model for theoretically classifying software measures.

2.6. Problems of Software Measurements

This section covers challenges faced when working with software measurements. Additionally to problems faced when dealing with measurements in general, software measurements bring their own set of challenges. Generally when using any kind of measurement we have to ensure that our measurement actually measures the empirically observable characteristic (f.e. complexity) we are interested in. Additionally for software measurements, we have to take care when using mathematical operations since those are restricted by the scale our measurement results in, as discussed in the previous section. Another issue for software measurement is that some measurements which are a combination of multiple software measurements (further referred to as *compound measurements*), such as the Maintainability Index [18], are difficult to work with when applied to a software project. When receiving a 'bad' value from the Maintainability Index practitioners often note that it is unclear on how to improve the Maintainability Index [65]. This can be broken down to the issue of an unclear mapping of the software measurement to observable and undesirable characteristics in the code itself.

2.6.1. Measuring what matters

When working with software measurements it is possible that we measure something different than what we actually intended to measure. Following the definitions of section 2.5.4 we know that software measurements are aimed at specific attributes of the software. McCabes cyclomatic complexity number [44] inspects the amount of conditional statements in the code, while Halsteads [32] measures depend on the amount of operators and variables. Antinyan et al. [3] identify this characteristic and classify the conventional software measurements presented in this thesis as 'measures of size'. In [3] Antinyan et al. describe complexity in psychological terms and combine it with software development. They identified two different kinds of characteristics that affect software complexity and classify them into two categories: accidental characteristics and essential characteristics.

Essential characteristics are growing proportionally with the size of the project. Antinyan et al. [3] argue that these kinds of characteristics can not be reduced as

they are 'essential' to the program itself. An example for such an essential characteristic would be the amount of conditional statements (not nested). If a program needs a certain set of decision to fulfill the requirements, it is impossible to reduce the amount of conditional statements to reduce the complexity of the program.

Accidental characteristics on the other hand can be avoided and refactored [3]. These characteristics include deep nesting and misleading comments.

Antinyan et al. [3] proposed a list of software characteristics such as 'Lack of Structure', 'Deep Nesting' and more [3]. Each of these characteristics is classified as either accidental or essential complexity. They conducted a survey asking more than 100 developers to rate the proposed characteristics based on their impact on complexity. Their results show that the characteristics classified as 'accidental' have a significantly higher impact on perceived complexity than the 'essential' characteristics.

Software measures such as the measures by McCabe [44], Halstead [32] and Chidamber and Kemerer [16] are measuring essential characteristics [3]. Measures aimed at essential characteristics can be seen as measures of size [3]. Antinyan et al. [3] conclude that conventional software measures for complexity measure the wrong set of characteristics. They propose that measures aimed at capturing the accidental characteristics would lead to a more meaningful measurement of software complexity.

This shows that even when dealing with measures aimed at quantitative attributes, we have to subjectively decide if the attribute our measurement captures is an indicator for the empirically observable characteristic (f.e. complexity) we want to investigate.

2.6.2. Knowing what we measure

The Maintainability Index by Coleman et al. [18] combines different software measurements, namely McCabes cyclomatic complexity [44] and Halsteads effort measure [32], into a single number representing the 'maintainability' of a software

product. It can provide some interesting insights when comparing software projects between each other, to a degree where one could decide whether to buy system A or B based on their maintainability. However, when used by practitioners *compound measures* such as the Maintainability Index are often criticized [65].

Through the combination of multiple software measurements into a single number the direct mapping of code characteristics (f.e. the amount of conditional statements captured by McCabes cyclomatic complexity [44]) is lost. This leads to issues when trying to improve the Maintainability Index of a file, since it is unclear which changes will positively affect the score. Wu et al. [65] acknowledge the importance of a direct connection from software measures to measurable code characteristics, which have a negative impact on software maintainability (or other important software attributes).

In [65] Wu et al. present their software architecture measurement system called *Standard Architecture Index* (SAI). They acknowledge the need for software measurements to effectively compare software projects between each other. However, Wu et al. were more focused on the impact of their measurements on the software development life cycle. By listening to the practitioners in their company they came up with a list of software measurements associated with specific so called '*architecture smells*' or '*defects*' [65]. These '*defects*' represent recognizable design flaws such as traditional code-smells [28, p. 63-73]. Thus, when confronted with a 'bad' SAI score, the development teams can look at the measurements taken in detail and identify specific flaws (f.e. duplicated code) in specific files. This approach led to a significant improvement in productivity and other aspects for HUAWEI [65] and shows a good example of the usefulness of software measurements.

3. Related Work

In this section other work aimed at comparing different web frameworks is investigated and the difference between existing approaches and the approach presented in this thesis are highlighted.

The applicability of software measurements for comparing different software projects is still up to debate. Problems such as the subjectivity of some measures and the missing theoretical foundation for meaningfully applying statistical operations such as the arithmetic mean are deterrent to the research in this field. This has led a lot of researchers to use subjective measures, since these can still be used to derive meaningful results for specific use cases [26]. While the goal of this thesis is to investigate the applicability of quantitative software measurements, most of the literature reviewed focused on qualitative attributes of the investigated frameworks.

Heitkötter et al. [34] propose a set of criteria and use it to compare different frameworks for the development of mobile web apps. In particular they compared jQuery mobile, The-M-Project, Sencha Touch and Google Web Toolkit based on qualitative attributes. In their work two reviewers were asked to rate the different frameworks based on predefined criteria on an ordinal scale. Their set of criteria was aimed at two different perspectives: the user perspective and the developer perspective.

The user perspective is aimed at capturing the acceptance of the software developed. In [34] this perspective was captured by criteria such as '*User Interface Elements*', '*Load Time*', '*Native Look and Feel*' and '*Runtime Performance*'. '*User Interface Elements*' were assessed by rating the amount of mobile friendly components a frameworks offers. Possibly quantitative measurements such as '*Load Time*' and '*Runtime Performance*' were also measured by subjectively rating them on an

ordinal scale.

The developer perspective includes properties of the framework which are aimed at or perceived by the developers. Heitkötter et al. [34] define a wide range of criteria aimed at this perspective, some examples include: '*License and Costs*', '*Learning Success*', '*Development Effort*', '*Extensibility*' and '*Maintainability*'. Again, subjective measurements on an ordinal scale were used to assess the different frameworks. Heitkötter et al. [34] measure Maintainability by assessing a given frameworks compliance to good programming practices such as modularization. For example, they concluded that the Maintainability of applications built using jQuery mobile may suffer from jQuery mobile's lacking support for modularization of components.

Sommer and Krusche [57] investigate different cross-platform frameworks for mobile application development. They mention the steadily increasing interest in the mobile market and the difficulties involved. Since developing the same application two or three times for different mobile platforms is unfeasible, the use of existing frameworks, which turn a single code base into native applications for multiple platforms is advisable. In [57] a collection of such frameworks is presented and evaluated based on different criteria: '*Functionality*', '*Usability Features*', '*Developer Support*', '*Reliability and Performance*' and '*Deployment, Supportability and Costs*'. Similar to Heitkötter et al. [34], Sommer and Krusche [57] employ subjective measurements using an ordinal scale to measure the presented criteria. '*Functionality*' for example is rated based on the investigated frameworks offer of API's expected from mobile applications, such as network access and geolocation. For their investigation the same application was built in the selected frameworks. They then derived the measurements for the specified criteria by evaluating their development experience and comparing the resulting applications between each other. Sommer and Krusche [57] identified noticeable differences between the investigated cross-platform frameworks investigated, especially concerning performance.

Another attempt at providing a framework to compare cross-platform frameworks for mobile applications is presented in [24]. Dhillon and Mahmoud [24] acknowledge the lack of comparison frameworks for cross-platform development tools (CPDT) and present their own approach. Their analysis was split in three phases.

Phase one analyzes the general capabilities of the CPDT in question. A list of expected features is produced and for each CPDT the amount of features on the list 'supported', 'not supported' and 'partially supported' is gathered. The goal of this phase is to get an overview about the general capabilities of the frameworks in question. The report of the first phase can therefore be used to determine if a framework fulfills the most basic requirements of the project at hand. When comparing this approach with [57], one can see that both works measure the features offered in a different way.

Phase two is aimed at measuring the performance of CPDT when compared between each other or against native software development kits. Dhillion and Mahmoud [24] use well established performance benchmarks for mobile applications and compare the resulting response time in milliseconds.

While phase one and two measure quantitative attributes, such as amount of features supported and different response times, phase three focuses on qualitative attributes of the frameworks. In phase three the 'development experience' is evaluated. This contains information about available tools, such as IDEs and a discussion about the learning curve.

When considering phase one of Dhillon and Mahmoud's work [24], one can see that they end up with numbers representing the amount of features supported, not supported and partially supported for each mobile platform. Sommer and Krusche [57] on the other hand go a step further and subjectively assign a value on an ordinal scale to allow easier comparison and communication. However, this additional step hides quantitative attributes behind a subjective mapping to an ordinal scale. While we could easily verify the list of supported features presented by Dhillon and Mahmoud [24], we would have to exactly know and follow the procedure used in [57] to reach the same result as presented in their work. Even though both works, [24] in phase one and [57] with their measurement of 'Functionality', measure the same thing - the features provided by the framework in question - the verifiability of the measures vastly differs. This difference shows the importance of measures aimed at quantitative attributes.

A recent study by Majchrzak et al. [42] presents a comparison between modern CPDT, namely React Native, the Ionic framework and Fuse. In [42] the same application was built using all three CPDT. Majchrzak et al. [42] then discuss the advantages and disadvantages met during the development with each framework. Their work focuses on qualitative attributes of the presented frameworks and provides an overview of the current state of the discussed frameworks and their applicability for practitioners.

We can see that works focused on framework comparisons mainly target qualitative attributes and involve subjective measurements. Therefore our work tries to apply measurements aimed at quantitative attributes to investigate their applicability to the domain of framework comparison and provide advice based on repeatable and generally verifiable measurements.

4. Software and Tools

During the experiment a wide variety of different software packages was used. The use of these tools ranges from being at the core of the system (the frameworks used) over assisting the development up to calculating the investigated measurements. This chapter will cover the software which was used during the experiment. The main focus of this chapter will be to describe the two frameworks - Laravel and ReactJS. The rest of this chapter is structured as follows: The following sections will provide a basic introduction into ReactJS and Laravel. The final section of this chapter will present the tools used for measurement. This includes a presentation of the self-made tool written to aid this experiment.

4.1. Frameworks

Software we build today is expected to be working for the next few years. For big projects this requirement poses an easily underestimated challenge as the increasing size or scope of the project at hand can make it harder to change the system to fit new requirements. We refer to this as the maintainability of a software project. With the wish to keep our projects maintainable modern software development makes use of established design patterns built upon a common consensus. Web frameworks are often built with some of these design patterns in mind and provide developers with an easily extensible system which implicitly enforces certain design patterns or software architectures.

The frameworks used for this experiment are ReactJS and Laravel. While Laravel focuses on established design patterns and a conventional Model-View-Controller software architecture, ReactJS provides a modern take on UI development which enforces a component based software architecture. The following sections will cover ReactJS and Laravel in more detail.

4.1.1. ReactJS

ReactJS [1] is a modern front-end framework developed by the company Facebook. It aims at providing a framework which makes it easy to develop maintainable user interfaces, while the main focus remains on usability. The main idea used to increase the maintainability of the resulting application is to use a component based structure. This aims at enforcing two important implications: First of all, the component like structure induces a separation of concerns which arguably may lead to less 'super components' responsible for all the work done by the application. Instead, each component is supposed to take care of the variables (the 'state') it needs to display itself and provide the functionality it is supposed to provide.
Secondly, to retrieve a component based structure one has to think about the structure of the application beforehand. This is an important step which may arguably lead to better software architectures as it requires developers to think about how components will interact with other components in the architecture.
Components in ReactJS come with properties, a state (often, but not always) and life-time functions.

Properties (or props for short) are passed to components when they are rendered. An example would be `<App debug=true/>` where `debug` is a property and `true` the assigned value. Props are usually used to pass down the state or functions of the current component to other components lower in the hierarchy. It is important to note that props are read-only, which is the reason why most of the time functions are passed via props. These functions can be called from the child components to trigger an action in parent components, practically allowing child components to modify the state of parent components. This way, all the logic needed to modify the state of a component is kept inside the component itself, as child components can only modify their parents state by using the functions passed down and defined in the parent components definition. As discussed earlier, this enforces a sense of separation of concerns which can result in a flexible system and reusable components.

State: The state of a component is supposed to encapsulate variables needed by the component to properly function. Simple components may not need a state. These components are called 'functional components' and are often written

[1] version 16.8.6

as pure Javascript functions, while conventional components - which use a state - are defined as Javascript classes, extending the React `Component` base class.
Most components will need to use a state. For example a form would keep track of the users input by writing whatever is typed into an input field into an associated variable kept in the state.
The state of a component has to be specifically defined in its constructor. At the same time the structure of the state should also be defined at this initialization step. This once more enforces the notion of planning how the component will look like and what functions it will fulfill. Once the state is setup a component will usually contain functions reacting to certain events f.e. a `handleChange` function which gets called once the user types something into an input field.
Both, changes to a components state as well as changes to their props will cause the component to be re-rendered - or in other words update. These updates and other events can be intercepted using 'lifecycle methods'.

Lifecycle Methods provide a way to execute specific code at certain points of time based on the current component. The lifecycle methods provided by ReactJS have changed over time and will probably change in the future. Examples for available lifecycle methods in the ReactJS version used for this experiment are `componentDidMount` which is called once the component got rendered the first time, `componentDidUpdate` which is called once the state or the props of the component changed and `componentWillUnmount` which is called once the component is no longer a child component of any rendered component.

Since ReactJS is specifically designed to build user interfaces it can not be considered an one-for-all solution as for example Laravel. This means, that unlike Laravel, ReactJS does only provide features for the front end part of an application but does not provide any functionalities expected from a back end as for example session storage or routing HTTP requests. In other words, when building a conventional web application with users logging in and working with resources, ReactJS will most likely not be enough.

ReactJS does not aim to be an all-in-one solution. It is a Javascript library aimed at creating user interfaces and nothing more. It is meant to be included into projects

and used along with other technologies. In our case we used a fairly common approach which is called a 'MERN' stack. This abbreviation stands for MongoDB, ExpressJS, ReactJS and NodeJS, which is a convenient shorthand capturing all technologies involved for the ReactJS side of this experiment.

NodeJS

NodeJS [2] is a Javascript runtime allowing the development of different types of applications using Javascript as a programming language. It automatically comes with a package manager called `npm` [3] which allows anyone to install publicly available packages in mere seconds. The vast amount of available packages can lead to significantly lower development times as parts of your business logic might be handled by an external library.

MongoDB

One of the more popular choices for non-relational or so called 'NoSQL' databases is MongoDB. MongoDB uses a 'Document-Store' which is one of multiple versions of data stores available for NoSQL implementations [33]. As a document store, data stored in the database is stored in a semi-structured fashion. We can create databases as usual and databases can contain collections, which can be understood as tables in a conventional MySQL database. In these collections we can now store documents. Again, compared to a MySQL database these documents are the entries of a table. The major difference between the two however is that the documents stored in a collection do not have to follow a strict format or more precisely a strict schema, while entries in a MySQL table have to fit the schema defined by the table.

When it comes to web development often a MySQL database is used for simplicity. Laravel for example is generally built to use a MySQL database. However, a fairly modern development especially fueled by cloud computing and cloud services has lead to a shift away from conventional SQL databases to databases which are generally described as NoSQL databases as for example MongoDB.

[2] version 10.15.3
[3] version 6.4.1

MongoDB is often used in cooperation with ReactJS or other Javascript frameworks since its structure for documents uses the javascript object notation (JSON). However, the selection of the database used should be considered a little more thoroughly and should not be based solely on the fact that the entries use the some object notation as the programming language used, since this can easily be achieved by converting the data formats.

With MySQL still being one of the leading database in use today a lot of research is put into this technology. MySQL will work fine for most applications. However, applications with a lot of transactions which can not be bundled together or a huge amount of different table entries pose a performance problem. Since MySQL works on relational databases it is impossible to horizontally scale the database, since this would break certain relations. The only possibility remaining for scaling is to use better hardware (scale vertically).

Scaling vertically is generally something you would like to avoid as it is expensive and has an upper bound; which means using the best hardware available and then having to wait until better hardware is available. This is already one of the big disadvantages of MySQL and one of the reasons why technologies like NoSQL appear to be a viable option.
NoSQL does not rely on relational concepts and therefore you can easily split your dataset which allows you to scale horizontally which is cheaper and virtually unlimited. This promise of being cheap makes it an attractive choice for new software projects.

In our experiment MongoDB [4] in combination with `mongoose` [5] was used. `mongoose` provides developers with an easy way to define schemas for models even though in the background we are using a non-relational database like MongoDB.

[4] version 4.0.2
[5] version 5.6.9

ExpressJS

Since ReactJS is only aimed at building user interfaces (or in other words the front end part of the application) we need an additional framework which aims at providing us with the tools we need to develop the back end part of the application. A well known framework for developing web-based applications with NodeJS is the ExpressJS [6] framework.

It provides developers with simplistic interfaces to handle HTTP routing and encourages the use of the so called 'Interceptor' design pattern. Instead of using the term 'interceptor' directly, ExpressJS uses the term 'middleware'.

Middleware can be used to modify the request or do some specific work before the request gets passed on to the next handler - which might also be another middleware. Built upon this principle there is a vast selection of different middleware available for ExpressJS - each of them aimed at a different aspect of back end development.
To aid the development of our application we used several of these middlewares. For example, to enable authentication for our application a library called `Passport` [7] was used. To use it together with ExpressJS specific middleware has to be registered. Additionally, to use conventional session management on the back end side a middleware specifically for this use case `express-session` [8] was also used.

ExpressJS provides the ability to define routers aimed at defining routes which get recognized by the application and then passed to their respective handler. These handlers are often grouped together on a per resource or model basis and organized in so called controllers. This quickly leads as to a conventional Model-View-Controller architecture, even though 'View' is not a fitting term for React applications which use `Redux`. Usually when talking about a view it is not supposed to contain application logic. However, for ReactJS - especially with a setup using `Redux` - a significant amount of application logic is found in the front end of the application while the back end mostly focuses on persisting the data on

[6] version 4.17.1
[7] version 0.4.0
[8] version 1.16.2

the server. This approach gives developers the ability to develop web applications, which can be used even though there is no connection to the web. The code used for the front end can be cached on the client while any actions which would result in changes on the back end can be stored and sent once the back end is available.

Redux

Even though not part of the MERN stack `Redux` [9] is commonly used when developing ReactJS applications. Usually components have a state and associated functions which modify said state. Since our application will consist of a multitude of components, it is likely that after the application reaches a certain size some components will depend on the state of another component. ReactJS is generally designed in such a way that the state is only passed down to childs, and not the other way around. For many use cases this can lead to problems once multiple components depend on the same set of variables as for example a list of resources. With ReactJS in general, such a list would have to be placed high in the hierarchy and passed down through multiple components until it reaches the component which needs them. This leads to arguably unnecessary code and props being passed to components which do not rely on these props specifically but only get them to pass them down to child components.

To prevent such important state variables from polluting the hierarchy a 'central' state, available to all components, can be used. `Redux` exposes a so called 'Redux store' to the application which is responsible to handle the applications state. This application state is generally available to every component, however since not every component will need everything that is contained in the application state each component has the ability to retrieve a subset of the application state.

To read data from the application state stored in the Redux store a component has to define a function which maps the data from the application state to the components properties. This function is usually called `mapStateToProps` and takes the application state stored in the Redux store as an input and returns a Javascript object where the key defines the name of the property under which the

[9] version 4.0.4

component can access the value. The value is simply the value of the application state the component needs to access as for example `state.user`. To access the value the component needs to access the property with the name defined in the `mapStateToProps` function. If the value in the application state changes the component gets re-rendered as expected.

While reading data from the Redux store is fairly simple, changing the application state follows a more sophisticated approach. Redux uses so called 'Actions' and 'Reducers' to manipulate the application state. Changes to the application state should solely be handled by reducers which modify the state in a way depending on the specific actions which occur.

Reducers are functions which take the current state and the current action as input and usually contain a switch statement or multiple if statements. If the type of the current action is recognized, the reducer executes certain functionality and usually updates the state in a specific way and returns the new state containing the changes induced by the action. If the action type is not recognized, usually the current state is returned unaltered.
It is important to note that an application will most of the time consist of multiple reducers and not only one. In this case, each reducer is responsible for a specific part of the application state. An example would be to use a reducer responsible for changes to the user model stored in the application state, while another reducer might be responsible for a specific type of resource which is also stored in the application state. Each reducer only receives its specific current state as input and also solely returns its resulting state. A so called 'root reducer' is then used to combine the results of all reducers into a single application state.

Actions are commonly defined as Javascript objects containing a type and a payload. The type is used in the reducer to trigger specific behavior or change the state in a specific way. The payload is used to provide any additional information the reducer might need to fulfill the expected behavior or change in the state. To trigger any behavior associated with an action, it has to be 'dispatched'. This can be understood as a way of broadcasting an event. By dispatching an action, every reducer will check if it has to react to the specific type of action which was dispatched.

There is a variety of ways of how to handle Redux actions in any application. For this experiment our Redux actions were merely Javascript objects containing a type and a payload which is used to transfer data from user interactions to the reducer.
The reducer in this case is the only instance which executes certain behavior and changes the application state. An example would be if we have an action which tells our application to update the profile picture of the user. Our reducer would react to an action we have dispatched and turn the image into a base64 URL before updating the application state by writing the new base64 into the `profile_picture` field of the user model.

Another approach for Redux actions which is commonly found is that they are not simple Javascript objects but instead functions. Taking the example from before, this would mean that the action itself is a function and would turn the image into a base64 URL. However, since the only instance which is supposed to change the state is a reducer, this style of actions will usually dispatch other actions (this time not functions but once again Javascript objects) which are then handled by the reducer to update the state. This has the advantage that the reducers contain less functionality and are solely focused on updating the application state, which most of the time can be done using simple operations. On the other hand, this adds the need for additional actions: One to signal that an event happened in the UI, which in turn executes a specific function. However, to update the application state this function now has to dispatch another action of a different type which is recognized by the reducer.

There are additional constraints to be aware of when dealing with actions. One of the most important ones is the fact that actions are not supposed to be asynchronous, but synchronous. The same holds for any functions in reducers. This has a significant impact on the design of our application as it does not allow us to handle any asynchronous actions (as for example any interaction with the back end) in Redux actions or the reducers.
To solve this problem there are libraries which change the way actions are handled by Redux and effectively allow them to contain asynchronous code. The solution used for this experiment however uses a different approach, where an additional type of element is added to the Redux ecosystem - so called ‘Sagas’.

Sagas provide a convenient solution for executing asynchronous code when using

Redux. They are provided by a library called `redux-saga` [10] and seamlessly integrate into an existing Redux application. Similar to the middleware of ExpressJS, Sagas also follow the notion of the Interceptor pattern to add additional functionality to the Redux ecosystem. Sagas need to be registered using so called 'watchers'. A watcher connects a specific action type with a specific Saga, which contains the functionality associated with the action type. Once registered, Sagas will react to actions dispatched in the application just like reducers. However, unlike reducers which sole purpose is to update the application state, Sagas are meant to execute some functionality based on the payload of dispatched actions. Most importantly they are built in an asynchronous fashion, allowing the developer to use asynchronous code in the Redux ecosystem. This allows us to react to any action dispatched in our application and execute asynchronous code like for example saving or updating resources using the back end application and waiting until said asynchronous interactions are finished. Usually Sagas execute some asynchronous code and after that is finished dispatch an action to change the application state based on the results of the request.
This is similar to the approach described above, for which actions contain functionality and use other actions which are recognized by the reducer to update the state. However, the important difference in this case is that, while with Sagas our actions are solely Javascript objects containing a type and a payload, without Sagas our actions would be of a mixed form: some actions are functions possibly containing application logic, while others are merely Javascript objects.

4.1.2. Laravel

Laravel [11] is a PHP framework aimed at providing developers with commonly used features out of the box and additional tools aimed at helping developers to write code which is easy to understand. Furthermore, a command line interface which can be used to generate basic scaffolding or even run a web server for quick development setup is included. Recent iterations also automatically come with additional tools which follow modern web development trends such as using `webpack` [12] to

[10] version 1.0.5
[11] version 5.8.22
[12] https://webpack.js.org/

bundle Javascript code.

One of the biggest advantages of using Laravel is the amount of features included in the framework out of the box. Laravel assumes that most applications will need basic login functionality. Even though they do not enforce this, one can use a single command using Laravels command line tool `artisan` - which is automatically included - to generate all the functionality associated with authentication. This includes

- Register
- Login
- Request password reset token
- Reset password using reset token

By configuring a mailing service in the configuration files of the framework we can end up with a fully functional authentication system without writing a single line of code. Other security related features are also handled by the framework out of the box and do not require the developers to take any additional action.

An example for this would be that every request sent to the back end must include a 'cross site request forgery' (csrf) - token. This token can be generated automatically in the view and added to the form with a single line of code. The library used to send asynchronous requests to the back end (`axios` [13]) is automatically configured to use any csrf - tokens found on the current page for every request which is sent.

Laravel also offers an extensive set of optional packages which can be used to quickly add other commonly used features to the application. An official package called Laravel Passport can be used to quickly add open-authentication functionality to the web application, which seamlessly integrates into the existing login and register procedure. Another example would be Laravel Cashier which can be used to quickly setup billing options for users.

Laravel uses multiple design patterns in different places to encourage good design principles. An example for this would be the use of the Facade pattern for most of

[13] version 0.19

Laravels helper functions. Facades are used to provide a simple interface for more complex underlying functionality. They usually implement multiple static functions which can be used to conveniently access important functionalities anywhere in the application by simply importing the facade which implements the desired functionality. Probably the most commonly encountered example for such a facade in Laravel would be the authentication facade which can be used to retrieve the currently authenticated user anywhere in the code. There is a multitude of facades providing different functionality ranging from providing routing information to storing files which are publicly accessible.

Even though the command line interface comes with a web server which can be used to quickly setup the project and start developing, it is not meant to be used in a production environment. This means to run a Laravel application on a server we have to use a web server which is capable of parsing PHP. Most commonly the Apache HTTP Server or `nginx` are used. For this experiment we used the Apache HTTP Server which was bundled with Ubuntu [14].

The general architecture of a Laravel application follows the Model-View-Controller architecture. While the controllers are built using conventional PHP classes, Laravel comes with its own set of features when it comes to models and views. It is not necessary to use the features Laravel offers for models and views, however since we wanted to focus on using each frameworks best practices and core features in our experiment, we decided to stick with the features Laravel offered and covers in its documentation.

Views in Laravel - Blade Templating Engine

To conveniently display information generated by the controller, Laravel uses its own templating engine called 'Blade'. By using Blade it is possible to write basic HTML files which contain fragments of PHP code. To keep the HTML files clean, Blade comes with its own syntax. Even though it is similar to basic PHP it does help to keep the code clean since its so called 'Blade directives' are much more

[14] version 18.04.2 LTS

concise than their PHP counterparts.
Blade directives are translated into specific PHP code by the templating engine before the HTML code is sent to the user. These directives provide basic functionalities needed for views; mainly focusing on displaying data and structuring the code. Common examples for displaying data include

- if statements

```
<html>

  @if($user->loggedIn)
   <h1>Welcome {{ $user->name }}.</h1>
  @else
   <h1>You have to be logged in to access this page.</h1>
  @endif

</html>
```

- for statements

```
<html>

  <ul>

  @foreach($artworks as $artwork)
    <li>{{ $artwork->title }}</li>
  @endforeach

  </ul>

</html>
```

The double curly braces `{{ $variable }}` are used to access the encapsulated value.
Effectively this is a shorthand for writing `<?php echo $variable; ?>`.
Blade comes with a multitude of different directives and not all of them will be discussed in this section. A comprehensive list can be found in the documentation for the Laravel framework.

Additionally to Blade directives for conditional statements and loops, Blade directives which allow us to split code into multiple files are also commonly used.

Blade allows to define a 'parent' view which can be extended by other views at a later point in time. This is done by defining so called sections using the `section` blade directive in the definition of the parent view. Other views which should inherit the structure of our parent view can then use the `extend` blade directive. By doing so the only thing left to do in the child view is to define the content which should be inserted into the sections of the parent view. Additionally, the `include` directive can be used to directly render another view at a specific point in the current view. This allows developers to use a component based approach when developing user interfaces which also means the components can be built in a reusable fashion.

Models in Laravel - Eloquent ORM

As most web based applications will use a database, Laravel comes with a library for an object-relational-mapping (ORM) to easily handle access to the database, called Eloquent. While it is possible to use different database types and possibly even NoSQL databases like MongoDB, Laravel assumes that a MySQL database is used. Eloquent supports MySQL databases out-of-the-box, so we decided to use a MySQL database for the Laravel side of our experiment.

For Eloquent to work it is not sufficient to only setup model files and define all fields and relations there. It is necessary to write so called 'migration files' which contain all changes which should be applied to the database. This means if we want to add a new model to our application we would have to first write a migration to create the table. Additionally, the migration file also has to include all the fields which should be stored in the created table. Laravel provides a convenient interface to write these migrations, however it is still necessary to precisely define every data type for every field of our model.

After writing and executing the migration file, it is necessary to write a conventional PHP class to define the Eloquent model. Most of the time this file will include

a list of all the fields defined in our migration file (which define the structure of the table stored in the database). If our model comes with any relations, these relations also have to be defined in the model files with their corresponding foreign keys. These model files can now be included at any point in our application and will give us access to the underlying model fields.

When defining a route to access a resource f.e. `/artworks/1` where 1 is the ID of the artwork we would like to view, Eloquent provides us with convenient ways to access the underlying model. Each route is handled by a specific controller function. Controller functions include a definition of the arguments they take. Usually they will take the request as an argument, however it is possible to additionally specify an argument with a specific model type. If this is done, Eloquent will automatically search for the model with the ID given by the route and make it accessible in the body of the controller function. If the model with the given ID is not found, then Laravel will automatically return a response with error code 404.

4.2. Measurement

This section will cover information about the tools used to conduct the various software measurements investigated during this experiment. Both sides of the experiment use their own set of tools, however it was necessary to develop additional tools mainly to assist in combining the results from various sources and to ensure the calculation of the measurements is done in a similar fashion. However, since we did not find any tools to calculate software measures for the Blade files of the Laravel side we wrote a tool to conduct those measurements. This tool will be described at the end of this section. The following sections will briefly cover the tools used to conduct the basic software measurements on each side. While this section aims to give a brief overview over the tools used, chapter 6 focuses on detailed information about how to specifically calculate each measurement and how they were calculated over the course of this experiment for both sides.

4.2.1. Tools used for ReactJS

There are different solutions for calculating software metrics for Javascript projects. However, since ReactJS uses JSX - an unique syntax, basically a combination of Javascript and XML - most tools will not work for ReactJS projects out of the box. The solution we used for our experiment is called `plato` [15] which is able to generate a HTML report for an entire software project. To further process the data generated using `plato` we used `gulp` [16]. With `gulp` we extracted the results of the measurements by `plato` and combined them with the results of the other tools used for measurement.

While `plato` was used to calculate the Halstead metrics [32] and McCabes Cyclomatic Complexity [44] a different tool called `dependency-analyze` [17] was used to calculate Afferent-Coupling, Efferent-Coupling and Instability [43]. `dependency-analyze` can be used to retrieve the list of dependencies a given Javascript file has. We calculated this list for every file in our project and used this information to calculate the values for Afferent-Coupling, Efferent-Coupling and Instability. The exact procedure we used to calculate these measurements can be found in section 6.1.

4.2.2. Tools used for Laravel

For Laravel we used `phpmetrics` [18] to calculate the Halstead Metrics, McCabes Cyclomatic Complexity, Afferent-Coupling, Efferent-Coupling, Instability and the Maintainbility Index [18]. However, `phpmetrics` can only be used to calculate these measurements for PHP classes. Since the project did not solely consist of PHP classes, but also included Javascript and Blade files (special syntax used for views) this single tool did not suffice to conduct the measurements on its own.

To calculate the measurements for the Javascript files in the project we used the same tools and methods as we did on the ReactJS side of the project and combined

[15] version 1.7.0
[16] version 4.0.2
[17] version 1.2.1
[18] version 2.4

the results of those measurement tools with the ones of the tools of the Laravel side. However, to conduct the measurement on the Blade files of the project, we had to develop a parser for Blade syntax and use it to calculate the values for the measurements.

Parsing Blade Syntax

To parse the views of the Laravel side and retrieve an abstract syntax tree which we can then traverse to conduct the software measurements, we used ANTLR [19]. Since ANTLR is a popular tool we were able to find the lexer and parser definitions [20] needed to parse basic PHP syntax. We adopted these lexer and parser definitions to support basic Blade syntax by adding lexer and parser definitions for Blade directives. While the tool is able to parse most of the Blade directives available, there still might be deviations which can not be parsed. However, for our experiment we agreed on using only a subset of the available Blade directives. We would like to add that even though this sounds severe, this did not have a significant impact on the development of the application since there are only a few simple and commonly used Blade directives while there is a multitude of specific complex ones, which can be helpful but are mostly syntactical sugar and not required.

Only a small set of Blade directives was relevant for our measurements. To calculate the cyclomatic complexity number we needed to detect any `if` Blade directives. To calculate the Afferent-Coupling, Efferent-Coupling and Instability of Blade files we used the `include` and `extends` Blade directives and treated each of them as an outgoing dependency or import in other words. For the remaining measurements no additional Blade directives had to be considered.

Our tool only provides functions to detect the three Blade directives mentioned above. Additional constraints apply: The lexer and parser definitions used were not able to parse new `php7` syntax like the new null coalescing operator `??`. Furthermore, we did not include definitions necessary to parse Blade comments. We developed our application with these constraints in mind, meaning that we did

[19] version 4.7.2, https://www.antlr.org/

[20] https://github.com/antlr/grammars-v4/commit/4fd80d744908569957eb54d87a51ae3368faa7b1

not use the ?? operator and also did not use any Blade comments in our views. Additional constraints not mentioned here may apply, however they were not discovered over the course of this experiment.

5. Method and Application

To evaluate the chosen software measurements and their applicability to the chosen web frameworks, we developed the same application twice in both frameworks. This chapter will cover information about the application developed and the development process. We will look into a brief description of the application developed and the expected features. This is followed by a description of the different sprints we used to split the development process. Finally, we will look at the different architectures implied by the two frameworks used and discuss their attributes and expected influence on the software measurements.

5.1. Overview and Expected Features

As it was necessary to have the same application with an equivalent feature set developed in two different frameworks we decided to develop our own example application. We decided to build an application for an actual real-life use case.

5.1.1. Overview

For this experiment a web-application which enables users to organize artworks and artists was developed. The application should allow users to upload artworks and add important information such as the title, the artist and more. This includes expected features such as the ability to edit and delete artworks. Additionally, users should also be allowed to add artists along with information about them to the system. This also includes the ability to edit and delete artists. As an additional feature users should be allowed to create an invoice based on a price for a specific

artwork. Fields to specify tax and discounts should be available and the generated invoice should be download- and viewable as a PDF document.

5.1.2. Expected Features

Following is an exhaustive list of features with a short description:

Authentication: Users should be able to register accounts and login afterwards. Additionally, this includes the ability to request a 'forgot password' token which can be used to reset the password.

Artworks: Users should be able to upload artworks and add additional information such as artwork title, artist and year of creation. They should also be able to change this information at a later point in time and delete any artworks if necessary. Also, a list of the currently added artworks should be shown with an overview containing the most important information about artworks.

Artists: Users should be able to implicitly add artists by entering their name when uploading an artwork but should also be able to add artists manually. Additionally, they should be able to edit existing artists to add additional information like date of birth. There should also be a list showing all artists currently known to the system.

Invoice: Users should be able to generate an invoice for a specific artwork. Users are able to select a price for an artwork and add additional tax and discount information to the form which is added to the invoice. The invoice should be download- and viewable as a PDF document.

Dashboard: After logging in, the user should see an overview of all the artworks currently known to the system. It should focus on displaying the artworks and not contain much information about the artwork itself. Specifically, in the dashboard view only a picture, the title and year of creation of the artwork are shown.

This list of features led to the definition of sprints described in the next section.

5.2. Sprints

To be able to investigate the evolution of software measurements over the development of an application that advances in a similar fashion we decided to split the development into a fixed set of sprints. Each sprint aimed at adding a specific set of features to each application. Measurements were conducted at the end of each sprint for each of the two frameworks. The following list shows the short-hand we chose for each sprint and a short description. As one can see we designed the sprints to contain specific features from the list shown in the previous section:

initial: Both examined frameworks required some specific steps to get the framework up and running. The end of this sprint marks the 'bare-bones' setup of each framework, showing a traditional 'Hello world' page.

framework-setup: This sprint aimed at adding a specific design template to the framework. At the end of this sprint the frameworks contained scaffolded components for login and registering without any functionality.

login: Here, all the functionalities needed for users to register and login were added to each framework. Additionally, the users are able to request a reset password token which can be used to reset their password.

artwork: At the end of this sprint users were able to add new artworks to the system, as well as edit and delete them. This includes asynchronous uploading and extensive form validation.

artists: After the previous sprint artists could already be implicitly added by naming them upon creating an artwork. After this sprint however users were able to explicitly add artists and also edit their information as well as delete them.

invoice: The sole purpose of this sprint was to add the ability to create an invoice for an artwork. The generated invoice should also be download- and viewable as a PDF document.

final-changes: The final sprint was used to clean up and refactor existing code and added a dashboard showing all artworks.

5.3. Architectures and Expected Impact on Measurements

This section will discuss the architectures of the resulting web applications. A short description of the architectural components is given and their interactions are described. Firstly, we will look at the architecture of the ReactJS application before we will cover the architecture of the Laravel application. The last section will discuss the connection between certain attributes of the architecture and the specific frameworks used, and will discuss the expected impact of those attributes on specific software measurements.

5.3.1. ReactJS

ReactJS suggests a component based approach to create maintainable user interfaces. This leads to a separation of concerns as UI components are built in a way that they can independently fulfill their specific function. In our architecture we payed additional attention to the separation of concerns, leading to multiple architectural components, each of them designed to fulfill a specific task. In the following paragraphs UI component refers to a component built and used by ReactJS where as an architectural component refers to a component in the software architecture.

Figure 5.1 shows the architecture for the web application built using ReactJS and demonstrates common interactions between the architectural components. Views consist of a multitude of UI components organized in a hierarchy similar to any HTML or XML document. In our application a 'layout' component is usually the highest in the hierarchy. There are only two layouts available: one for users who are not currently logged in, which is used to display any authentication functionality and the second one which is a dashboard layout and used for any other functionality. Each layout contains a so called 'React Browser Router'. When using the browser router it is possible to define 'React Routes' which can be used to decide which UI component should be rendered at the moment based on the URL the user is viewing. This means we can link an URL f.e. `localhost/login` to an UI component like a login form. As shown in Figure 5.1 this leads to a hierarchical structure in the view as the login form UI component will contain multiple different UI components

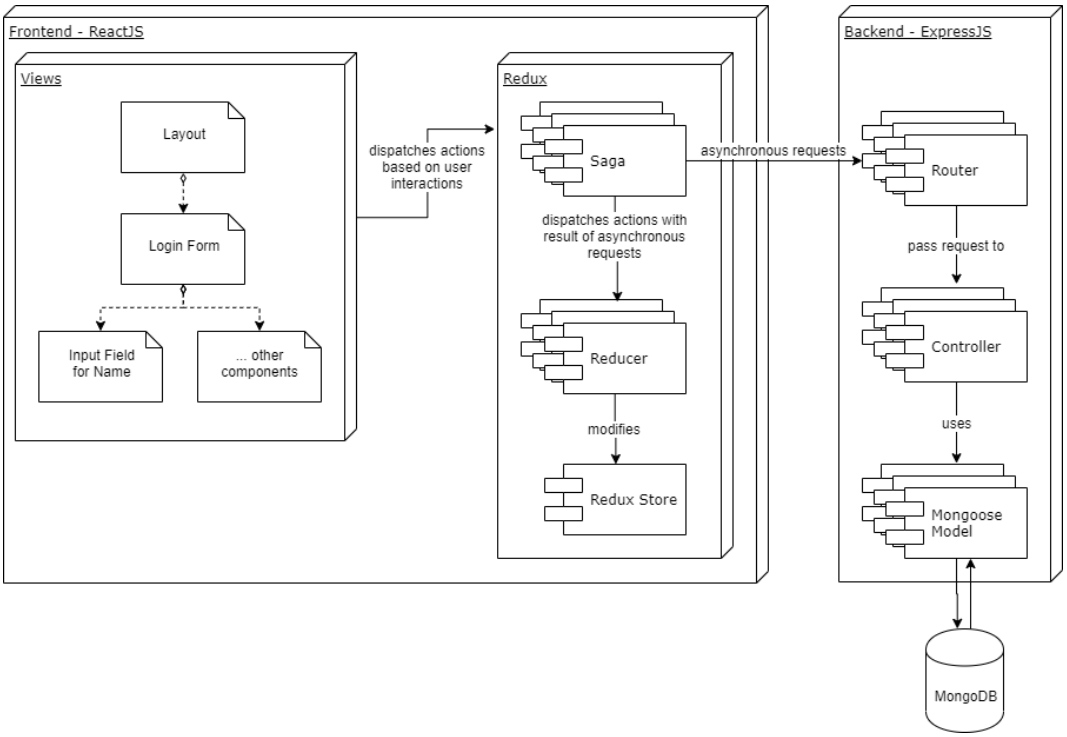


Figure 5.1.: Architecture of the web application built using ReactJS. For views, every module shown represents a ReactJS UI component. The connections between the UI components represent the hierarchical structure of the views. The layout components are the highest in the hierarchy and are responsible to render the correct component based on the URL the user is currently viewing. As an example the 'Login-Form' - component is rendered which itself imports and renders other components. For the rest of the figure every module represents architectural components while the arrows indicate interactions between these architectural components. A stack of modules indicates that there are multiple instances of the specific architectural component. For example there are multiple controllers - one for each resource - and multiple routers. Specific user interactions f.e. submitting a form should most likely cause changes which do not only affect the UI components currently rendered, but instead affect the entire application. (f.e. creating a resource) These changes affect the application state which is stored by Redux. To change this application state the views can dispatch Redux actions. Redux actions are then handled by Sagas and Reducers. Sagas contain processing logic and usually use the information gathered during an user interaction to send asynchronous requests to the back end. Sagas send their requests to specific URLs provided by the routers defined in the back end. Routers are used to expose specific URLs and pass the request to controller functions which handle the specific request. Controllers use `mongoose` models to interact with the database. After processing a request, controllers return a JSON response. This response is once again handled by Sagas. The information contained in the response is checked for errors and in both cases causes specific Redux actions aimed at modifying the application state to be dispatched. These actions are recognized by the Reducers and lead to a change in the application state which automatically leads to an update in components which are affected by the change.

to f.e. show input fields and so on.

After the user filled out a form we usually want to use the information in the form to update or create a resource. In our application the form itself should only contain code relevant for keeping track of the input fields and reacting to user input such as hitting the cancel or submit button. It should not be responsible to send the request to the back end nor should it be responsible to keep track of the progress of the request. Instead, it should format the data received from the user and pass it to a Redux Saga and then be notified if the state of the request changes. To pass the information gathered from the user to the corresponding Redux Saga responsible to handle this specific request, the form component dispatches a Redux action with a specific action type. The corresponding Redux Saga was registered in a way that it reacts to any actions dispatched which have this specific action type.
The Redux Saga is now responsible to transform the formatted information into a HTTP request and send it to the correct endpoint defined by the back end. The available endpoints are defined using so called 'routers' provided by ExpressJS.

It is common practice to have one router per controller and since we use one controller per resource, we end up with one router per resource. Each of these routers defines a set of routes, each of them associated with a specific HTTP method. If a request is received which fits a route defined by any router, it is usually passed to a set of middlewares which are used to verify the validity of the request received and finally passed to a handler which is a function provided by a controller.
Controllers are responsible to process the request and manipulate existing models or add new ones. The models we use are provided by `mongoose` and allow access to the information stored in our `MongoDB` database in a structured fashion. Specifically, this means in our code we are solely editing Javascript objects and after we are done working with them, we call specific functions f.e. `.save()` to update their information stored in the database with the new values.
After the controller is finished processing the request it returns a JSON response containing request specific information. Generally, information about the success of the request and possibly error messages are contained in the responses. The responses are once again handled by the Redux Sagas. Depending on the success of the request actions to update the application store are dispatched. These actions forward the information contained in the response received from the back end and are handled by their corresponding Reducers which in turn update the application

state. Once again we generally use one Reducer for each resource, however there are additional Reducers which keep track of other important information.
For example, we use an 'API - Reducer' to keep track of the state of any requests sent to the API. Components can use the information produced by this Reducer to change based on the current progress and state of any requests.

Our Sagas were also organized on a per-resource-basis originally. However, since our views were held responsible to transform the information entered by the user into the correct format for processing by Sagas, we ended up with similar code for each resource. So instead of having a Saga for each resource we ended up combining the Sagas into a single Saga which is responsible for handling API requests and responses, independent of the type of the resource. This lead to organizing Sagas in a way where each Saga is responsible for a specific category of interactions with the back end; for example one Saga for any CRUD interaction (create, read, update, delete of a resource) and another one which keeps track of any interactions related to authentication.

5.3.2. Laravel

Laravel provides an all-in-one solution for web applications. For ReactJS we had a clear separation between front end and back end. Laravel does not have such a clear separation, as the same framework is used for both all back end tasks and generating the views sent to the user. Additionally, the views and the controllers are closely connected as any functions available to the controller are also available to the views.

In a routing file provided by Laravel one can add the routes which should be recognized by the web application and associate them with the appropriate controller functions. For resources which should follow the commonly used CRUD (Create, Read, Update, Delete) scheme, Laravel provides a helper function which generates all routes for a resource based on established conventions. Laravel additionally provides command line instructions which can be used to quickly generate scaffolding for resource controllers.

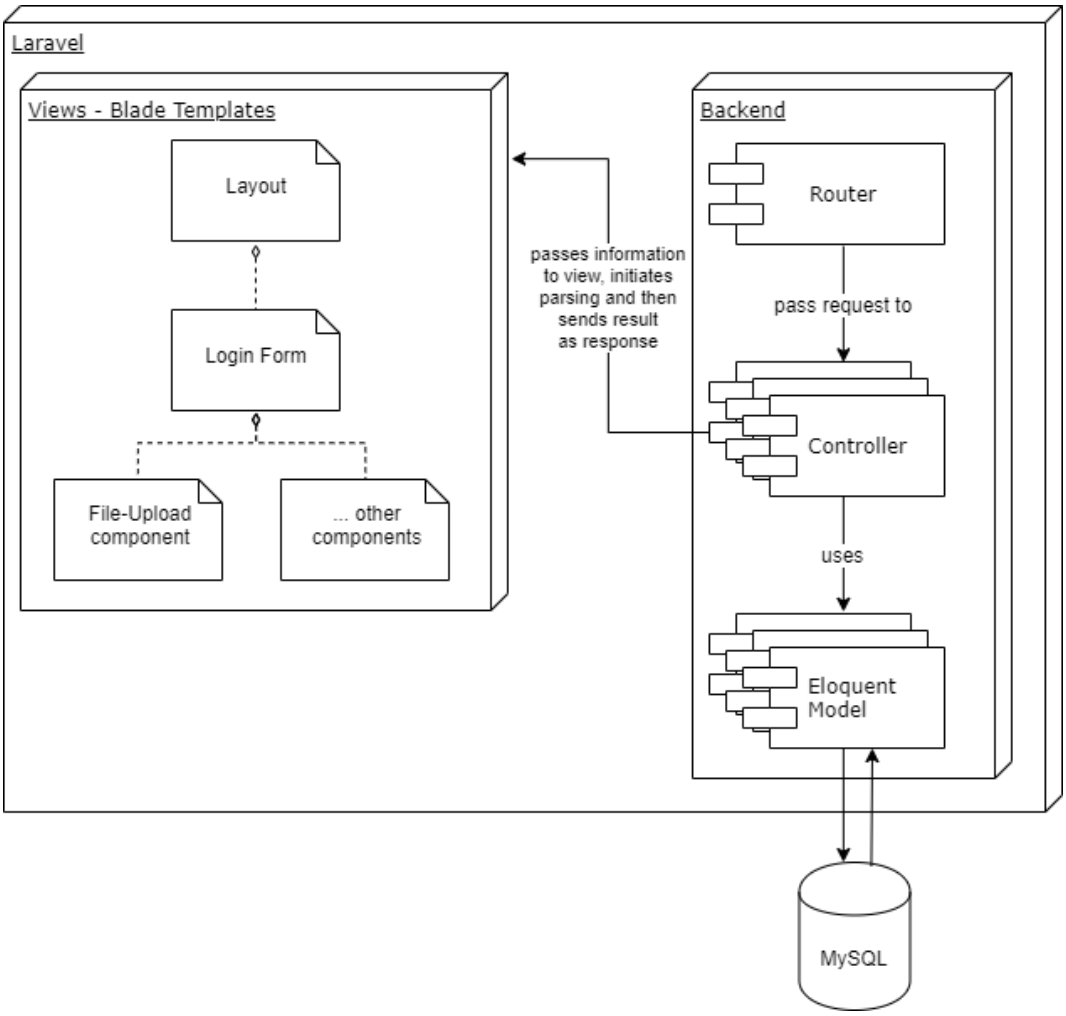


Figure 5.2.: Architecture of the web application built using Laravel. When looking at the views every module shown is an UI component while the connections indicate their hierarchical structure. For the rest of the figure, modules indicate architectural components and arrows indicate interactions between those architectural components. For simplicity, not all interactions are shown. In general, the user accesses certain URLs which are provided by and defined in a router file. In the router file one can define specific URLs accessed via certain HTTP actions and associate this combination with specific controller functions which provide the handling for requests. There are multiple controllers - one per resource and additional ones for specific groups of actions such as f.e. authentication (register and login) or password recovery. To modify data stored in the MySQL database controllers use Eloquent models. Eloquent is the object-relational-mapping library used by Laravel. It provides simple interfaces for accessing and modifying data stored in a MySQL database. The MySQL database is never modified using conventional SQL statements but instead is solely accessed using models, which in turn automatically generate and execute the queries necessary to fulfill the modifications done during the execution of a controller function.

As Figure 5.2 shows, the web application built with Laravel follows a conventional Model-View-Controller architecture. Unlike the ReactJS application, which exclusively communicates with the back end in an asynchronous way beyond the initial loading of the page, the majority of the requests for the Laravel application are handled synchronously. This is the case because we intended to use the entire feature set provided by Laravel, which includes using the Blade template engine for the views. Since the Blade template engine is practically identical to any PHP file, it has to be completely parsed by the server before we can send a response (the resulting HTML file) to the user. This synchronous fashion of handling requests comes with a mix of features and drawbacks.

Since the views are generated on the server and only sent to the user once they are finished, the entire information available to the server can be used to parse the views. This is especially useful since we can use information stored in the session for multiple purposes. Laravel provides simple interfaces to store and display information stored in the session such as error messages. The framework comes with its own tool for form validation. Once setup, it automatically checks any requests sent via forms and checks the fields of the request. If any fields do not match our validation rules, Laravel automatically generates a response and sets error messages specifically for the fields which caused the validation to fail. All of this works out-of-the-box if a synchronous approach is used. However, if we decide to use an asynchronous approach for our requests, we are forced to use Javascript. Logic to parse the information returned by the Laravel framework and to display the error messages in the view becomes necessary. Since we cannot use the features of the template engine such as conditionally displaying content using the `if` Blade directive, we have to define selectors for every HTML element we would like to show / hide. This produces a relatively high amount of lines of code when compared to the synchronous approach.

The views - called Blade templates - were structured in a similar fashion to the ReactJS application. We ended up using two layouts which could be extended by views and filled with the content of the current UI component. Additionally, a component based approach was used to create a set of reusable components which could easily be imported in other views such as header and footer components. This quickly lead to a similar hierarchical structure for views for the Laravel application

when compared to the ReactJS application.

Eloquent models are used by Laravel to provide a simple and high-performance interface for accessing information in the database. Similar to `mongoose` models, they can be edited and are only saved back to the database if the `save()` method is explicitly called.

5.3.3. Attributes and Expected Impact on Measurements

The web application developed using ReactJS shows a high degree of separation of concerns. This should lead to a relatively high number of files where each of them should have a relatively high Maintainability Index, as their complexity is low and the amount of lines of code should also be limited due to the fact that each class only focuses on a specific task.

Additionally, the component based nature of ReactJS automatically leads to a lot of interdependent components, where as each of them can be seen as a class importing other classes. Again, this should lead to relatively high values for the Maintainability Index as the complexity of the software and the amount of lines of code is spread around the entire project and not concentrated in a single file.

Furthermore, the high interdependency between the components should be captured by the coupling measures we use for this experiment. Compared to the Laravel application, the ReactJS application should show higher values for Afferent- and Efferent-Coupling as import statements are used more frequently.

Typically, in a bigger ReactJS application one would achieve high values for Afferent-Coupling as the goal of a component based structure is to built reusable components which are rendered multiple times throughout the application. However, for our web application specifically there were not a lot of components aside from forms which were used multiple times. This should lead to an imbalance between Afferent- and Efferent-Coupling as the amount of outgoing dependencies is generally high as expected from ReactJS components. This imbalance should be reflected by the Instability measurement, which should show values close to one for the ReactJS

application. In other words, the Instability measurement should reflect the high probability for changes required once any of the dependencies imported change. This includes changes in external libraries as well as changes to any other UI components rendered.

Another important difference can be found in the amount of lines of code (LOC). During the `login` iteration basic authentication functionality as described in section 5.2 was implemented. However, the set of features required for authentication was automatically included by the Laravel installation. The scaffolding we used additionally came with a finished set of views. For ReactJS on the other hand it was necessary to develop the entire authentication functionality by hand. This leads to a lot of code being added for the ReactJS side while on the Laravel side only the views were added to the measured code base. For this very reason, our ReactJS application should overall have higher values for LOC or similar measures such as the amount of logical lines of code used during this experiment.

There are other additional factors which should contribute to a significant difference in the amount of LOC since Laravel focuses on providing convenient ways to lessen the amount of code required to do common actions. An example can be observed when comparing the architectures presented in the previous section. As Figure 5.1 and Figure 5.2 show, while ReactJS makes use of multiple routers on a per-resource basis, Laravel only uses a single router. Generally, having a single file containing all routes would be great as long it does not grow too big. Since we had to define every route explicitly for the ReactJS application (the URL of the route and the HTTP method used) every resource led to multiple lines of code necessary to associate the routes with their corresponding controller function. This quickly lead to a lot of code being necessary, so it made sense to split the code across multiple routers. However, Laravel provides a single command which automatically registers all routes expected from a CRUD (Create, Read, Update, Delete) resource controller. This leads to a single line of code to register routes for every resource as opposed to 4 - 6 lines of code for every resource. Additionally, since ReactJS uses multiple routers instead of a single one, additional code for importing dependencies in each router and code to register each router adds to the total amount of LOC.

6. Measures

Software measurements can be used to assess the complexity, maintainability, level of class interdependency and other properties of a given program. For this purpose, one can choose from a wide variety of available software measurements. For this experiment we selected a set of well-known software measurements which are aimed at desired properties of software such as maintainability and flexibility. Additionally, the set of software measures was limited by the amount of software available to actually conduct the measurements. Thus, we ended up with the following set of software measurements:

1. Afferent-Coupling, Efferent-Coupling and Instability [43]
2. Logical Lines of Code (LLOC)
3. Cyclomatic Complexity [44]
4. Halsteads Metrics [32]
    a) Vocabulary
    b) Length
    c) Volume
    d) Difficulty
    e) Effort
    f) Time
    g) Bugs
5. Maintainability Index [18]

The following sections will cover the individual measurements in detail, providing general information, a detailed description of how they are calculated, possible interpretations and finally a thorough description how they were calculated during our experiment.

6.1. Afferent-Coupling, Efferent-Coupling and Instability

This set of metrics proposed by Martin [43] consists of three different measurements, which are considered to be 'object-oriented' measurements. Each of them captures a different aspect of class-interdependency. Martin uses these measurements to categorize classes. Based on the values for the measurements he distinguishes between classes which are hard to modify and therefore unlikely to change and classes which are likely to change. A possible interpretation of these property would be to call it flexibility, as classes which are hard to modify are less flexible.

6.1.1. Afferent-Coupling

This measurement can provide us with interesting insights on the different classes in our software project. Afferent-Coupling measures the amount of incoming dependencies to a class, in other words, the value for the Afferent-Coupling measurement for a given class is equal to the amount of classes which depend on the given class.

Afferent-Coupling can be interpreted as an indicator for high or low flexibility. Martin [43] acknowledges that classes with high values for Afferent-Coupling are classes which have a lot of dependents. This means changing a class with high Afferent-Coupling has a high probability to require additional changes in one of the dependents. Since this can cause severe problems or at least a significant amount of additional work, these classes often tend to be stable; as it is unlikely that they will be changed. In other words, classes with high values for Afferent-Coupling tend to be more rigid (less flexible) than other classes. An interesting interpretation of the overall (arithmetic mean) value for Afferent-Coupling of the project discussed in chapter 7 is that it might give us information on the overall flexibility of the project.

Calculation on React side was achieved using a library called `dependency-analyze`. `dependency-analyze` was used to get a map containing the file name of every file in the project as key supplemented with a collection of this file's dependencies on other files in an array as

```
{
  "resources\\js\\app.js": [
    "./bootstrap",
    "./artworkInvoice"
  ],
  "resources\\js\\artworkInvoice.js": [
    "pdfmake/build/pdfmake",
    "pdfmake/build/vfs_fonts"
  ]
}
```

Figure 6.1.: Example for the output produced by `dependency-analyze`. The hash-map has the file-paths as key and the value is an array containing all outgoing dependencies of the file.

value. Figure 6.1 shows the hash-map for two files containing a list of their individual dependencies.
This map is produced by parsing the input files and storing the filename from every `import` and `require` statement. Given this map we now need to go over every key and for each key look at every imported file and check if it is part of the project files (local files), since we are only interested in measuring the files created by ourselves and not any external libraries. To achieve this, we check the file path of the current dependency which is provided by `dependency-analyze` and check if it is a local file. Generally, most local imports will start with some kind of way to traverse the directory tree of the project f.e. `import "../models/User"` while imports for external libraries (installed using `npm`) never start with any dot (`.`) notation.

We leveraged this distinction and agreed on always using a dot notation at the start of local files when importing them to make this step of measurement easier. This means that any import of a local file starts with either one or two dots f.e. `import "./Alert.js"` or `import "../models/User"`.

To calculate the value for the Afferent-Coupling we first create a hash-map

(called coupling hash-map in the following paragraphs) containing the path to every local file in our project with an integer value of zero for all three measurements (Afferent-, Efferent-Coupling and Instability).
Next we iterate over the map produced by `dependency-analyze` (dependency map) and for every file stored as key we then iterate over every dependency listed in the file's dependency list. For every dependency encountered, if it is a local file, we increase the value for Afferent-Coupling stored in the coupling hash-map by one. After looking at every dependency of every file in the project source files, the Afferent-Coupling values in our coupling hash-map will be equal to the amount of files importing a local file registered in the coupling hash-map.

Calculation on Laravel side: For Laravel we used a tool called `phpmetrics`. Alongside most other measurements described in this thesis, `phpmetrics` provided us with the measurements for Afferent-Coupling for every PHP class in the project source files. However, our Laravel project and most others do not consist solely of PHP classes. Particularly the views play an important role in every standard Laravel application. These views however are not PHP classes. They are not even PHP, but use their own syntax called `Blade syntax`, which is a mix of HTML and special commands which get replaced with PHP code by the templating engine.

Additionally, since the Laravel side also contains Javascript files, the same tool used for the React side was used to calculate the coupling measures for any Javascript files.

Calculation for Blade files: To calculate the coupling measures for Blade files our tool looks for any `include` and `extend` Blade directives in the views. Every `include` and `extend` was treated as an outgoing dependency. Again, we ended up with a hash-map mapping the file-paths of every Blade file in our project to a list of their outgoing dependencies. Using the same approach as described for ReactJS, we now traverse each list and check how many times a given view gets imported by any other view in the project. The number of times a view gets imported by any other views in the project is equal to the value for Afferent-Coupling.

6.1.2. Efferent-Coupling

Just like Afferent-Coupling, Efferent-Coupling looks at another aspect of class-interdependency. Efferent-Coupling measures the amount of outgoing dependencies, in other words the value for Efferent-Coupling is equal to the amount of classes on which the given class depends. According to Martin [43], classes with high values for Efferent-Coupling have a high probability of requiring change in the future, since any change in one of their various dependencies might require the class to adapt. Following this and the previous interpretation of Afferent-Coupling, classes with high values for both Afferent- and Efferent-Coupling could pose a significant challenge for the future development of a software at hand, since they are both likely to change due to a high Efferent-Coupling but also hard to change due to the high Afferent-Coupling. Possible interpretations for overall (arithmetic mean) high or low values of Efferent-Coupling are presented in chapter 7.

Calculation on React side was achieved using the same library used for Afferent-Coupling. However, the calculation of Afferent-Coupling required additional steps, while the calculation of Efferent-Coupling was practically done after running `dependency-analyze`. The tool returns a hash-map with the file-paths of every file in the project as keys supplemented with a list of dependencies for each file as values. The value for Efferent-Coupling for each file is equal to the amount of files it depends on, which means to calculate the Efferent-Coupling from the hash-map produced by `dependency-analyze` we just had to iterate over every key in the hash-map and for every file in this hash-map calculate the length of the list which contains the dependencies for the current file. Unlike Afferent-Coupling which only regards local files (since external libraries are unlikely to depend on project source files anyway) the value for Efferent-Coupling includes dependencies to external libraries. In the example shown in Figure 6.1 both files listed would have an Efferent-Coupling of two, since both import two files.

Calculation on Laravel side was done using the same tools described for Afferent-Coupling.

Calculation for Blade files: As described above our tool returns the same type of hash-map as `dependency-analyze` does for the ReactJS side. To calculate the Efferent-Coupling of our Blade files we also iterate over every file-path in the hash-map and take the length of the list of dependencies of each file as the value for the measurement. For Efferent-Coupling we

additionally counted any usage of assets (Javascript files, CSS files, images...) as an outgoing dependency. This was achieved by looking at every function call encountered while parsing the Blade files. If the `asset` function was used, we treated it as an outgoing dependency to an asset file. The `asset` function takes a file-path which points to a file in Laravels `resources` directory and transforms it into a file-path for the exact same file but after it was moved to a publicly accessible location f.e. the `public` directory of any Laravel application.

6.1.3. Instability

We can combine both Afferent-Coupling and Efferent-Coupling to calculate an additional measurement called Instability. This measurement assigns a value between zero and one to each class. It is calculated using the following formula [43]:

$$\frac{C_e}{C_a + C_e} \tag{6.1}$$

where C_e is the value for Efferent-Coupling and C_a is the value for Afferent-Coupling. Using this measurement we can categorize our project files into three categories depending on the value for Instability:

1. Highly stable (rigid): with a value close to zero these classes have more or even solely incoming dependencies (dependents) and only little or no outgoing dependencies. These classes are usually hard to modify since any changes to these files have a high probability to cause changes in other files.

2. Highly unstable (flexible): a value close to one on the other hand indicates that a class has more outgoing dependencies than incoming dependencies. These classes are expected to have a high probability to change since any changes in one of their dependencies might require changes in the class itself. However, since there is a relatively low amount of incoming dependencies such changes do not have a high probability of causing additional work outside of modifying the class at hand.

3. In-Betweens: Unlike the previous categories this one is not specifically mentioned by Martin [43] himself but instead is an interpretation of his work.

   It is quiet likely that classes do not fit into any of the above categories but simply result in values equal or close to 0.5. This value indicates classes which come with a mix of the properties discussed for the above categories: They have an equal amount of outgoing and incoming dependencies. If the amount of dependencies is low there is no need to worry. However, classes with a high amount of outgoing dependencies and an equally high amount of incoming dependencies are both likely to change and difficult to change at the same time. For such classes the Instability measure could be used as an indicator for bad design since most likely one would be able to split the logic of files in this category up into multiple smaller files, which would lead to different values for the Instability measure and therefore a more stable or unstable class.

Calculation for both sides: Since Instability is just a combination of the Afferent-Coupling and Efferent-Coupling measurements which were described before the only additional step necessary on both sides was to use formula 6.1.

6.2. Logical Lines of Code

Lines of code (LOC) are a probably the most common measure used for measuring the size of software [26, p. 338]. However, most of the time we do not simply take the amount of lines our source files contain since this comes with several drawbacks. When using software measurements most of time our goal is to compare a piece of software in some way to another. If we would simply look at the amount of lines of code to judge whether or not a piece of software is more complicated than another one we could remove any line breaks from our source files and end up with a single line of code. For most cases however removing every line break in a source file will lead to hardly comprehensible source code and by no means less complicated software. This was an extreme example, however it clearly shows that even different coding styles (line breaks before curly brackets of if statements or not) could affect the raw measurement of LOC.

For this very reason deviations of the traditional LOC measurement are used to get more comparable results.
The measurement we used for this experiment was logical lines of code (LLOC). This measurement counts every statement as its own line and removes any empty lines. Comments are also not counted towards the total amount of LLOC. However, depending on the concrete implementation it is still possible that the coding style affects this measurement. Since this was the case for the way we used to calculate the LLOC we agreed on using the same coding style on both sides of the experiment to guarantee that the coding style does not considerably affect the measurement.

Calculation for both sides: Measuring LLOC was provided by the tools used for both sides of the experiment. However, upon closer inspection it became obvious that the way LLOC was calculated was vastly different for both sides.

Since we wanted both sides to calculate the measurement the same way for more convenient inspection we added the ability to calculate the LLOC to our own measurement tool on both sides. Since the calculation of LLOC for all PHP files was done by `phpmetrics` we decided to adapt our tool to use a similar formula. The formula used by `phpmetrics` is presented below:

```
// count all lines
$loc = sizeof(preg_split('/\r\n|\r|\n/', $code)) - 1;

// code for handling comments omitted for clarity

// count and remove empty lines
$code = trim(preg_replace('!(^\s*[\r\n])!sm', '', $code));
$lloc = sizeof(preg_split('/\r\n|\r|\n/', $code));
```

We adopted this simple approach for our own tool, reusing the regular expressions:

```
this.loc = input.split(/\r\n|\r|\n/).length;
this.cloc = 0; // no comments in measured files

input = input.replace(/(^\s*[\r\n])/gm, '').trim();
this.lloc = input.split(/\r\n|\r|\n/)
                 .map((line) => line.trim())
                 .filter((e) => e !== '{' && e !== '}')
```

```
. length ;
```

To simplify the calculation of the LLOC measurement in our experiment we did not use any comments in the files evaluated by our own tool. This means for the React side no comments were used and for the Laravel side the Blade files did not contain any comments.

6.3. Cyclomatic Complexity

McCabes cyclomatic complexity number [44] is one of the most used software measurements to this date. It describes complexity of software as the amount of conditional statements in a program. This idea of complexity may not be unique to software as Gucegliogl u, A. Selcuk et al. [30, 31] show. In their work they use the cyclomatic complexity number to describe the complexity of business processes based on the amount of decisions they contain.

The cyclomatic complexity (CC) of a program is equal to the amount of unique paths in the control flow graph of a program. McCabe describes two possible ways to calculate it. The first approach would be to directly look at the control flow graph and count the amount of unique paths. However, this would require us to generate a control flow graph for each program we want to measure, which is impractical. The second approach acknowledges this problem and therefore provides a much simpler way of calculating the CC. McCabe showed that the CC of a program is equal to the amount of conditional statements plus one. This means we can calculate the CC of a program by simply counting the amount of conditional statements in it.

Cyclomatic Complexity is most of the time used as part of a warning mechanism. Specifically, if the CC of a program reaches a certain threshold it may be necessary to pay closer attention to the program at hand or even refactor parts of it. Following this idea the cyclomatic complexity number found many uses in software fault prediction models [52, 17].

Calculation for both sides: As CC seems to be one of the most used software measurement in practice, the tools we used to calculate most of our measure-

ments already supported the calculation of the CC. No further adaption was necessary.

Calculation for Blade files: To calculate the cyclomatic complexity number for Blade files we counted the amount of `if` Blade directives in the views.

6.4. Halstead Metrics

Halsteads Software Science [32] presents a collection of measurements designed to capture the complexity of software. These software measurements each target different properties of a given program and are the result of an attempt to combine psychological findings with software measurement. The Halstead measures used in this experiment are:

1. Vocabulary
2. Length
3. Volume
4. Difficulty
5. Effort
6. Time
7. Bugs

The main code attributes measured - which are used to derive most of the other measurements - are the operators and operands of a program.

6.4.1. Operators and Operands

The definition of what counts as an operator and what counts as an operand for any given programming language may vary. One of the problems of Halsteads measurements is that a clear consensus for every programming language has yet to be found and defined. This is an important note to keep in mind when trying to compare two projects with different programming languages between each other. Differing definitions for what exactly counts as an operator and what exactly counts as an operand can lead to vastly different results in the measurement.

In general operators are keywords (or symbols) provided by the programming language. This most commonly includes all deviations of parenthesis f.e. `()`, `{}`, any functions provided by the core of the programming language f.e. `printf` for C and other symbols such as mathematical symbols `+`, `-` and special symbols such as colons and dots.
Operands generally are the symbols defined by the programmer, meaning variables and any constants. Each variable name is considered an operand and constants may either be numbers or strings, whereas for strings the whole string is counted as a single operand (instead of each character).

To calculate any of the Halstead measures we first have to count the amount of operators and operands, as well as the amount of unique operators and unique operands; unique meaning if the program contains 15 + signs it is only counted as one towards the amount of unique operators.

Calculation on both sides: was done using the respective tools `plato` for ReactJS and `phpmetrics` for Laravel.

Calculation for Blade files: A problem with the measurements suggested by Halstead is the missing definition of what counts as an operator and what counts as an operand. Without a common consensus it is difficult to compare the values between two tools calculating the same Halstead measurement, since they might use different definitions for operators and operands.

For the Blade files in our experiment the following definitions of operators and operands were used:

Operators: The definition of operators follows the definition given by the documentation of PHP [1]. Additionally, the member access operator '`->`' is also considered an operator as it is the most used operator in the evaluated views.

Operands: For operands we consider every scalar expression (f.e. numbers and strings), every name of any variable and the name of every accessed member variable.

[1] version 7.4, https://www.php.net/manual/de/language.operators.precedence.php viewed on November 17th, 2019

After we determined the amount of operators and operands we can use the formulas provided by Halstead [32] to calculate further measurements. The following formulas are the ones investigated by Shen et al. [55]

6.4.2. Vocabulary and Length

Using the amount of operators, operands, unique operators and unique operands we can calculate Halsteads measures for vocabulary and length, which are used to calculate additional measurements.
The vocabulary η of a program as defined by Halstead is equal to the amount of unique operators plus the amount of unique operands, commonly written as

$$\eta = \eta_1 + \eta_2 \tag{6.2}$$

where η_1 is the amount of unique operators and η_2 is the amount of unique operands.

Similarly, the length N of a program as defined by Halstead is equal to the amount of operators plus the amount of operands, commonly written as

$$N = N_1 + N_2 \tag{6.3}$$

where N_1 is the amount of operators and N_2 is the amount of operands.

6.4.3. Volume and Difficulty

Using the measurements we got up to this point we can now calculate Halsteads Volume and Difficulty.
Volume can be interpreted in different ways. Shen et al. [55] describe it as the amount of bits needed to store the program on the disk. In general terms it is a measurement for program size. The volume V of a program can be calculated using the length and the vocabulary measurement from earlier:

$$V = N \times \log_2 \eta \tag{6.4}$$

The difficulty D of a program as defined by Halstead aims at capturing the difficulty experienced when writing or trying to understand the measured program. The formula

$$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} \tag{6.5}$$

tries to approximate this attribute based on the following assumptions pointed out by Shen et al.[55]. The difficulty of a program can increase in many different ways. The first part of the equation $\frac{\eta_1}{2}$ aims at capturing a change in the amount of operators. When the amount of unique operators changes (f.e. a new operator is added) this term captures this change and the difficulty increases accordingly. The second part of the equation $\frac{N_2}{\eta_2}$ captures changes in the operands. The assumption behind this part of the equation is that if an operator is used more frequently, the difficulty of the program increases.

We can combine these two measures into another one of Halsteads measurements: the effort.

6.4.4. Effort, Time and Bugs

Halsteads measurement for effort is described as the mental effort required to understand or produce a program. This definition is similar to the definition of difficulty in the previous paragraph. The reason for this is that Halesteads measure for effort is calculated using the difficulty measure in relation to the volume - the size of the program. Therefore, Halsteads measure for effort E can be calculated as

$$E = D \times V \tag{6.6}$$

Using the measure for effort we can calculate the last two Halstead measures used in this experiment: Time and Bugs.

Time

Halsteads measurement for time tries to convert the calculated effort into a conceivable unit of time. As Shen et al. [55] point out, Halstead describes the effort measure as '*number of elementary mental discriminations*' [55]. Halstead [32] used

a definition provided by John Stroud [59] to map his measure to a time unit. Stroud [59] claimed that the human mind is only capable of making a limited amount of elementary decisions per second. Based on this assumption Halstead defined his measurement for time T as

$$T = \frac{E}{18} \text{ seconds} \tag{6.7}$$

where 18 is a constant chosen by Halstead which lead to the best results, which represents the limited amount of elementary decisions per second as suggested by Stroud [59].

Bugs

Another attempt from Halstead to map his measurement for effort to a conceivable unit is to project the difficulties encountered while writing a program to a probability of the program containing an error. Halstead assumes that the probability of the program containing an error correlates with the ability of the developer. Therefore his formula to calculate the amount of bugs B in a program contains a constant called 'developer ability'. This constant can be adjusted to accommodate differences in development experience. While more experienced developers are attributed a higher value for the developer ability, less experienced developers should be attributed with a lower value for the developer ability. The formula presented by Halstead is

$$B = \frac{E^{\frac{2}{3}}}{3000} \tag{6.8}$$

where 3000 is the generally suggested value for the 'developer ability'. The tools utilized for this experiment use another version of this formula:

$$B = \frac{V}{3000} \tag{6.9}$$

The resulting value is an estimation for the total amount of bugs which are expected to be part of the measured software.

6.5. Maintainability Index

Similar to McCabes cyclomatic complexity number [44] and Halsteads complexity measures [32] described in the previous section, the so called Maintainability Index

by Coleman et al. [18] is one of the more well known software measurements. As a combination of three different software measurements - Halsteads Volume, McCabes cyclomatic complexity number and lines of code - it assigns a single number representing the maintainability to either single files, entire parts of a software (f.e. individual components of a large system) or even the entire project. Coleman et al. [18] argue that each of the three software measurements used captures different attributes of the underlying software which have a negative impact on the maintainability of the software at hand.

The Maintainability Index (MI) can be calculated for either a group of files or single files. The formula is almost identical for both cases. When calculating the MI for multiple files at once we use the arithmetic mean of the underlying measurements. When calculating the MI for a single file on the other hand, we directly use the value of the underlying measurements.

Below we can see the formula for the Maintainability Index as shown in the original work by Coleman et al. [18]:

$$\begin{aligned} Maintainability = 171 \\ - 5.2 \times ln(aveVol) \\ - 0.23 \times aveCC \\ - 16.2 \times ln(aveLOC) \\ + \left(50 \times \sin\left(\sqrt{2.46 \times perCM}\right)\right) \end{aligned} \tag{6.10}$$

Where $aveVol$ is the average Halstead Volume, $aveCC$ is the average cyclomatic complexity, $aveLOC$ is the average lines of code and $perCM$ is the 'percent of comments'.

The last part of the original formula $+ \left(50 \times \sin\left(\sqrt{2.46 \times perCM}\right)\right)$ aims at mitigating the impact comments have on the MI. Since the original formula uses lines of code and not logical lines of code any comment has the same impact on the measurement as any other line of code. Coleman et al. realized this and adopted their formula to make comments less impactful. As the percentage of comments increases, this part of the formula will return greater values which get added to the

MI calculated up to this point. The encapsulating term limits the resulting value to be between 50 and 0.

However, most tools found during the preparation for our experiment did not use the exact same formula as suggested by Coleman et al. We found that there is a multitude of slight deviations from the original concept being used by many different systems.

Calculation for the React side: As mentioned earlier the MI is one of the most used software measurements. As to expect the tools we used for our experiment already included the calculation for the MI. However, since we used our own tool to calculate the value for logical lines of code (LLOC) we had to recalculate the MI with our own LLOC values. The following code fragment is taken from our measurement tool used for the React side:

```
metrics.mi = 171
- (5.2 * Math.log(metrics.volume))
- (0.23 * metrics.cc)
- (16.2 * Math.log(metrics.lloc));
```

It is important to note, that this formula does not contain the last part of the original formula aimed at mitigating the impact of comments on the MI. The reason for this is that we are using the average logical lines of code (LLOC) instead of the average of plain lines of code. The LLOC measurement does not contain any comments as they are removed together with empty lines before counting the amount of lines in the source code. This is a common deviation of the original formula found in practice.

An additional common adjustment is to scale the MI in such a way that its values range between 0 and 100. The following code fragment demonstrates how we applied the scaling in our implementation:

```
metrics.mi = Math.round(
  (
    Math.min(
      metrics.mi, 171
    ) * 100 / 171 // scaling between 0 and 100
```

```
    ) * 100 // round to two decimals
) / 100; // restore two decimals after round
```

Calculation for the Laravel side: For the normal PHP files such as controllers and services `phpmetrics` was used to calculate the MI. As we can see from the code presented below it uses the original formula given by Coleman et al.:

```
$MIwoC = max(
    (171
    - (5.2 * \log($volume))
    - (0.23 * $ccn)
    - (16.2 * \log($lloc))
    ) * 100 / 171
    , 0);
if (is_infinite($MIwoC)) {
    $MIwoC = 171;
}

// comment weight
if ($loc > 0) {
    $CM = $cloc / $loc;
    $commentWeight = 50 * sin(sqrt(2.4 * $CM));
}

// maintainability index
$mi = $MIwoC + $commentWeight;
```

Since we did not use the comment weight on the React side, we decided to also not use it on the Laravel side. Additionally to the normal MI `phpmetrics` provided us with values for the MI without comment weight. The only thing left to do for standard PHP files was to scale the value received from `phpmetrics` between 0 and 100 as we did for the React side. This is done by our own tool before combining the results of `phpmetrics` with the results for the Blade files calculated by our own tool:

```
mi: Math.round(
  (
    Math.min(
```

```
            currentClass.mIwoC, 171
        ) * 100 / 171
    ) * 100
) / 100,
cc: currentClass.ccn,
ac: currentClass.afferentCoupling,
// other measures are just read from the phpmetrics report
// ... omitted for clarity
```

Calculation for Blade files: For Blade files we used our own tool to calculate the MI in the same way we did it on the React side:

```
this.mi = 171
- (5.2 * Math.log(this.volume))
- (0.23 * this.cc)
- (16.2 * Math.log(this.lloc));
```

The code used to project the result onto a scale between 0 and 100 is identical to the ones shown before.

7. Results and Discussion

The presented software measurements were evaluated by developing the same application twice in different frameworks and inspecting the evolution of the measurements over a set of 'sprints'. In this chapter the results for every software measurement are presented, discussed and interpreted. Specifically, we will investigate the connection between the software measurements used and actual attributes of the measured code. For every measurement we will take a look at the attributes discussed in subsection 5.3.3 and investigate whether the different architectures had the expected impact on the measurements or not. An overview of the resulting values for the arithmetic mean for each of the measurements after the completion of the `final` sprint can be seen in Table 7.5.

7.1. Afferent- and Efferent-Coupling

The following sections present the results for the coupling measures by Martin [43]. For a detailed explanation of the measurements and how to calculate them see section 6.1.

7.1.1. Afferent-Coupling

Figure 7.1 shows the evolution of the Afferent-Coupling measurement for both frameworks. It shows that in general the Afferent-Coupling is higher for the ReactJS side, while it is also increasing steadily and faster with any additions to the project. The Laravel side shows a remarkable increase for the value of the Afferent-Coupling measure between the `initial` and the `framework` sprint, while afterwards the value does not change significantly. This makes sense because during the `framework` sprint a lot of layout components were added to the project. These

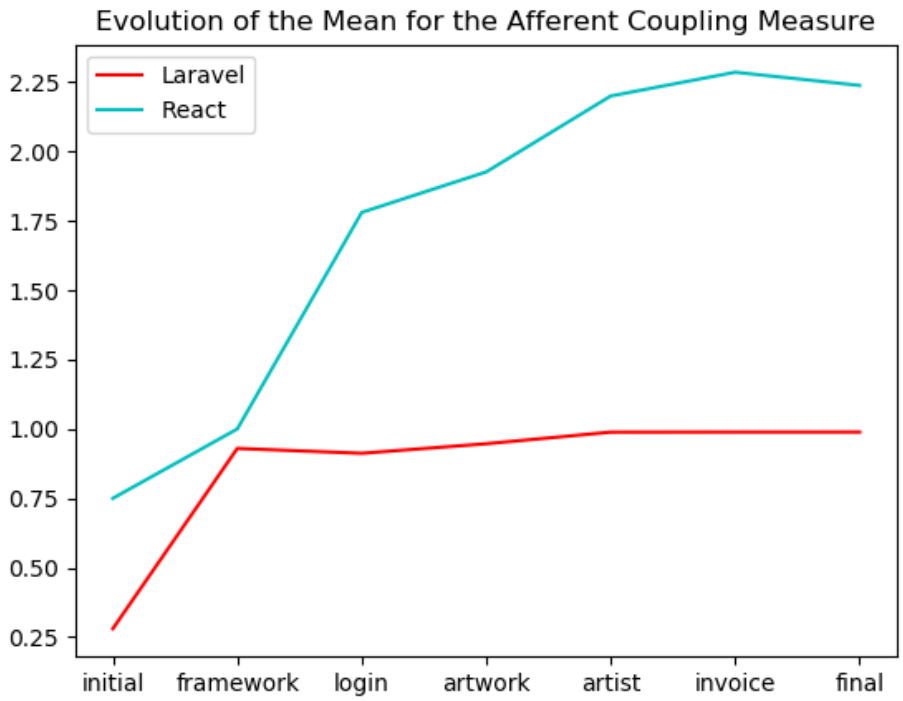


Figure 7.1.: Plot for the evolution of the arithmetic mean for the Afferent-Coupling measure over the course of the development. The X-axis shows the different sprints / points in time where measurements were conducted. The Y-axis shows the resulting value for the arithmetic mean of all the Afferent-Coupling values of every measured file. The significant increase for the Laravel side between the `initial` and `framework-setup` sprint and for the ReactJS side between the `framework-setup` and `login` sprint is due to the addition of authentication logic to the existing code base. During these sprints multiple new files were added to the measurements. Along those files layout files which get imported by a significant amount of other files were added for both frameworks. We can see that for the Laravel side the value of the arithmetic mean remains almost unchanged over the course of the experiment, while the ReactJS side shows a vastly different evolution and steady increase.

layout components are imported by other views and filled with the content of these views. This leads to high values for the Afferent-Coupling measure as suddenly there are a lot of views importing a single layout file.

However, after initially adding a layout for the project it is unlikely that more layouts are added. In our case, supporting additional layouts was not part of the project. The resulting values do not change significantly afterwards because other components added to the project such as controllers usually do not show high values for Afferent-Coupling as they are imported automatically by Laravel and not specifically listed in every file which uses them. In a similar way models are usually only imported by a single controller and therefore most of the time end up with an Afferent-Coupling value of one. After the `framework` sprint the value for the Laravel side stabilizes and does not change significantly. This is likely due to the fact that since the amount of files measured already grew relatively large as shown in Figure 7.5 the impact of outliers on the arithmetic mean is decreased.

A similar significant increase can be observed for the ReactJS side between the `framework` and `login` sprints. Actually, the cause for this significant increase is the same for both frameworks.
During the `login` sprint authentication functionalities such as registering and logging in were added to the project. However, for the Laravel side these features were already automatically included by the framework. This means, that all the code necessary for authentication (including views) was already added to the project with the `framework` sprint of the Laravel side. For the ReactJS side however, we had to implement all the authentication logic ourselves which lead to a lot of code being necessary for both the front end and back end part during the `login` sprint. Furthermore, as Figure 7.5 shows only 10 files were measured at the end of the `framework` sprint for the ReactJS side while at the end of the `login` sprint we ended up with 41 files. As the amount of files measured increases the impact of outliers on the arithmetic mean decreases which is why most measurements grow relatively steady after the `login` sprint. However, the addition of multiple files during the `login` sprint introduced a wider spread of values for the Afferent-Coupling measures as shown in Figure 7.2. This lead to the significant increase observed in Figure 7.1.

The value for the arithmetic mean of the Afferent-Coupling measurement for the Laravel side actually decreased during the `login` sprint because the `framework` sprint added a number of unnecessary views which were removed during the `login` sprint.

ReactJS generally uses an interconnected architecture of components. As discussed in subsection 5.3.3, this architectural attribute should be reflected by higher values for both coupling measures. For our project, the results for Afferent-Coupling reflect this property even though the project itself does not necessarily contain a significant amount of remarkably often reused components. Figure 7.3 shows the distribution for the Afferent-Coupling measure for each framework after completion of the `final` sprint.
It shows that for the Laravel side most of the files have an Afferent-Coupling of zero. These files are mainly views - which are never imported somewhere else unless they are components - and controllers - which are never imported directly as they are managed by the Laravel framework. Additional helper classes which are directly managed by Laravel also fall into this category. It is important to note that Laravels way of automatically connecting framework-related architectural components hides a significant amount of interdependencies between classes from the Afferent- and Efferent-Coupling measurement. This disconnect between observed behavior (for example connections between routers and controllers, since every route is connected to a controller function which handles the request) and measured values (Afferent-Coupling value of 0 for almost all controllers) is an important observation for the Laravel side of our experiment. When interpreting the results of these measurements we have to keep in mind that the measurements do not capture the observed attributes correctly.

For both sides of our project only a little amount of files is imported a significant amount of times. Specifically, there is a small set of files which have more than 5 dependents.

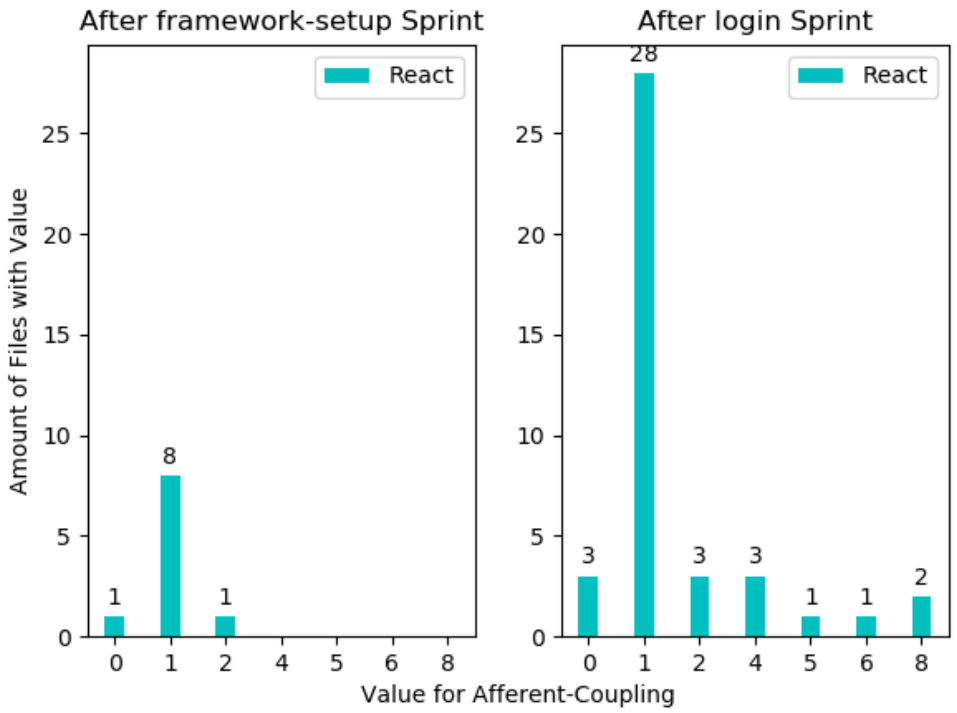


Figure 7.2.: Barcharts visualizing the distribution of values for the Afferent-Coupling measure after the `framework-setup` and the `login` sprint. The X-axis shows the values for the Afferent-Coupling measure. The Y-axis shows the amount of files with the specific value for the Afferent-Coupling measure shown on the X-axis. The numbers above the bars indicate the exact amount of files in this specific group. The number of files increased significantly between these two sprints. The wider spread of the values lead to a sudden increase in the value of the arithmetic mean for the Afferent-Coupling measure.

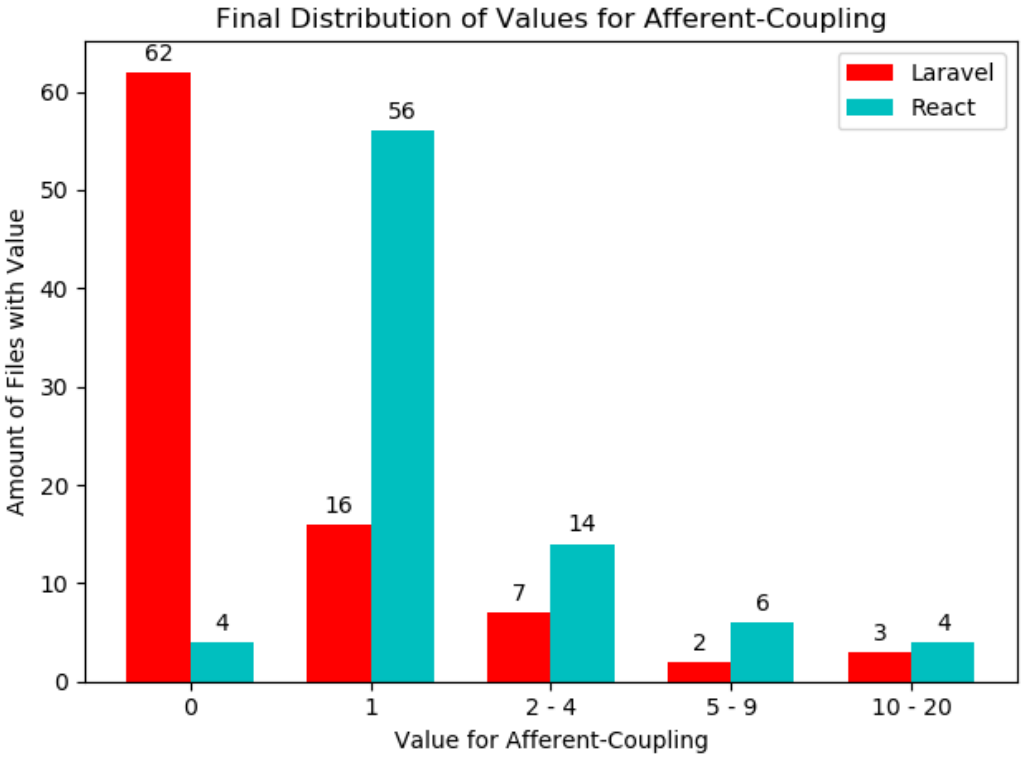


Figure 7.3.: Barchart visualizing the distribution of values for the Afferent-Coupling measure after the `final` sprint. The X-axis shows the values for the Afferent-Coupling measures. Values from 2 to 4, 5 to 9 and 10 to 20 were grouped together for the sole reason of clarity. Each interval is inclusive. The Y-axis shows the amount of files with the specific value for the Afferent-Coupling measure shown on the X-axis. The numbers above the bars indicate the exact amount of files in this specific group. These results show that a significant amount of files on the Laravel side have an Afferent-Coupling measure of zero, which means they are never imported anywhere in the code. This is due to Laravels feature to automatically provide access to framework related classes without needing to explicitly import them.

7.1.2. Interpretation

High values for Afferent-Coupling (relative to Efferent-Coupling) suggest that a class is more difficult to change than one with lower values [43]. This follows the assumption, that classes which get imported by a significant amount of other classes might lead to necessary changes in those other classes. As Figure 7.3 shows that only a little amount of classes show a significantly high value for Afferent-Coupling in our project. After inspecting the files with the highest value for Afferent-Coupling we assessed that in our specific case these files would not be difficult to change as their high value for Afferent-Coupling would suggest.

For the ReactJS side the files with the highest value for Afferent-Coupling were mainly files containing constants and helper functions, while the file with the highest Afferent-Coupling was the so called 'Root Reducer' of our Redux Store.
As the high value for Afferent-Coupling correctly suggests, changes in files which contain constants can lead to a significant amount of changes in different files which depend on these constants. If we would change the name of one of our constants we would have to update every reference to this constant in every file. However, such a change is not a difficult change but solely a change of a variable name.
For files exporting helper functions Afferent-Coupling also correctly captures the high probability for the necessity of changes if changes are made to one of the exported helper functions. However, as most of the time helper functions are added instead of changed these files do not pose a significant challenge in our project.

The 'Root Reducer' combines all other reducers into a single application state. As this is the only place where the final layout of the application state is known and stored, this file usually contains so called 'selectors' which can be used to retrieve specific parts of the application state. As this is one of the most essential functions associated with Redux (accessing the global application state) it is not surprising that this file is imported the most.

For the Laravel side, the files with the highest value for Afferent-Coupling are the layout files which define the overall look of the application. They are used by other views which solely contain the content which should be inserted into the existing

layout. Editing these files usually changes the entire look of the website. However, since in Laravel our views simply define the content which should be put into specific sections defined in the layouts, changes to those layouts most of the time will not affect any of the views using it code wise.

We can conclude that for our experiment the Afferent-Coupling measure did not specifically highlight classes which are hard to change. However, in our project it was helpful as an indicator to find classes which play an important role in the software architecture. Additionally, the measurement reflected the higher interdependency of classes in the ReactJS application.

7.2. Efferent-Coupling

The evolution of the Efferent-Coupling measure vastly differs for both frameworks as shown in Figure 7.4. While the value for the arithmetic mean of the Efferent-Coupling measure does not significantly change for the Laravel side beyond the `framework` sprint, the ReactJS side shows a sudden increase between the `initial` and `framework` sprint followed by a rather steady increase for the following sprints.

The sudden increase between the `initial` and `framework` sprint for the ReactJS side is due to the low amount of files measured during both sprints. As Figure 7.5 shows only four files were part of the measurement for the `initial` sprint and only ten for the `framework` sprint. With such a low amount of files outliers have a significant impact on the value of the arithmetic mean. Since the `framework` sprint introduced a set of UI components required for basic authentication the number of imports increased.

After the `login` sprint the value for Efferent-Coupling stabilizes and increases in a steady fashion.

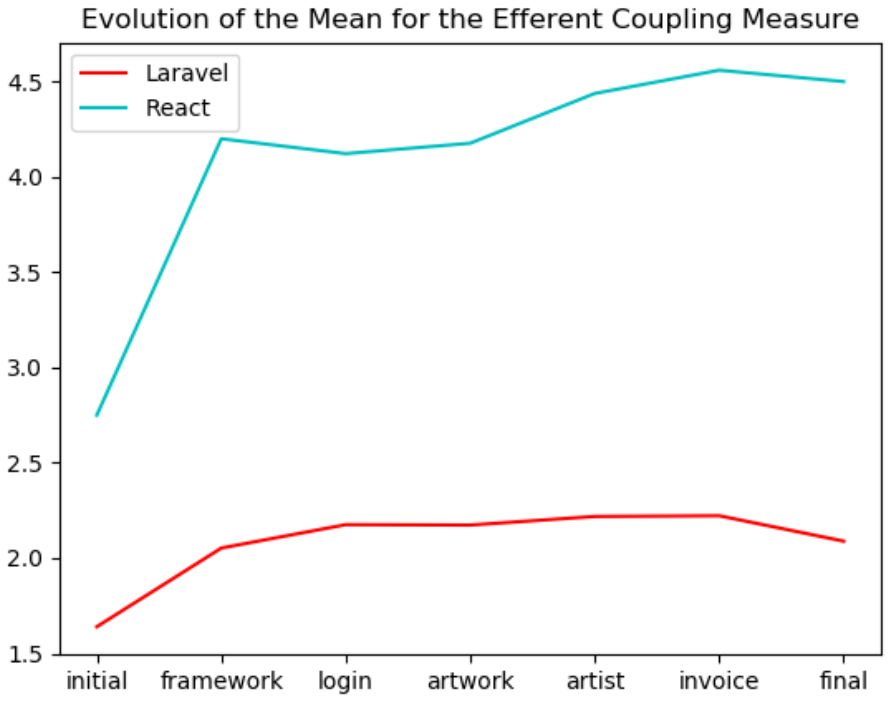


Figure 7.4.: Plot for the evolution of the arithmetic mean for the Efferent-Coupling measure over the course of the development. The X-axis shows the different sprints / points in time where measurements were conducted. The Y-axis shows the resulting value for the arithmetic mean of all the Efferent-Coupling values of every measured file. The significant increase observed between the `initial` and `framework-setup` sprint for the ReactJS side is due to the addition of several views, which themselves depend on a significant amount of components and therefore result in high values for the Efferent-Coupling measure. The ReactJS side shows a significantly higher value in general, however after the value for the arithmetic mean stabilizes after the `framework-setup` sprint, the evolution for both frameworks is similar, while ReactJS shows a steeper increase over time.

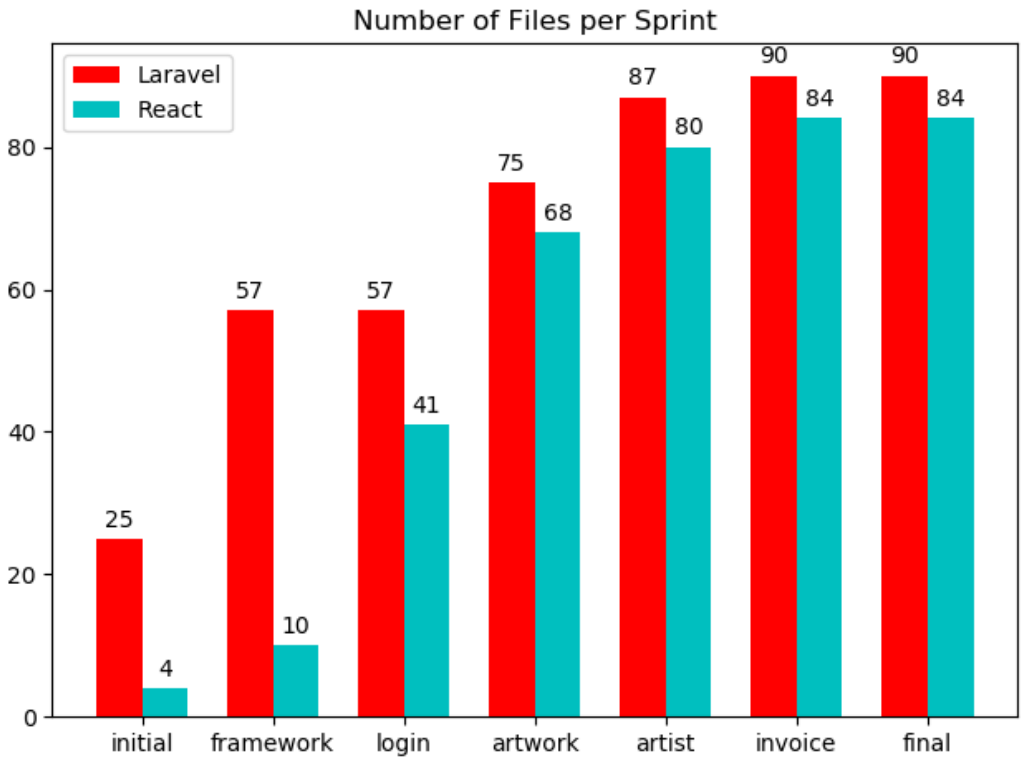


Figure 7.5.: Barchart showing the amount of files included in the measurements for each framework after each sprint. The numbers above the bars indicate the exact amount of files for each framework. Laravel comes with a set of standard files out of the box. Therefore it starts with a relatively remarkable amount of files, however the amount of additional files needed for certain features as for example login functionality is relatively low. The ReactJS side shows a fairly steady increase which is due to the fact that almost any functionality comes with the necessity for a specific set of UI components and back end functionality.

7.2.1. Interpretation

The Efferent-Coupling measure can be used to detect classes which are prone to changes due to their high amount of outgoing dependencies [43]. As Figure 7.4 shows, the Efferent-Coupling for the ReactJS application is generally higher than for the Laravel application. For our application the Efferent-Coupling measure correctly captures the relatively high probability of required changes for the ReactJS side.

Our application uses a multitude of different third-party libraries. ReactJS by itself also depends on a vast amount of third-party libraries. Updates to any of those libraries might require changes in our own application to preserve compatibility and functionality.

However, it is important to note that Laravel automatically connects framework-related architectural components, which means there is no need to specifically import them. While this is helpful when developing an application, it negatively affects the capabilities of our measurements. Without `import` statements (or any programming language specific equivalent) it is not easily possible to measure the interdependencies inside a project. Specifically, as Laravel hides a significant amount of `import` statements, the Efferent-Coupling measure should in general be lower than the actual amount of outgoing dependencies in the project. For the rest of this interpretation the reader is advised to keep in mind that the measurements do not capture the observed attributes correctly.

For both sides of our application the files with the highest Efferent-Coupling were the ones at the core of the system. For the ReactJS side the file responsible for setting up the web server and authentication shows the highest Efferent-Coupling as it acts as central point for configuration. Other files with high Efferent-Coupling serve a similar purpose of combining multiple different components into a single file.
For the Laravel side the main layout used by most of the views has the highest value for Efferent-Coupling as it imports different style sheets and Javascript files.

These results reflect the expectations stated in subsection 5.3.3. As ReactJS uses a component based structure with relatively high granularity, components should be importing a significant amount of other components. Additionally, if the application is built with the reusability of components in mind, certain components should be imported multiple times as they are used more often. Both of these attributes should be captured by the Efferent-Coupling and Afferent-Coupling measures respectively.

For the ReactJS side of our experiment the Efferent-Coupling measure increases steadily as every new component introduced to the application is usually imported somewhere else. It correctly reflects the relatively high probability for required changes once any of the outgoing dependencies of the ReactJS application change.

7.3. Instability

Instability is a combination of Afferent- and Efferent-Coupling. By looking at the relative amount of outgoing dependencies and incoming dependencies we can categorize files as stable and instable. As described in subsection 6.1.3 this property can also be seen as the flexibility of certain classes.

Figure 7.6 shows the evolution of the arithmetic mean of the Instability measure for both frameworks. By looking at the arithmetic mean for the whole project we can get an overview if our project mainly consists of flexible classes, rigid classes (which are hard to change) or a mix of both.

For the Laravel side we can see that the framework starts out (between the `initial` and `framework-setup` sprint) with a value close to one which means the majority of classes measured can be categorized as close to maximally instable / flexible. This is an interesting result considering that when first installing a framework you would of course expect it to be flexible enough to adapt to the requirements of the application you want to develop. For our experiment the Instability measurement for the Laravel side reflects this idea. The value of the arithmetic mean for the Instability measurement drops significantly between the `initial` and `framework-setup` sprint because during the `framework-setup` sprint

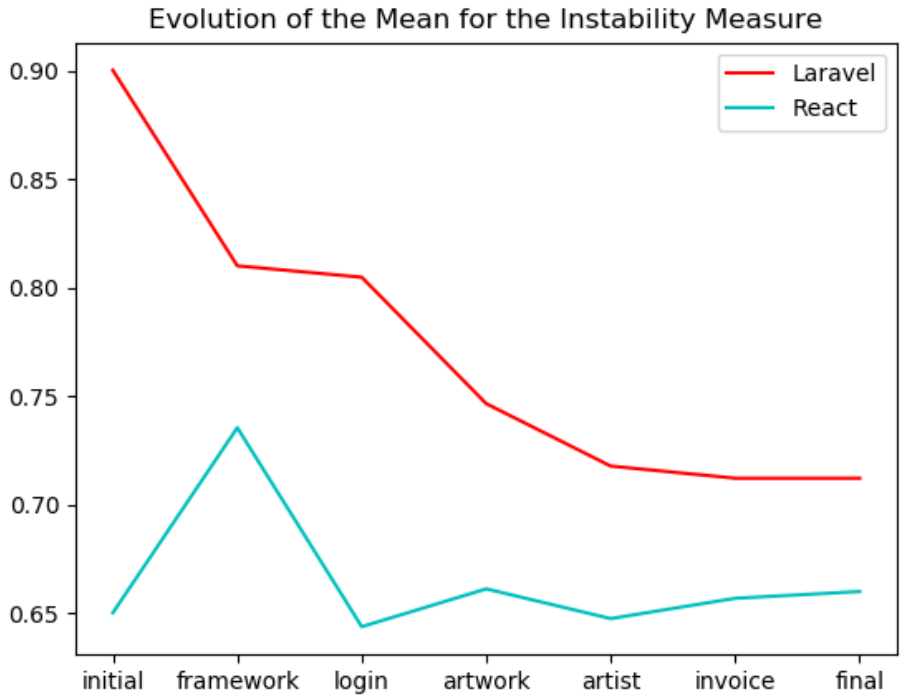


Figure 7.6.: Evolution of the value of the arithmetic mean for the Instability measure for both frameworks over the course of development. The X-axis indicates the different sprints, while the Y-axis shows the value of the arithmetic mean after completion of the sprint shown on the X-axis. Values close to one indicate that the project files are in general maximally instable (likely to change, flexible), due to a high amount of outgoing dependencies relative to incoming dependencies. Values close to zero would indicate that the project files are in general maximally stable (unlikely to change, hard to change). As a project should have a mix of files which are maximally stable and maximally instable, the arithmetic mean of the Instability measure should be close to 0.5 for most software projects. The sudden shifts observed for the ReactJS side of the experiment are caused by the changes of the Afferent-Coupling measure, which themselves are caused by the addition of a significant amount of files between the `initial` and `login` sprint as shown in Figure 7.5.

views for basic login functionality were added as part of the design template used throughout the project. These views were organized as components. Particularly simple components, such as headers and footers, do not import other views. This leads to a Efferent-Coupling value of zero, which in turn leads to a value of zero for the Instability measure, as a class which does not depend on any other class can be considered maximally stable. However, in the case of simple UI components it would be wrong to assume they are hard to change or 'rigid'. Changing the header of a page is as simple as manipulating a single line of HTML code. Of course this will lead to a different look of the overall application if the header is reused on every page, however no adaption in other files will be necessary.

The ReactJS side shows significant changes between the `initial`, `framework-setup` and `login` sprint. Once more the reason for the sudden shift of values during the `initial` and `framework-setup` sprint is most likely due to the relatively low amount of files measured during the `initial` and `framework-setup` sprint as shown in Figure 7.5. We can see that the measure stabilizes after the `login` sprint from where on a significant amount of files is measured which lessens the impact of outliers on the value of the arithmetic mean. Nonetheless we can see that the ReactJS side shows values closer to 0.5 for the Instability measure.

7.3.1. Interpretation

Generally we can see that for the Laravel side the Instability measure steadily decreases over time. Our assumption is that over time the arithmetic mean for the Instability measure should reach a value of 0.5. This is due to the fact that as the application grows a generally equal mix of highly stable and highly unstable classes is added: New UI components are added which do not depend on any other components and are therefore considered maximally stable. On the other side, as the application grows the amount of interdependencies between the classes rises, slowly shifting the value of the Instability measure towards one.

Our results indicate that the Laravel application starts out as a fairly flexible system which slowly gets less flexible over the course of development. For our experiment specifically the `framework-setup` and `artwork` sprint resulted in a significant decrease for the value of the arithmetic mean of the Instability measure.

This is the case because both of these sprints introduced a multitude of views and UI components. These UI components are usually classes which do not explicitly import any other classes and are solely imported by views. Therefore these UI components end up with a value of zero for the Instability measure.
While the value of the measurement changes due to the reasons described in the previous paragraph we would like to draw attention to an important connection between the measurement and the attributes of the application at this point.

As the development of an application progresses it deviates from the initial framework setup. Requirements are implemented and the application turns into a tool for a specific purpose. In our case the application started as an application with basic authentication functionality and slowly turned into an application for managing artworks and artists. For the Laravel side of our application the evolution of the arithmetic mean of the Instability measure reflects this change from an abstract system to a more concrete solution.

The ReactJS side of our experiment shows a similar evolution, however it already starts out with values closer to 0.5. Since ReactJS components are built in an hierarchical structure similar to HTML there is a significant interdependence between every component. This leads to generally high values for Efferent-Coupling as every component depends on a multitude of other components. Additionally, components built in a reusable fashion result in high values for Afferent-Coupling. While the increase in Efferent-Coupling happens rather naturally as the application grows the only way to increase the value for Afferent-Coupling is to pay additional attention to the reusability of components. In general this probably leads to an imbalance between Afferent- and Efferent-Coupling for a ReactJS application and therefore to values slightly above 0.5 for the arithmetic mean of the Instability measure. Therefore, lower values for the arithmetic mean of the Instability measure might be a valid indicator for the amount of reusable components in a ReactJS application.

Instead of looking at the arithmetic mean we can also directly look at the resulting values for the Instability measure to identify highly stable and highly unstable classes. Figure 7.7 shows the distribution of values for the Instability measure after the `final` sprint for both frameworks. It highlights the vast amount of files with an Instability value of one for the Laravel side. These files are mostly views and

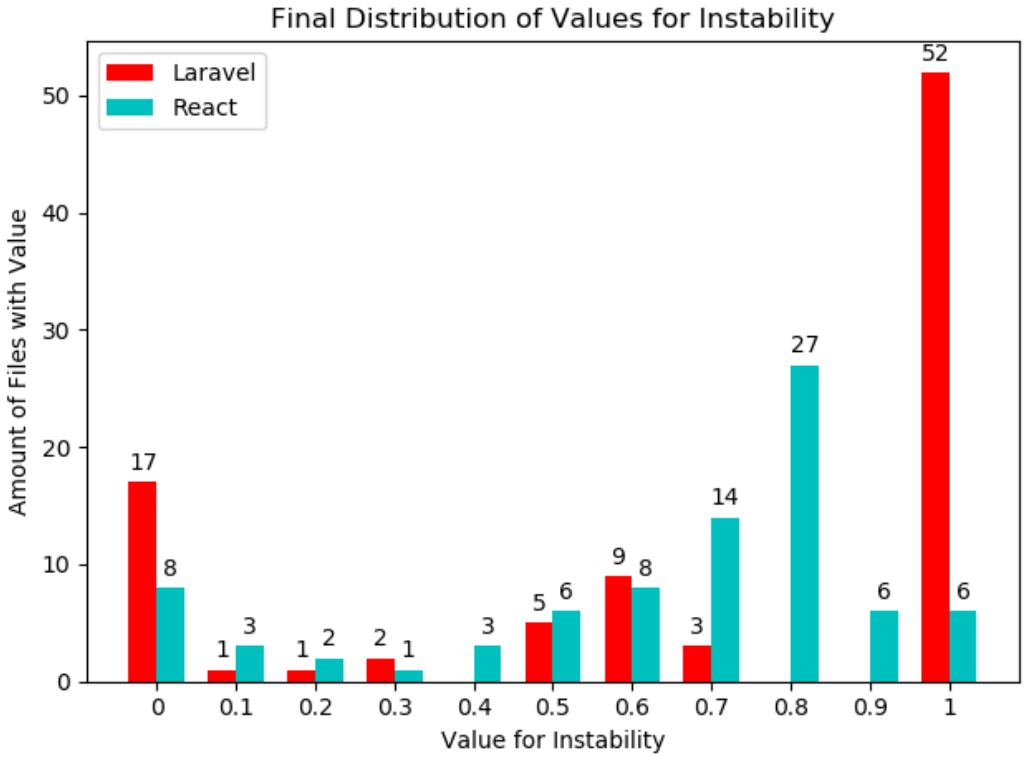


Figure 7.7.: Bar Chart showing the amount of files with specific values for the Instability measure. The values presented are the values measured after the `final` sprint. The vast amount of files with an Instability value of one for the Laravel side is due to different reasons. First and foremost Laravel includes a significant amount of 'helper classes'. These helper classes are used to define certain behavior or act as a point of configuration as for example a class used to configure maintenance mode. They are handled automatically by Laravel and therefore it is not needed to import them anywhere else in the code. Secondly, views combine the layout, their content and other components into the final HTML representation. This means, almost any view is solely importing other classes while they are never imported by any other class. This leads to a value close or equal to one for the Instability measure.

files managed by the Laravel framework such as controllers and helper classes.

Laravel views end up with an Instability value close to one since they usually combine an existing layout and components into a single HTML representation which is sent to the user. Views are managed by Laravel, this means a controller which wants to send a view as a result does not have to import the class which defines the view. Therefore, views for the Laravel side import a multitude of different components and probably a layout, while they are never imported by any other class which leads to an Instability value of one.

The term 'helper classes' tries to capture a set of classes related to the Laravel framework such as Requests, which can be used to setup form validation, and policies, which can be used to prevent unauthorized access to certain resources. They will not be covered in detail, however the important thing to note is that since they are managed by Laravel directly it is unnecessary to import them anywhere. This leads to an Instability value of one for a substantial amount of files for the Laravel framework.

The Instability measure shows that Laravels nature of managing a multitude of things internally effectively hides a significant amount of interdependencies in the system. While a policy file might not be directly imported by any other class it affects multiple controllers at once. This means, changing such a policy file can potentially change the behavior of controllers. However, since the policy files are never imported by any controller, this dependency is hidden from the measurements. Additionally, the files categorized as highly unstable usually contain one to three imports. With such a low amount of incoming dependencies the probability for change is relatively low. This means, for the Laravel side of our experiment the Instability measure can not be reliably used to identify files which are prone to changes.

For ReactJS Figure 7.7 shows that the values for the Instability measure are fairly spread out, while the majority of files show values above 0.5, indicating more unstable classes overall. The ReactJS side shows significantly less files with an Instability value of one since unlike for the Laravel side no dependencies are hidden by the framework. This means most files in the project source are imported at least

once by another file.

The files with the values closest to one for the ReactJS side are views which import and combine a multitude of components. Here, the Instability measure reliably captures the high probability for change due to changes in imported classes. A ReactJS component usually defines properties which are used to trigger certain behavior. When certain behavior is added to a ReactJS component the properties of this component usually change. A change in properties will almost certainly require changes in the parent component.

Values close to zero for the Instability measure also correctly indicate highly stable classes for the ReactJS side. Here, files containing constants and helper functions which solely export variables and functions are indicated by an Instability value of zero. Files with values close to zero share similar properties, namely that they are mostly defining constants or utility functions.

Overall for our experiment the Instability measure seems to fulfill the expectations for the ReactJS side, but can not be reliably used for the Laravel side as a significant amount of dependencies are hidden from the measurement. However, the arithmetic mean of the Instability measure might be capable of indicating an imbalance between highly stable and highly unstable classes. This knowledge could be helpful when trying to classify architectures as flexible or rigid.

7.4. Halstead Metrics

Halsteads 'Software Science' [32] presents a set of measurements for different properties of the measured program. These measurements have been investigated and criticized over the years. Shen et al. [55] point out a significant amount of flaws for both the practical and theoretical background of the Halstead measures. Nonetheless, these measurements were included during our experiment to give additional information about their applicability and usefulness.

Halsteads measures all show a similar evolution over the course of the sprints. For this reason we did not include figures for every Halstead measure in the main document. Additional figures can be found in the appendix.

Figure 7.8 shows the steady increase of the arithmetic mean of Halsteads Vocabulary measure over the course of the development. While this value generally increases we can see a drop of the value after the `final` sprint which makes sense considering that it was used to refactor and therefore most likely shorten the code.

Similarly, Halsteads Length measure steadily increases over the course of the development as shown in Figure 7.9. The evolution of the arithmetic mean of Halsteads Length measure is almost identical to the evolution of the rest (Volume, Difficulty, Effort, Time, Bugs) of Halsteads measures.

7.4.1. Interpretation

For our experiment, all of Halsteads measures show a similar evolution over the course of the sprints. This overlaps with our assumption, that Halsteads measures generally grow with the size of the code for the following reason: Since the 'advanced measures' (Effort, Volume, Time, Bugs) are based on the more general measures (Vocabulary, Length), an increase in the more general measures will lead to an increase for the advanced measures. Furthermore, since the general measures grow with the size of the code, or generally speaking grow with the amount of lines of code one can expect Halsteads measures to grow in a similar fashion as the software project continues to grow. The following sections will go into detail for specific measurements.

Vocabulary and Length

These measures serve as an indicator for the size of a program. This is done by counting the amount of operators and operands. One of the problems with Halsteads measure is that there can be various differences when it comes to different programming languages [55]. Some programming languages may require more operators or more operands than others. Additionally, there is no clear definition

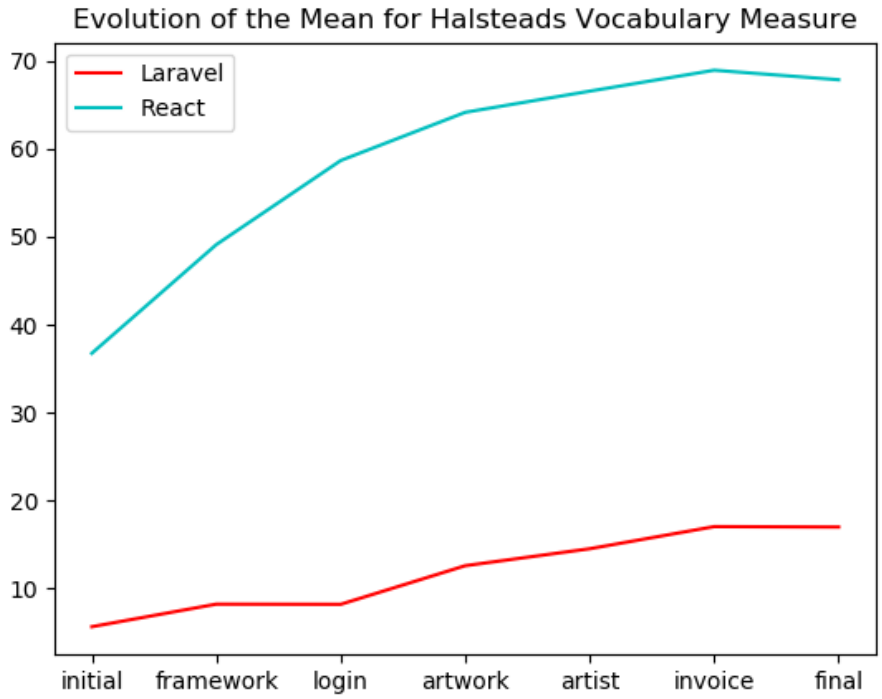


Figure 7.8.: Evolution of the arithmetic mean of Halsteads Vocabulary measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the arithmetic mean after the completion of the sprint shown on the X-axis. The steady increase of this measure is likely the cause for the similarly steady increase of the other Halstead measures. For both sides of the experiment we can see a drop for the value of the arithmetic mean after the `final` sprint, which was used to refactor the existing code.

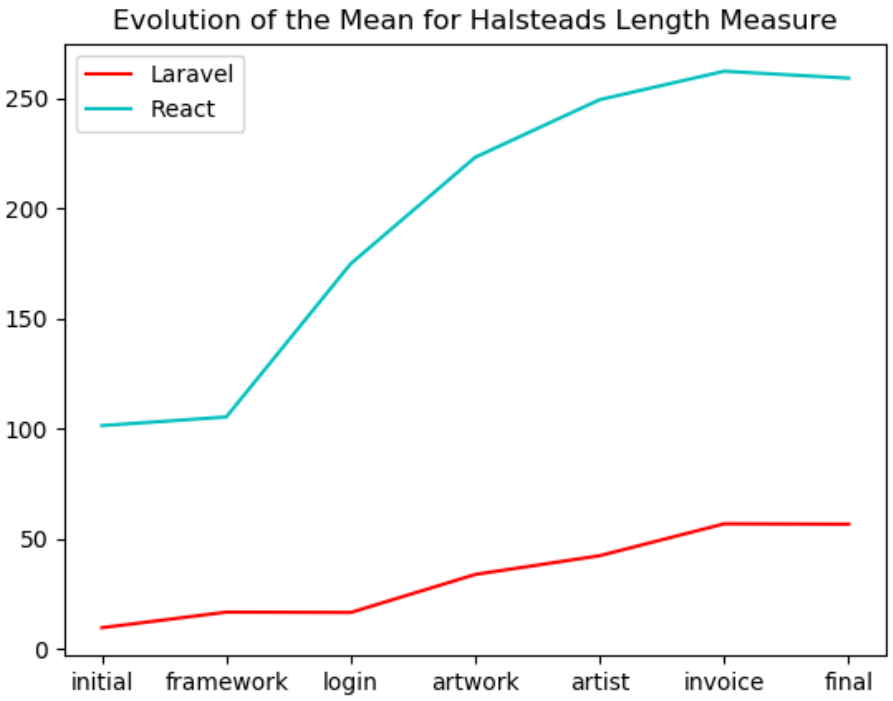


Figure 7.9.: Evolution of the arithmetic mean of Halsteads Length measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the arithmetic mean after the completion of the sprint shown on the X-axis. The evolution of this measure is almost identical to the evolution of the more advanced Halstead measures (Volume, Effort, Time, Bugs).

of what counts as an operator and what counts as an operand for any given programming language. Without a clear consensus it is not meaningful to compare the results of Halsteads measurements taken for two different programming languages.

These discrepancies are likely the cause for the significantly different evolution of Halsteads measures as shown in Figure 7.9. While they should not be used for comparing the frameworks, the measures still capture important attributes when it comes to the evolution of the project. As we can see the Halstead measures correctly capture the constant increase of size of the software projects over the course of the development. Furthermore, the decrease of size during the `final` sprint was also captured for both frameworks.

Volume

Halsteads measure for Volume should allow us to measure the 'size' of files. According to Shen et al. [55] it can be understood as bits needed to save a file on a disc.

When looking at the actual file size of our measured files and comparing it with the values calculated using Halsteads Volume measure we can see that for our experiment 'bits' are far off from the actual file size, while if we would interpret it as 'bytes' it would be 'somewhat close' at best. Additionally, if Halsteads Volume measure would reliably capture the size of a file it should result in the same order as if we ordered files by their actual file size. For the Laravel side of our experiment, after ordering the files measured by their file size in bytes (Table 7.1), the resulting order is different to the one we get when we order the same files by the resulting value for Halsteads Volume (Table 7.2).

However, for the ReactJS side of our experiment the ordering of elements by size on disk and Halsteads Volume is roughly the same. From the 15 largest files when judging by actual file size on disk on one side and judging by the resulting value for Halsteads Volume on the other side, 9 out of those 15 file are contained in both sets. The ordering of these 9 files is still slightly different as shown in Table 7.3 and Table 7.4.

Filepath	Bytes on Disk
views/artworks/form.blade.php	12771
views/profile/edit.blade.php	10629
views/artworks/invoiceForm.blade.php	9573
js/artworkInvoice.js	8932
views/artists/index.blade.php	8289
views/artists/form.blade.php	7800
views/artworks/index.blade.php	7692
views/auth/register.blade.php	6467
Controllers/ArtworksController.php	6092
views/layouts/headers/cards.blade.php	5088
views/users/index.blade.php	4999
views/layouts/navbars/sidebar.blade.php	4835
js/artworkForm.js	4786
views/users/edit.blade.php	4747
views/auth/login.blade.php	4744

Table 7.1.: Actual file size on disk for the Laravel side of the experiment. Only the 15 largest files are shown. This table shows that we obtain a different ordering when compared to the files ordered by Halsteads Volume measure and that the actual file size is vastly different from the amount of bits calculated using Halsteads Volume measure.

For our experiment Halsteads measure for Volume did not lead to meaningful results. Nonetheless it can still serve as a tool to detect files which contain a vast amount of logic. However, plainly using the actual file size (in bytes) might lead to the same (if not better) results with less effort.

Difficulty and Effort

The measured Difficulty as defined by Halstead changes based on the relation between the amount of operators and unique operators, and operands and unique operands. It should capture how difficult a piece of software is to understand / write. Figure 7.10 shows the evolution of the arithmetic mean of Halsteads Difficulty measure which, similarly to most other Halstead measures, grows steadily. We did not investigate the Difficulty measure particularly in detail as it is mainly used in combination with Halsteads Volume measure to calculate Halsteads Effort measure.

Filepath	Halstead Volume
js/artworkInvoice.js	10147.939
js/artworkForm.js	4826.562
js/artistForm.js	3782.737
Controllers/ArtworksController	1387.61
js/fileUploadPreview.js	992.674
views/artworks/form.blade.php	878.766
views/profile/edit.blade.php	617.343
Controllers/ArtistsController	529.4
views/artists/index.blade.php	498.246
views/artists/form.blade.php	470.648
js/artworkIndex.js	406.997
views/artworks/invoiceForm.blade.php	395
js/artistIndex.js	375
views/artworks/index.blade.php	357.576
views/users/edit.blade.php	353.042

Table 7.2.: Values for Halsteads Volume measure for the Laravel side of the experiment. Only the 15 files with the largest value for Halstead Volume are shown. This table shows that we obtain a different ordering when compared to the files ordered by size on disk (in byte) and that the actual file size is vastly different from the amount of bits calculated using Halsteads Volume measure.

Halsteads Effort measurement tries to assign a numerical value to the mental effort required to develop or understand a piece of software. It is calculated as a relation between Halsteads Difficulty and Volume, therefore the more difficult a program is to understand / write and the bigger it is, the higher the mental effort required to understand / write it.

Judging by subjective observations, Halsteads measure for Effort worked surprisingly well for both sides of our experiment. The files with the highest value for Effort are the ones which subjectively took a significant amount of mental effort to develop and seem like the ones which would take the most amount of mental effort to understand.

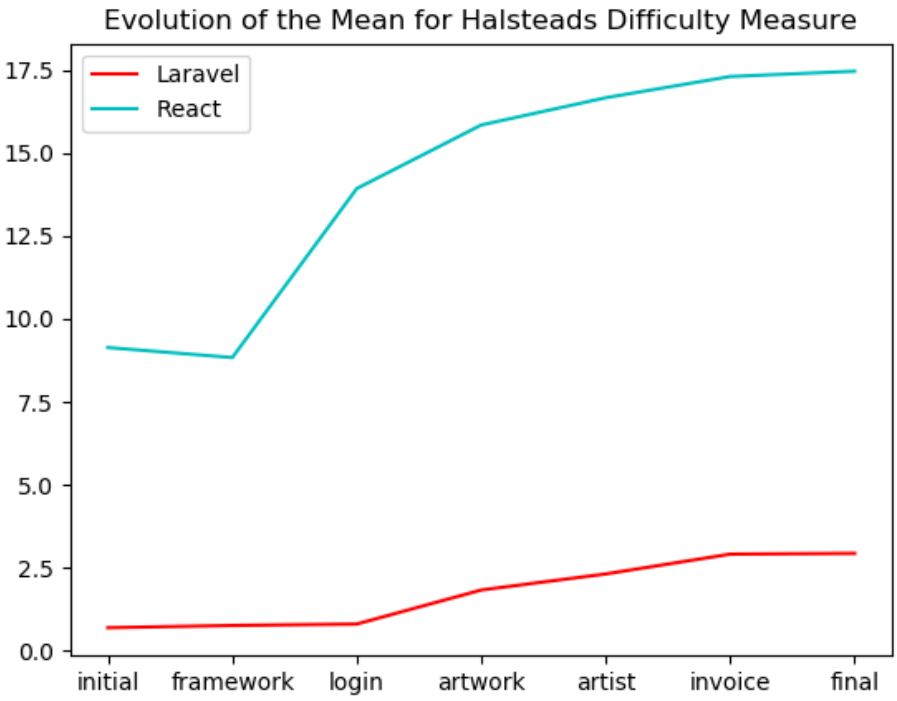


Figure 7.10.: Evolution of Halsteads Difficulty measure across all measured files for each sprint. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the arithmetic mean after the completion of the sprint shown on the X-axis. The evolution of this measurement is similar to the evolution of the other Halstead measures.

Filepath	Bytes on Disk
views/artworks/ArtworkForm.js	24734
views/artists/ArtistForm.js	20547
views/artworks/pdf/ArtworkInvoiceForm.js	17096
backend/controller/ArtworkController.js	12065
navs/Sidebar.js	9798
views/Register.js	9621
views/artworks/read/ViewArtwork.js	6912
backend/controller/ArtistController.js	6755
views/Login.js	6490
views/artists/read/ViewArtist.js	5773
views/ResetPassword.js	5358
src/serviceWorker.js	4951
views/artworks/ArtworkRow.js	4581
views/artists/Artists.js	4286
navs/AdminNavbar.js	4232

Table 7.3.: The 15 largest files ordered by their actual file size on disk of the ReactJS side of the experiment. This table shows that even though the exact ordering is different, 9 of the 15 files shown are also included in the 15 files with the largest value for Halsteads Volume. We can also see that the actual file size is vastly different from the amount of bits calculated using Halsteads Volume measure.

The file with the highest value for Effort for the Laravel side specifically is the one used for validating and processing of the invoice generation form. It contains a large amount of logic required to retrieve the involved HTML elements, code for validation and code for creating and sending requests to the back end, as well as handling responses. Generally, files related to User Experience (for the Laravel side of our project these files were Javascript files) are the ones with the highest value for Halsteads Effort measurement and also the ones which subjectively required the most amount of mental effort to develop for the Laravel side. Based on these subjective observations we would argue that Halsteads Effort measurement correctly reflects the mental effort required for the Laravel side of our project. However, over the course of the experiment we did not find a reliable way to objectively pinpoint the mental effort required to compare it to the results of Halsteads Effort measurement.

Filepath	Halstead Volume
views/artworks/ArtworkForm.js	13014.691
backend/controller/ArtworkController.js	12775.532
views/artists/ArtistForm.js	12089.87
views/artworks/pdf/ArtworkInvoiceForm.js	8163.526
backend/controller/ArtistController.js	6578.529
views/Register.js	4367.661
navs/Sidebar.js	4063.607
src/services/pdfService.js	3458.358
views/Login.js	3113.799
views/artworks/read/ViewArtwork.js	3032.552
backend/server.js	2944.16
backend/routes/authenticationRoutes.js	2869.399
views/ResetPassword.js	2852.92
backend/middleware/routeProtection.js	2763.653
views/artworks/ArtworkRow.js	2715.677

Table 7.4.: Values for Halsteads Volume measure for the ReactJS side of the experiment. Only the 15 files with the largest value for Halstead Volume are shown. Even though their ordering is different 9 of the 15 files shown are also part of the 15 largest files judging by the files size on the disk. It also demonstrates that the actual file size is vastly different from the amount of bits calculated using Halsteads Volume measure.

Additionally, for our experiment the results for Halsteads Effort measurement shown in Figure 7.11 suggest, that it correctly captures an important attribute of ReactJS. As the separation of concerns is one of the main focuses of the framework itself we expected to see a distribution of the mental effort required along multiple files. The results in Figure 7.11 show that for both frameworks we are dealing with some files which involve a significantly greater amount of mental effort required. However, for the ReactJS side we can see that apart from the five biggest files, the mental effort starts to evenly split across the remaining files. On the other hand, the Laravel side shows some files with an exceptionally high value for Halsteads Effort, while the rest of the files show low values. This result suggests that the difficulty of the entire software project is more spread out across the different architectural components for the ReactJS side, while for the Laravel side of our experiment some components involve significantly more effort than others.

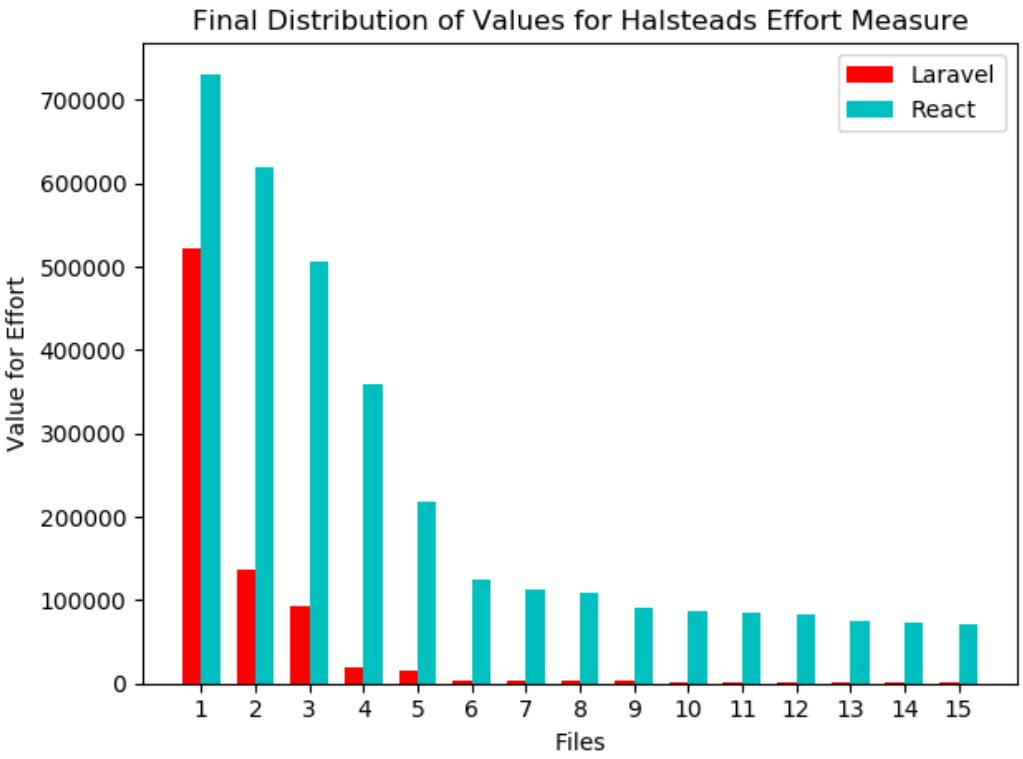


Figure 7.11.: Distribution of Values for Halsteads Effort measure after the `final` sprint for both frameworks. The chart shows the 15 files with the largest value for Halsteads Effort measure for both frameworks. The Y-axis shows the values, while the X-axis is a simple numeration from 1 (for largest) to 15 (to smaller). For both frameworks we can see that there is a small set of files which have a significantly larger value for the effort measurement as the rest of the files. Additionally the chart shows that the values for ReactJS are generally larger and more distributed across the files.

Bugs

Halstead originally assumed that the probability for an error can be seen as a relation between the complexity of the software and the experience of the developer, which he calls 'developer ability'. Therefore his initial formula uses Halsteads Effort measure in relation to a constant which represents the 'developer ability'. The tools used for this experiment use a deviation of the original formula which uses Halsteads Volume instead. Therefore, the amount of bugs in the software in the sense of Halsteads Bugs measurement can be seen as a relation between the size of the program and the 'developer ability'.

Since Halsteads Bugs measurement is directly relative to Halsteads Volume measurement the evolution over the sprints of the experiment is the exact same for both. Additionally, the 15 largest files - judging by Halsteads Volume measurement - are the ones with the highest amount of expected bugs. The distribution shown in Figure 7.12 suggests that for the Laravel side there are a few files which are expected to contain the most amount of bugs, while the rest of the files are expected to have almost none. For the ReactJS side of our experiment we see a similar trend, however the lower values are more spread out and generally higher. This is probably due to the fact that ReactJS applications usually spread the application logic across multiple smaller files (components).

For our experiment Halsteads Bugs measurement did not prove particularly meaningful. It proved useful for identifying files which are more likely to contain bugs, however as the measurement itself is just a deviation of Halsteads Volume we can just use Halsteads Volume instead of doing any further calculations. In the literature reviewed it was mentioned that Halsteads Bugs measure is useful for dynamic testing, as it can be seen as a lower bound for the amount of bugs which should be found by the testing suite. However, we did not find any additional information on this topic.

Time

Halsteads measure for Time tries to map his measure for Effort to conceivable units of time. Specifically, the formula presented in subsection 6.4.4 returns an estimate

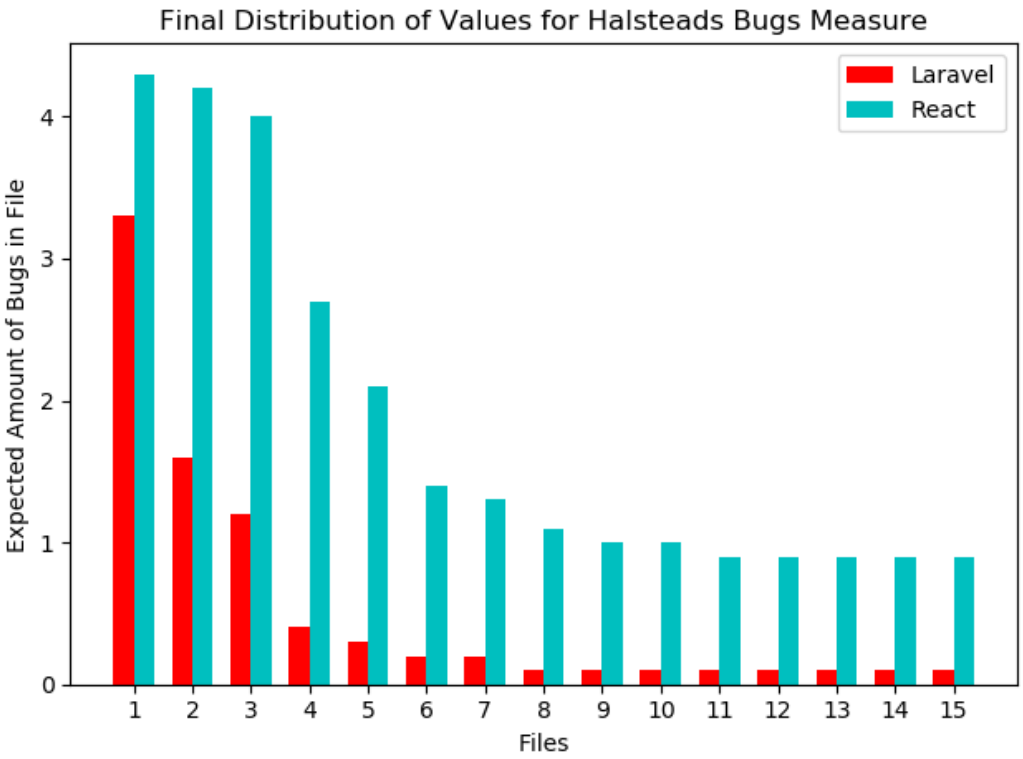


Figure 7.12.: Barchart showing the distribution of values for the 15 largest files judging by Halsteads Bugs measure for both frameworks after the `final` sprint. The Y-axis shows the resulting value for Halsteads Bugs measure, which translates to the expected amount of errors in the specific files. The X-axis shows a numeration for each of the 15 largest files ordered by the resulting value for Halsteads Volume measure. Since the formula we used to calculate Halsteads Bugs measure is directly relative to Halsteads Volume measure, the files with the largest Halstead Volume are the ones which are expected to contain the most amount of errors.

for the amount of seconds required to develop the measured program.

Figure 7.13 shows the evolution of the sum for Halsteads Time measure for all measured files. The results show a significant difference between the ReactJS and the Laravel application. However, this results seems unjustified.
While we did indeed perceive a significant difference of time investment needed between the two applications, the suggest ratio is too far off from reality. The graph indicates a value close to 250.000 seconds for the ReactJS side which roughly translates into 70 hours of work. For the Laravel side the graph shows a value close to 45.000 seconds, which is roughly 12 hours. These results suggest the ReactJS application almost took 6 times longer to develop than the Laravel application, while in reality roughly (lower bound for both) 150 hours were spent on the ReactJS application and 100 on the Laravel application.

This huge disconnect from the actual development time is probably due to various reasons. As discussed in [55] too many discrepancies are involved in the measurements used to calculate Halsteads Time measure, while the theoretical basis the Time measurement itself builds upon is also questionable.

For our experiment the results for Halsteads Time measure did not prove to be particularly meaningful. It might still be useful to identify files which consumed most of the development time, however we can also identify those by using any of the more basic Halstead measures which capture the size of files and are less criticized such as Volume or Length.

7.5. Logical Lines of Code

The logical lines of code (LLOC) are the most basic measurement for size used during this experiment. Figure 7.14 shows the almost identical evolution of the arithmetic mean for the LLOC for both sides of the experiment over the course of the development. During the first three sprints of the experiment (`initial`, `framework-setup`, `login`) we can see drastic shifts for the values of the arithmetic mean. This is caused by the design template which was added during these sprints. The example files contained by the design template lead to a

Figure 7.13.: Evolution of the sum of Halsteads Time measure across all measured files for each sprint. The values on the Y-axis are shown in seconds. The huge difference between the time needed measured by Halsteads Time measure for both sides of our experiment seems unjustified. While there was a significant difference of time investment between the two frameworks, the difference captured by Halsteads Time measure seems too big. Additionally, the resulting values of the measurement are too far off: The graph indicates a value close to 250.000 seconds for the ReactJS side which roughly translates into 70 hours of work. For the Laravel side the graph shows a value close to 45.000 seconds, which is roughly 12 hours. These results suggest the ReactJS application almost took 6 times longer to develop than the Laravel application, while in reality roughly (lower bound for both) 150 hours were spent on the ReactJS application and 100 on the Laravel application.

significant increase in the amount of LLOC when they were added during the `framework-setup` sprint for the Laravel side of the experiment and during the `login` sprint of the ReactJS side. The impact of these files is shown more clearly by Figure 7.15. Figure 7.15 also shows the decrease of LLOC for the Laravel side after the initial scaffolding was adjusted to fit the requirements during the `login` sprint.

7.5.1. Interpretation

The LLOC measurement was mainly used to calculate the Maintainability Index. Nonetheless our results suggest that it can be useful for detecting files which contain too much logic compared to other files in the architecture. Additionally, the connection between quantitative attributes of the code (the amount of lines of code) and the resulting value of the LLOC measurement is clear. This means, if we would use the LLOC measurement as a measurement of complexity, we could decrease the complexity of a program by reducing the amount of lines of code.

The results of our experiment suggest, that the software built with ReactJS requires more LLOC for building the same feature built in Laravel. Additionally, our results (see Figure 7.15) correctly capture the significant overall difference between the amount of LLOC between each framework. This difference is mainly due to the absence of authentication logic for the Laravel side of the experiment, since authentication is solely handled by Laravel directly and therefore lead to little amounts of LLOC. For the ReactJS side on the other hand it was necessary to develop the entire authentication logic from the ground up.

Our results correctly capture Laravels strength of providing a significant amount of basic functionalities (as for example authentication logic) out of the box. Furthermore, they also suggest that the LLOC measurement captures Laravels notion of 'doing more with less'. Particularly, Laravel tries to provide simple interfaces and a multitude of helper functions aimed at reducing the amount of code necessary to develop certain features. Figure 7.15 indicates that the Laravel side has a slightly less steep increase for the amount of LLOC starting from the `login` sprint until the end of the `final` sprint. This suggests, that less code is required to develop

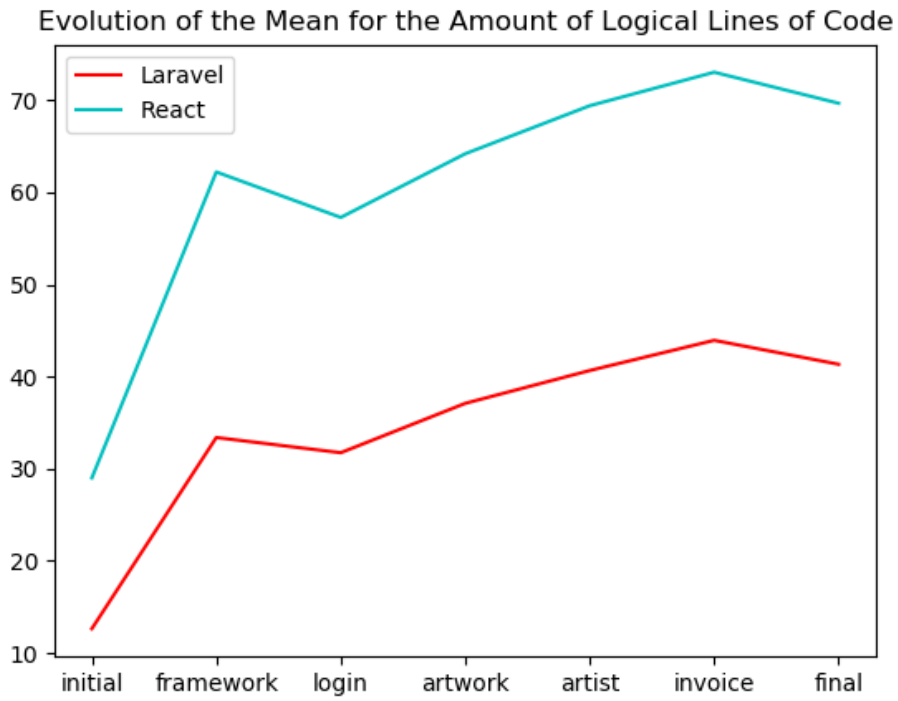


Figure 7.14.: Evolution of the arithmetic mean of the amount of logical lines of code across all measured files for each sprint. The graph shows the almost identical evolution for both sides of the experiment. There are significant jumps for the value of the arithmetic mean during the first three sprints. During these sprints a design template was added to the existing framework skeletons. At the end of the `login` sprint both sides of the experiment included the same scaffolding needed for future developments. Specifically, between the `initial` and `framework-setup` sprint example files were added to the project, which were removed during the `login` sprint. The increase of the arithmetic mean for the logical lines of code is caused by the initial addition of these example files, and the decrease is caused by their deletion/adjustment during the `login` sprint.

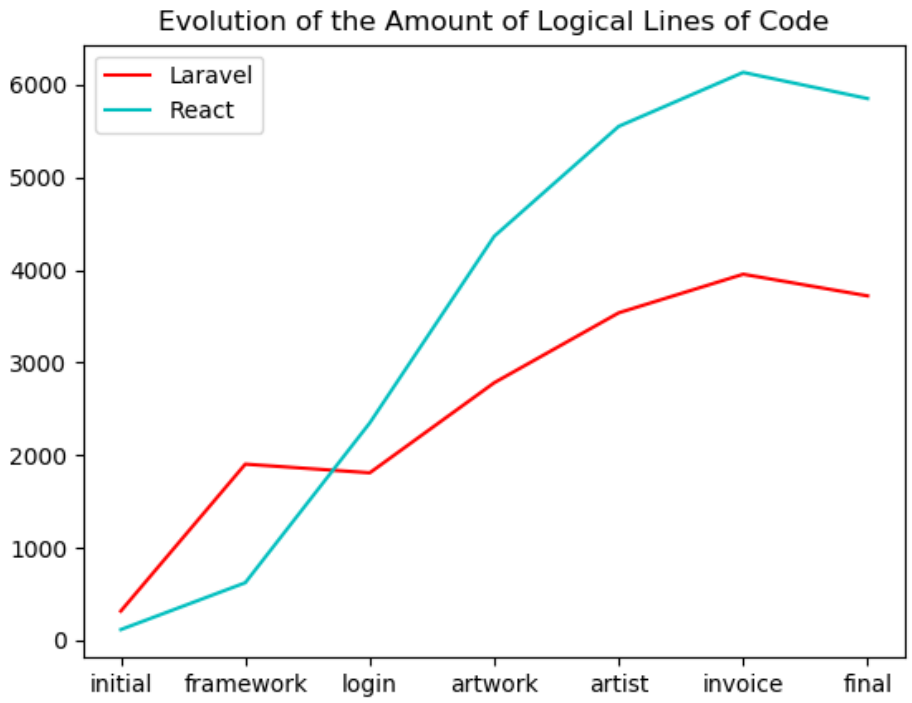


Figure 7.15.: Evolution of the total amount of logical lines of code (LLOC) over the course of the development. The LLOC were calculated for each file and summed up for each sprint. We can see the steady increase for both sides of the experiment and the almost identical evolution after the `login` sprint. The Laravel side of the experiment shows a significant increase at the end of the `framework-setup` sprint, since during this sprint the scaffolding of the design template was added to the code base. For the ReactJS side the same design template was added at a later point in time. We can see this increase for the ReactJS side after the `login` sprint. While the evolution is almost identical, the ReactJS side shows a slightly steeper increase past the `login` sprint.

equivalent functionalities on the Laravel side of our experiment, relative to the ReactJS side.

7.6. Cyclomatic Complexity

Figure 7.16 shows the evolution of the arithmetic mean of McCabes Cyclomatic Complexity measure. We can see that the value of the cyclomatic complexity (CC) for the ReactJS side of the experiment significantly drops between the `initial` and `framework-setup` sprint. One could interpret this result in such a way that the ReactJS application instantiated by `create-react-app` introduces a lot of overhead or unnecessary complexity to the project, which is removed after moving on with the initial setup of the project. However, for our experiment this is not the case.

The reason for this initial drop of CC is that during the `framework-setup` sprint additional views and other files were added to the project; each of which have a low value for the CC measure. Since the graph shows the evolution of the arithmetic mean, the addition of new files with low values for the CC measure lead to a significant decrease of the value for the arithmetic mean. Particularly, as Figure 7.17 shows the initial setup only contained 4 files, where 3 (views) had a CC of 1, and one file which had a CC of 18. After the `framework-setup` sprint however, the project already contained 10 files which were included in the measurements.

The remaining evolution of the measure follows the expected results. As additional features are added to the software, existing files grow and additional application logic leads to more conditional statements, which in turn lead to a steady increase of the value for the arithmetic mean for the CC measure.

7.6.1. Interpretation

We can see that on average, the Laravel project shows a significantly lower value for the CC measure. This makes sense, because of the difference between the two

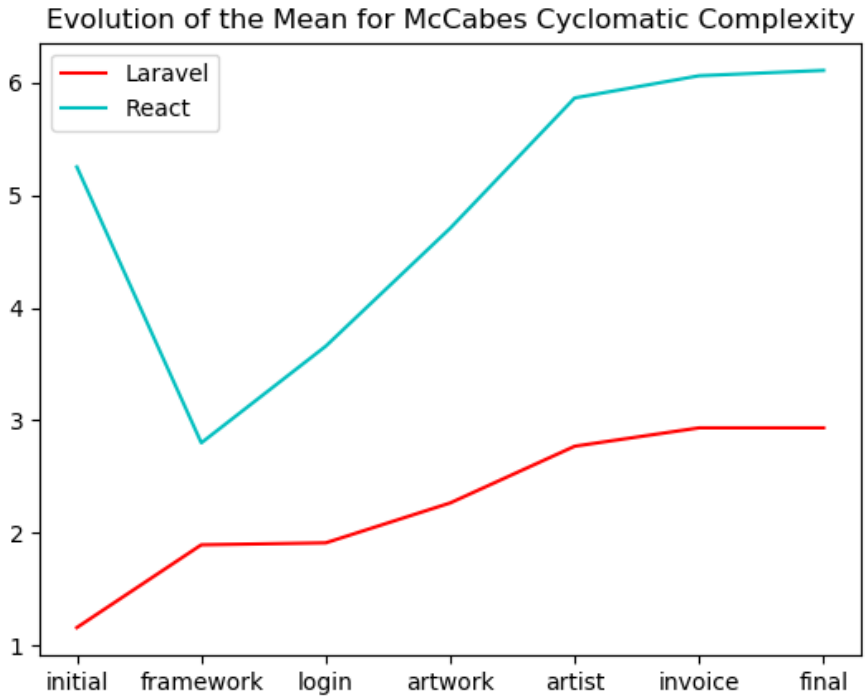


Figure 7.16.: Evolution of the arithmetic mean for McCabes Cyclomatic Complexity measure. The Y-axis shows the values for the Cyclomatic Complexity measure and the X-axis indicates the different sprints. The huge drop between the `initial` and `framework-setup` sprint for the ReactJS side of the experiment is due to the low amount of files at the beginning of the projects as shown in Figure 7.5. A single file with a large value for the Cyclomatic Complexity measure lead to a high value for the arithmetic mean of the measure. After additional files were added the impact of outliers was reduced which resulted in a drop for the value of the measurement.

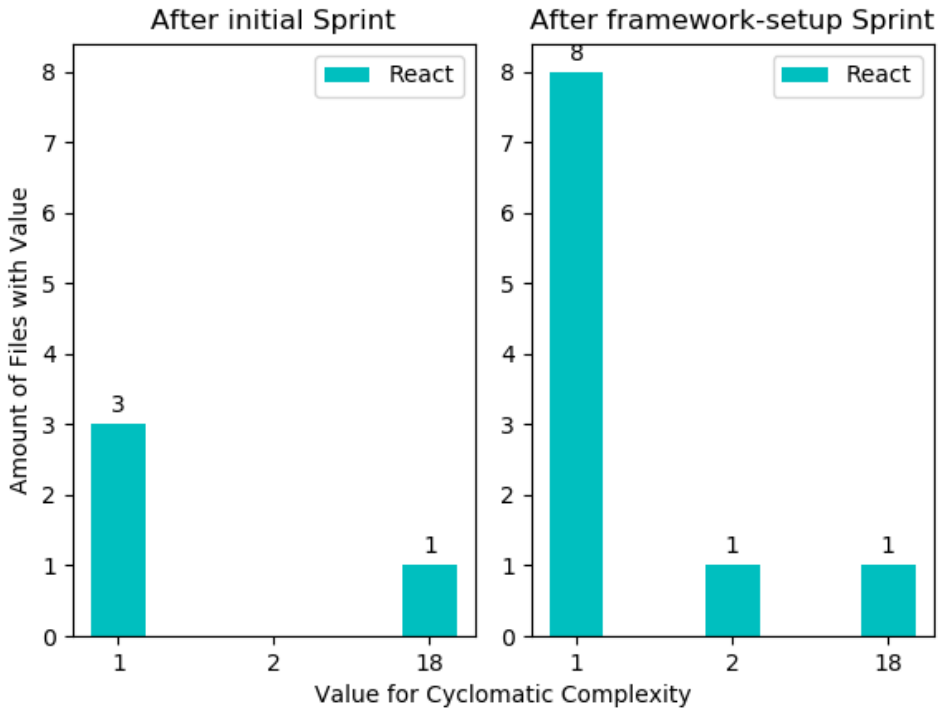


Figure 7.17.: Barcharts visualizing the distribution of values for McCabes Cyclomatic Complexity measure after the `initial` and the `framework-setup` sprint. The X-axis shows the values for the Cyclomatic Complexity measure. The Y-axis shows the amount of files with the specific value for the Afferent-Coupling measure shown on the X-axis. The numbers above the bars indicate the exact amount of files in this specific group. The number of files measured more than doubled between these two sprints. The wider spread of the values lead to a sudden decrease in the value of the arithmetic mean for the Cyclomatic Complexity measure, as the impact of outliers was lowered.

frameworks. The ReactJS side contains a variety of stand-alone components, each containing their own logic for managing themselves. This means, that on average the amount of conditional statements per file should be higher. The Laravel side on the other hand should have a small number of files, which have a high value for the CC measure. Specifically, the controllers and the views contain most of the applications logic. Apart from the controllers and the views however, the rest of the files contain little application logic and therefore only a small amount of conditional statements, which on average leads to a smaller value for the CC measure.

In Figure 7.18 we can see the distribution of values for the CC measure. It shows that for the Laravel side of our experiment most files show values of one or two for the CC measure. The ones with values above 10 are views with significant amount of logic for displaying information correctly and controllers which contain a significant amount of application logic for storing and updating resources. The ReactJS side does not only show higher values for the CC measure on average, it also has the files with the highest value for the CC measure.

The largest value for the CC measure on the ReactJS side of our experiment is 62. The file associated with this value is responsible for displaying a form to create and edit an artwork. The reason for such an alarming value for the CC measure is that this file contains both the logic for creating and updating an artwork, while also facilitating form validation and request generation. This case in particular shows that the CC measure can be useful to detect files which most likely can be redesigned or split up into smaller components to significantly reduce their complexity.

During our research we did not find a clear consensus on which values for the CC measure are alarming values (high complexity) and which ones indicate a low amount of complexity. In general we suggest to use the CC measure in relation to other files in the same project. As an example, in our experiment most files have a CC below 10 which leads to the assumption we should probably try to refactor files in a way such that they have a CC below 10 or at least as close to 10 as possible. Any file above such a threshold can be considered to have a high degree of complexity and should be looked into in more detail.

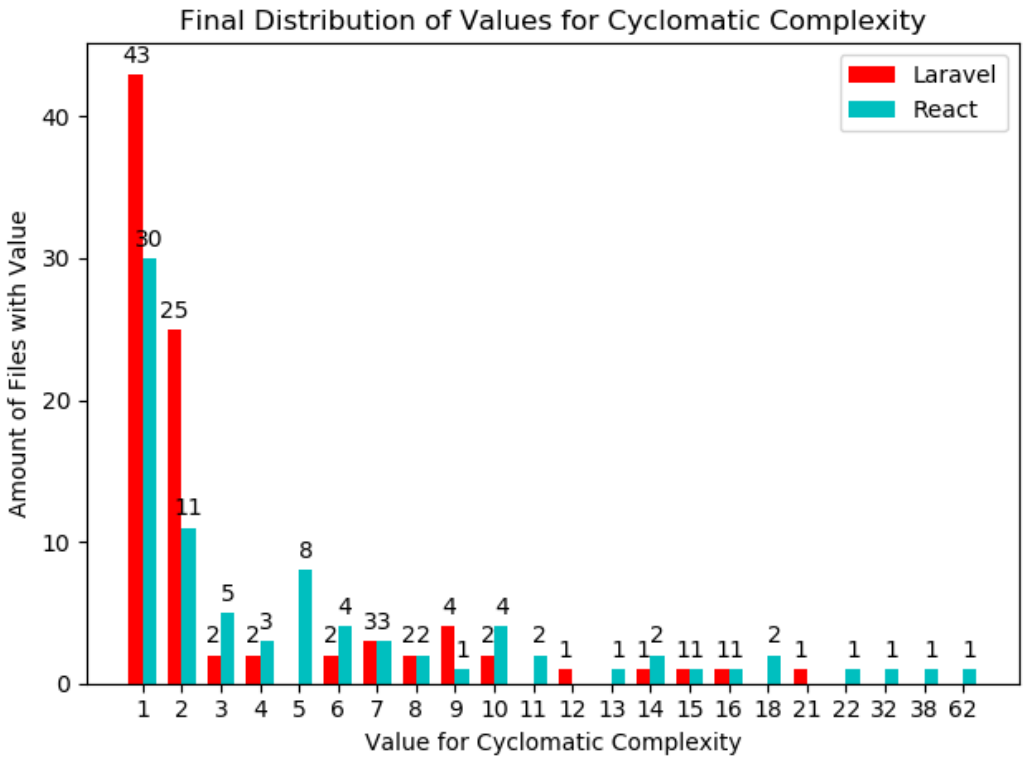


Figure 7.18.: Barchart visualizing the distribution of values for McCabes Cyclomatic Complexity measure after the `final` sprint. The X-axis shows the values for the Cyclomatic Complexity measure. The Y-axis shows the amount of files with the specific value for the Cyclomatic Complexity measure shown on the X-axis. The numbers above the bars indicate the exact amount of files with this specific value. We can see that for the Laravel side of our experiment most files have a cyclomatic complexity (CC) of one or two. This is the case because the Laravel side of our experiment contains a significant amount of helper files, which are used to extend the framework. An example for this would be using 'Requests' to define rules for automatic request validation. These files usually contain a single function without any additional logic, which results in a CC of one. Only a few files show a CC above 10, while for the ReactJS side the values are fairly spread out between 2 and 10. For the ReactJS side there is also a significant amount of files with values above 10.

7.7. Maintainability Index

The Maintainbility Index by Coleman et al. [18] is a combination of different software measurements, namely Halsteads Volume [32], McCabes Cyclomatic Complexity [44] and logical lines of code. According to Coleman et al., each of these software measurements is assumed to capture different attributes of software which have a negative impact on its maintainability.

Figure 7.19 shows the evolution of the arithmetic mean for the Maintainability Index over the course of the experiment. Our results show that the Laravel side of our experiment has a higher value for the Maintainability Index right from the initial setup. The evolution is similar for both frameworks; as the project grows the value for the arithmetic mean of the Maintainability Index steadily decreases. Interestingly, we observed an increase of the value for the arithmetic mean of the Maintainability Index for the ReactJS side between the `framework-setup` and `login` sprint. This increase is due to the significant increase in the amount of files measured as shown in Figure 7.5. The addition of multiple files with higher values for the Maintainability Index lead to a positive shift in the value of the arithmetic mean. For the Laravel side the only positive change for the Maintainability Index observed during our experiment was after the `final` sprint, while any other sprint lead to a decrease of the value.

7.7.1. Interpretation

With the high degree of separation of concerns for the ReactJS application we expected to see relatively high values for the arithmetic mean of the Maintainability Index. As logic and therefore complexity as well as amount of lines of code (LOC) is split across multiple files, we should end up with numerous files with a relatively low amount of LOC, low complexity and therefore higher Maintainability Index.

However, due to the properties of the Maintainability Index and of the attributes of the Laravel framework our results show that the Laravel side of our experiment has higher values for the Maintainability Index. The Maintainability Index combines

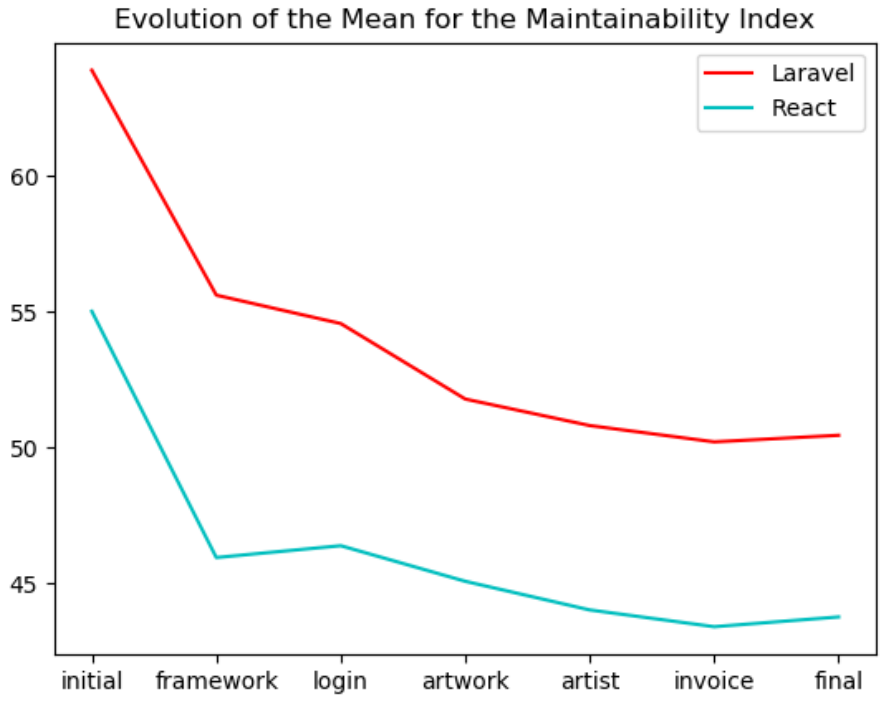


Figure 7.19.: Evolution of the arithmetic mean for the Maintainability Index [18]. Larger values indicate better maintainability. The Y-axis shows the values for the arithmetic mean of the Maintainability Index and the X-axis indicates the different sprints. The drop between the `initial` and `framework-setup` sprint for both sides of the experiment is due to the low amount of files at the beginning of the projects as shown in Figure 7.5. After the initial setup only a few files containing almost no logic and therefore with a Maintainability Index of 100 are contained in the project. After the `framework-setup` sprint basic scaffolding to display information was added to the project, which lead to the addition of multiple files with a Maintainability Index below 100, which lead to a decrease of the value of the arithmetic mean.

three measurements, of which two are considered a measurement of size: Halsteads Volume and logical lines of code (LLOC). The formula used for our experiment is

$$\begin{aligned} Maintainability = (& \\ & 171 \\ & - 5.2 \times \ln(HalsteadVolume) \\ & - 0.23 \times CyclomaticComplexity \\ & - 16.2 \times \ln(LLOC) \\ &) * 100/171 \end{aligned} \tag{7.1}$$

, which is a slight deviation from Coleman et al. [18] original formula. The deviations and reasonings behind them are discussed thoroughly in section 6.5. In the formula used, the two main factors negatively influencing the Maintainability Index are foremost the amount of logical lines of code (LLOC) and Halsteads Volume. While the complexity by the means of McCabes Cyclomatic Complexity has an impact on the measurement, the size of the file - especially in terms of LLOC - has a significantly higher impact on the measurement.

Halsteads Volume was attributed to capture not only the size of the file, but also to capture the complexity contained in the file. However, as shown by Shen et al. [55] and discussed by Fenton et al. [26, p. 344-348] the theoretical background of Halsteads measurements is questionable and Fenton et al. suggest to classify Halsteads Volume solely as a measurement of size.

Assuming Halsteads Volume is a measurement of size, the main factor responsible for a decrease of the Maintainability Index for our experiment is the size of the measured file. This means, that the smaller a file is in terms of LLOC and Halstead Volume, the higher the Maintainability Index of that file is.

One of Laravels focuses is to 'do more with less', specifically this means that the framework provides a multitude of simple interfaces and basic functionalities out of the box which aim at reducing the amount of code necessary to reach a certain goal. For our experiment Figure 7.15 shows a slight difference in the amount of LLOC required for the same set of features. Since for our experiment the Maintainability Index is mainly influenced by the size of the files, the Laravel side generally has a

higher Maintainability Index than the ReactJS side, since the files of the Laravel side are smaller (in the sense of Halstead Volume and LLOC).
An additional aspect, which leads to a higher value for the arithmetic mean of the Maintainability Index for the Laravel side, is a specific set of simple yet powerful files, which we will call 'helper files'. The Laravel framework contains a multitude of functionalities which are related to back end development out-of-the-box. For example, the ability to set the application into a maintenance mode is included in every project and it can be triggered by setting a specific flag in the underlying database. Such features are included in the project as simple files which contain little logic, as they are supposed to be used as configuration files. Additionally, files defining the layout of a request and validation rules also contain little logic even though they play an important role in the application. These helper files specifically have a significantly high value for the Maintainability Index, most of the time ranging between 80 and 100. Therefore, these files cause a significant positive shift of the value of the arithmetic mean of the Maintainability Index. We think that this is an important observation and connection between the measurement and quantitative code attributes, since splitting functionalities up into small pieces and have single files which are responsible for one specific part of the application is considered to have a positive impact on perceived maintainability.

Figure 7.20 shows the distribution for the Maintainability Index for both sides of our experiment after the `final` sprint. The sections 'critical' ($[0, 10]$, red), 'ok' ($(10, 20]$, yellow), and 'good' ($(20, 100]$, green) visualize the classification used by Visual Studio [47]. These thresholds still have to be tested until a clear consensus can be found. However, for our experiment they served as a good indicator. As Figure 7.20 shows the ReactJS side of our experiment contains two files which have a particularly low Maintainability Index, which lies in the 'critical' interval. Specifically, these files are the forms used to create and edit the resources associated with the system, which means they contain logic for validating user input, general user feedback and creating requests which are then sent to the back end. Most of these functionalities could be split up into multiple smaller files, which would lead to a higher Maintainability Index. In this case, the Maintainability Index correctly indicated bad design choices and a low perceived maintainability.
The rest of the distribution does not show anything of particular interest, however we can see that for the Laravel side the files tend to generally have higher values for the Maintainability Index.

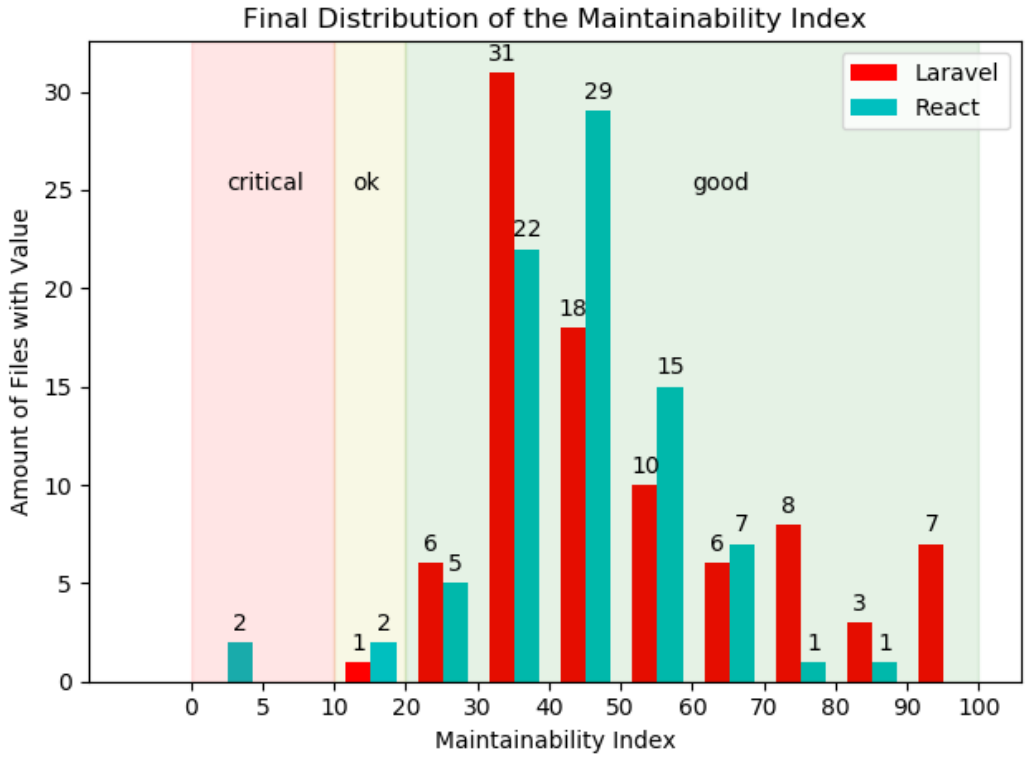


Figure 7.20.: Barchart visualizing the distribution of values for the Maintainability Index after the `final` sprint. The X-axis shows intervals for of the Maintainability Index. Each interval is inclusive f.e. $(5, 10]$, $(10, 20]$, . . . , $(90, 100]$. The Y-axis shows the amount of files with a Maintainability Index in the interval specified on the X-axis. The numbers above the bars indicate the exact amount of files in this specific interval. The sections 'critical' ($[0, 10]$, red), 'ok' ($(10, 20]$, yellow), and 'good' ($(20, 100]$, green) visualize the classification used by Visual Studio [47]. Specifically for our experiment we can see that the ReactJS side has two files with an alarmingly low value for the Maintainability Index, which can probably be refactored and simplified. The worst file for the Laravel side on the other hand has a Maintainability Index between 10 and 20 which is considered to be an alarming value. However, values between 10 and 20 are often files which have a significant amount of responsibility and probably can not be split up into smaller files, as there are logical reasons behind having everything in one place. For our experiment in particular the file with the lowest value for the Maintainability Index is in fact a controller which contains all the logic associated with working with a specific type of resource. While it would be possible to split up the logic of the controller into multiple smaller classes, it is often desired to have all logic associated with a single resource in a single file.

7.8. Meaningfulness of Derived Statements

When drawing conclusions based on data collected using any form of measurement we have to be mindful of the scale the measurement operates on. Depending on the scale, certain statistical operations might not be meaningful and can lead to wrong results. Specifically, most of the time the measurements we use will result at least in an ordinal scale. This type of scale assigns a specific ordering to the objects measured and disregards the specific intervals between each measured object. However, applying the arithmetic mean to the result of a measurement which operates on an ordinal scale is not meaningful, since the intervals between each measured object may vary drastically. To use statistical operations such as the arithmetic mean and retrieve meaningful results, the scale our measurement operates on has to be at least an interval scale.

For the measurements used during this experiment only McCabes Cyclomatic Complexity [44] was shown to use a ratio scale [66, p. 151 - 167]. This means when interpreting the results of this experiment we have to be mindful of the meaningfulness of derived statements. Since most of our measurements use an ordinal scale, the alternative to the arithmetic mean would be to look at the median, which is meaningful for ordinal scales. During our experiment we calculated both, the median and the arithmetic mean. The evolution of the median for all measurements differs from the results received from the arithmetic mean. For some of the measurements used the median does not change at all, which made it less useful for our experiment. For this reason we decided to go into detail with the values received from the arithmetic mean, since it is known that even though theoretically it might not be meaningful to use the arithmetic mean on an ordinal scale, it can still lead to 'fruitful results' [58].

7.9. Applicability of the selected Software Measurements for Framework Comparison

The results discussed in the previous section show that not all software measurements chosen for this experiment are suitable for comparing the selected frameworks reliably, and that not all of our expectations stated in subsection 5.3.3 were

Measurement	Laravel	React
Maintainability Index [18]	50.464	43.777
Cyclomatic Complexity [44]	2.933	6.107
Afferent-Coupling [43]	0.989	2.238
Efferent-Coupling [43]	2.089	4.5
Instability [43]	0.712	0.66
Halstead Effort [32]	9068.081	53282.042
Halstead Volume [32]	336.507	1733.059
Halstead Length [32]	56.5	259.06
Halstead Time [32]	503.77	2960.113
Halstead Bugs [32]	0.112	0.578
Halstead Difficulty [32]	2.937	17.466
Halstead Vocabulary [32]	17.011	67.833
Logical Lines of Code	41.333	69.655

Table 7.5.: Overview of the results received for the different measurements used. The columns show the different measurements while the respective row for each framework contains the values of the arithmetic mean for each of the measurements. The values shown are the result of the measurements after the `final` sprint.

met.

For example, the coupling measures used during this experiment [43] seemed to correctly capture the desired attributes of coupling between classes and files in general for the ReactJS side. However, Laravel automatically imports a vast amount of classes into every file associated with the framework, effectively hiding a significant amount of interrelationships between the measured classes. Therefore, the coupling measures used during this experiment can not be reliably used when trying to compare the Laravel framework to another framework. Nonetheless we would like to point out, that our results for the ReactJS side indicate that the coupling measures in particular change with any changes in the quantitative attributes (increase in outgoing and incoming dependencies) and therefore can reliably be used to measure the coupling of a software project.

This highlights the problem that the applicability of software measures for framework comparison mostly depends on which frameworks are compared and which

programming languages they use. The programing language used has a significant impact on Halsteads measures, as the definitions for operators and operands differ for each language and the frequency of their usage may also vary drastically.

We believe that the software measures used can be useful for detecting files with high complexity. However, the selected set of software measures does not seem to be particularly reliable for comparing the selected frameworks with each other. This is mainly due to the various differences in the frameworks and programming languages used, which directly impact the measurements as for example Laravels automatic imports.

We also believe that if we were to inspect the evolution of our software project over a longer period of time, we would, over time, see higher values for the Maintainability Index of the ReactJS side relative to the value of the Laravel side. This is due to the fact, that the value for the arithmetic mean of the Maintainability Index of the Laravel side steadily decreased over the course of development (apart from the slight increase observed after the refactoring done during the `final` sprint) while it increased for the ReactJS side after the addition of multiple new files during the `login` sprint. We think that the high degree of separation of concerns will over time lead to better results for the Maintainability Index of the ReactJS side when compared to the Laravel side.

Furthermore, we realized that while some measures are relatively easy to influence as there is a clear connection between code attributes and the resulting value of the measurement, there are others for which it is hard to find a clear solution of how to improve the code in such a way that it is reflected in the measurements. For example, if we wanted to reduce the coupling measured using the coupling measures used during this experiment [43] we could reduce the amount of import statements and this change would immediately result in lower values for the Afferent- and Efferent-Coupling measures. In a similar way we can reduce the complexity measured by McCabes Cyclomatic Complexity [44] by reducing the amount of conditional statements in a file.
However, particularly the Maintainability Index proved rather difficult to influence. Reducing the complexity (in the sense of McCabes Cyclomatic Complexity [44]) leads do slightly higher values for the Maintainability Index. For our experiment, the main factor influencing the Maintainability Index was the size of the file in the

sense of logical lines of code (LLOC) and Halstead Volume.

As software projects grow, it is likely that certain files will reach a significant amount of LLOC, even though most of the logic is already split up into other architectural components. In such a case, it seems impossible to improve the Maintainability Index of a file at which point the applicability of the Maintainability Index becomes questionable. Additionaly, Halsteads Volume, the third measurement included in the calculation of the Maintainability Index, also depends on the amount of operators, operands, unique operators and unique operands. Just like LLOC these amounts depend on the size of the file and most of the time it can prove difficult to lower the amount of operators needed to achieve a certain goal.

McCabes Cyclomatic Complexity [44], Martins coupling measures [43] and logical lines of code are perceivably linked to quantitative code attributes and therefore easy to influence. On the other hand, especially Halsteads measurements [32] and the Maintainability Index [18] proved to be relatively difficult to influence and did not provide much information on how to specifically improve the code once reducing the size of the file was not possible. Antinyan et al. [3] highlight this flaw of existing software measurements and draw attention to the need for new software measures targeted at specific code attributes which are known to cause high complexity.

8. Conclusions and Future Work

During this experiment we investigated a set of software measurements aimed at quantitative software attributes and their applicability for framework comparison. Specifically, we build the same application in two different web frameworks namely ReactJS and Laravel and defined clear intervals which we call 'sprints'. During each of these 'sprints' a specific set of features was added to both sides of the experiment. After each sprint, the following software measurements were used:

1. Afferent-Coupling, Efferent-Coupling and Instability [43]
2. Logical Lines of Code (LLOC)
3. Cyclomatic Complexity [44]
4. Halsteads Metrics [32]
    a) Vocabulary
    b) Length
    c) Volume
    d) Difficulty
    e) Effort
    f) Time
    g) Bugs
5. Maintainability Index [18]

The main take aways of this thesis are as follows:

- The applicability of software measures depends on the framework and programming language involved. This includes both, their applicability for frameworks in general and the applicability for framework comparison. As an example our results suggest, that the Coupling measures by Martin [43] are not suitable for measuring the coupling of a Laravel application. On the other hand, the results for the ReactJS side suggest that the Coupling measures capture attributes of the software architecture.

- Some measures proved particularly useful as they can be used to detect certain attributes which are directly linked to perceivable quantitative attributes of the software. Specifically McCabes Cyclomatic Complexity [44] was helpful at detecting files with significant amounts of conditional statements, which for our experiment were also perceivably more complex to adjust than other files. For the ReactJS side of our experiment Martins Coupling measures [43] were used for two different purposes.
  First of all we were able to identify files which have a relatively high probability to change since they depend on a multitude of other components, which results in high values for the Efferent-Coupling measure.
  Secondly we were able to identify core entities of our architecture as they were highlighted by a relatively high amount of incoming dependencies, and therefore high values for the Afferent-Coupling measure.
- On the other hand, the remaining software measures used did not prove particularly useful. While Halsteads Measures and the Maintainability Index can be used to detect files with a significant amount of responsibility / application logic, their usefulness beyond that is questionable. If a file gets a low value for the Maintainability Index it is often unclear which changes would help improve the file in the sense of the Maintainability Index, as the connection between the result of the measurement and quantitative attributes of the measured software is unclear. The Halstead Measures and the Maintainability Index both seemed to be mainly influenced by the size of the software measured.
- Optimally, the set of software measurements used for this experiment is applied to two projects using the same programming language. The results of Halsteads measures in particular highly depend on the programming language used, which makes them less suitable for comparing two projects which are written in different programming languages, like the ones investigated during this experiment.

8.1. Future Work

The field of software measurements leaves much to be desired. Existing software measurements have to be validated, the scale they operate on has to be clearly defined and new measurements with a clear connection to quantitative software

attributes have to be found. Additionally, software to quickly integrate software measurements into existing projects has to be developed.

Future work should focus on the development of new software measures aimed at harmful and quantitative software attributes as suggested by Antinyan et al. [3]. Further, a clear consensus for the applicability of existing software measures has to be found. Halsteads measures in particular depend on the definition of operators and operands for each programming language. Without a clear consensus, different software measurement tools could lead to different results.
Additionally, software which allows the calculation of software measurements at least for a specific programming language could be an important step towards investigating and potentially validating the applicability of existing software measurements.

Bibliography

[1] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra. Empirical study of object-oriented metrics. *Journal of Object Technology*, 5(8):149–173, 2006.

[2] M. AlSharif, W. P. Bond, and T. Al-Otaiby. Assessing the complexity of software architecture. In *Proceedings of the 42nd annual Southeast regional conference*, pages 98–103. ACM, 2004.

[3] V. Antinyan, A. B. Sandberg, and M. Staron. A pragmatic view on code complexity management. *Computer*, 52(2):14–22, 2019.

[4] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95, 1993.

[5] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[6] D. Bell and J. E. Sullivan. *Further investigations into the complexity of software*. Mitre Corporation, 1974.

[7] C. Berge. Graphs and hypergraphs. 1973.

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[9] B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. Cost estimation with cocomo ii. *ed: Upper Saddle River, NJ: Prentice-Hall*, 2000.

[10] B. Boehm, R. Valerdi, J. Lane, and A. Brown. Cocomo suite methodology and evolution. *CrossTalk*, 18(4):20–25, 2005.

[11] B. W. Boehm, J. R. Brown, and H. Kaspar. Characteristics of software quality. 1978.

[12] B. W. Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[13] M. Bunge. *Treatise on basic philosophy: Ontology I: the furniture of the world*, volume 3. Springer Science & Business Media, 1977.

[14] M. Bunge. *Treatise on basic philosophy: Ontology II: A world of systems*, volume 4. Springer Science & Business Media, 2012.

[15] Z. Chang, R. Song, and Y. Sun. Validating halstead metrics for scratch program using process data. In *2018 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*, pages 1–5. IEEE, 2018.

[16] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[17] M. Clark, B. Salesky, C. Urmson, and D. Brenneman. Measuring software complexity to target risky modules in autonomous vehicle systems. 2008.

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

[19] B. Curtis. In search of software complexity. In *Workshop on quantitative software models*, pages 95–106, 1979.

[20] B. Curtis. The measurement of software quality and complexity. *Software Metrics: An Analysis and Evaluation*, pages 203–224, 1981.

[21] W. H. DeLone and E. R. McLean. Information systems success: The quest for the dependent variable. *Information systems research*, 3(1):60–95, 1992.

[22] W. H. Delone and E. R. McLean. The delone and mclean model of information systems success: a ten-year update. *Journal of management information systems*, 19(4):9–30, 2003.

[23] Y. Deng, P. Frankl, and J. Wang. Testing web database applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, Sept. 2004.

[24] S. Dhillon and Q. H. Mahmoud. An evaluation framework for cross-platform mobile application development tools. *Software: Practice and Experience*, 45(10):1331–1357, 2015.

[25] B. M. Edmonds. *Syntactic measures of complexity*. University of Manchester Manchester, UK, 1999.

[26] N. Fenton and J. Bieman. *Software metrics: a rigorous and practical approach, 3rd edition*. CRC press, 2014.

[27] A. Fitzsimmons and T. Love. A review and evaluation of software science. *ACM Comput. Surv.*, 10(1):3–18, Mar. 1978.

[28] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[29] R. D. Gordon and M. H. Halstead. An experiment comparing fortran programming times with the software physics hypothesis. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 935–937. ACM, 1976.

[30] A. S. GÜCEĞLİOĞLU. *THE DEPARTMENT OF INFORMATION SYSTEMS*. PhD thesis, Citeseer, 2006.

[31] A. S. Guceglioglu and O. Demirors. A process based model for measuring process quality attributes. In *European Conference on Software Process Improvement*, pages 118–129. Springer, 2005.

[32] M. H. Halstead et al. *Elements of software science*, volume 7.

[33] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing*, pages 336–341. IEEE, 2011.

[34] H. Heitkötter, T. A. Majchrzak, B. Ruland, and T. Weber. Evaluating frameworks for creating mobile web apps. In *WEBIST*, pages 209–221, 2013.

[35] H. Ji, S. Huang, Y. Wu, Z. Hui, and C. Zheng. A new weighted naive bayes method based on information diffusion for software defect prediction. *Software Quality Journal*, Jan 2019.

[36] C. F. Kemerer. Reliability of function points measurement: a field experiment. 1990.

[37] B. Kitchenham. Counterpoint: the problem with function points. *IEEE software*, (2):29–31, 1997.

[38] D. Krantz, D. Luce, P. Suppes, and A. Tversky. Foundations of measurement, vol. i: Additive and polynomial representations. 1971.

[39] J. Kriz. *Methodenkritik empirischer Sozialforschung: Eine Problemanalyse sozialwissenschaftlicher Forschungspraxis*. 1988.

[40] T. Kuipers and J. Visser. Maintainability index revisited–position paper. Citeseer.

[41] G. C. Low and D. R. Jeffery. Function points in the estimation and evaluation of the software process. *IEEE Transactions on Software Engineering*, 16(1):64–71, 1990.

[42] T. Majchrzak and T.-M. Grønli. Comprehensive analysis of innovative cross-platform app development frameworks. In *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[43] R. Martin. Oo design quality metrics. *An analysis of dependencies*, 12:151–170, 1994.

[44] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[45] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical report, GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

[46] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2006.

[47] Microsoft Code Analysis Team. Maintainability Index Range and Meaning. http://web.archive.org/web/20190131235829/https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/, 2007. [Online; accessed 21-August-2019].

[48] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient javascript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 74–83. IEEE, 2013.

[49] N. Mohamed, R. F. R. Sulaiman, and W. R. W. Endut. The use of cyclomatic complexity metrics in programming performance's assessment. *Procedia-Social and Behavioral Sciences*, 90:497–503, 2013.

[50] S. Morasca. Software measurement. In *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, pages 239–276. World Scientific, 2001.

[51] M. F. Oliveira, R. M. Redin, L. Carro, L. da Cunha Lamb, and F. R. Wagner. Software quality metrics and their impact on embedded software. In *2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, pages 68–77. IEEE, 2008.

[52] S. S. Rathore and S. Kumar. A study on software fault prediction techniques. *Artificial Intelligence Review*, 51(2):255–327, 2019.

[53] F. S. Roberts. *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences (Section, Mathematics and the Social Sciences)*. Cambridge University Press, 1984.

[54] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[55] V. Y. Shen, S. D. Conte, and H. E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. *IEEE Transactions on Software Engineering*, (2):155–165, 1983.

[56] S. B. Sheppard, P. Milliman, and B. Curtis. Experimental evaluation of on-line program construction. Technical report, GENERAL ELECTRIC CO ARLINGTON VA, 1979.

[57] A. Sommer and S. Krusche. Evaluation of cross-platform frameworks for mobile applications. *Software Engineering 2013-Workshopband*, 2013.

[58] S. S. Stevens et al. On the theory of scales of measurement. 1946.

[59] J. M. Stroud. The fine structure of psychological time. 1956.

[60] J. Sullivan. Measuring the complexity of computer software. *MITRE Corp. MTR-2648*, 1973.

[61] H. Van Vliet, H. Van Vliet, and J. Van Vliet. *Software engineering: principles and practice*, volume 13. Citeseer, 2008.

[62] J. M. Veiga and M. J. Frade. Treecycle: a sonar plugin for design quality assessment of java programs. *Techn. Report CROSS-10.07-1*, 2010.

[63] I. Vessey and R. Weber. Research on structured programming: An empiricist's evaluation. *IEEE Transactions on Software Engineering*, (4):397–407, 1984.

[64] A. H. Watson, D. R. Wallace, and T. J. McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of …, 1996.

[65] W. Wu, Y. Cai, R. Kazman, R. Mo, Z. Liu, R. Chen, Y. Ge, W. Liu, and J. Zhang. Software architecture measurement—experiences from a multinational company. In *European Conference on Software Architecture*, pages 303–319. Springer, 2018.

[66] H. Zuse. Software complexity. *NY, USA: Walter de Cruyter*, 3, 1991.

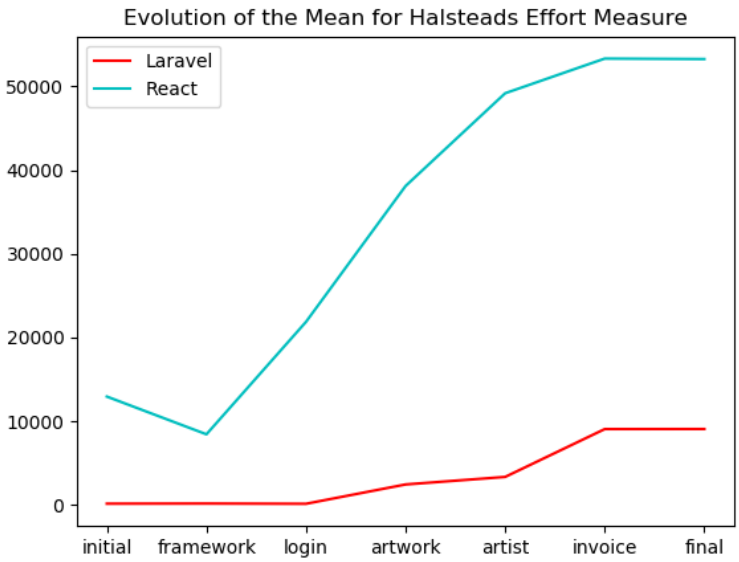
Appendix

Appendix A.

Plots for Halsteads Measures

Since the evolution is almost identical for most of the Halstead measures, not all of them were included in chapter 7. Following are the missing plots for the evolution of Halteads measures.

Appendix A. Plots for Halsteads Measures



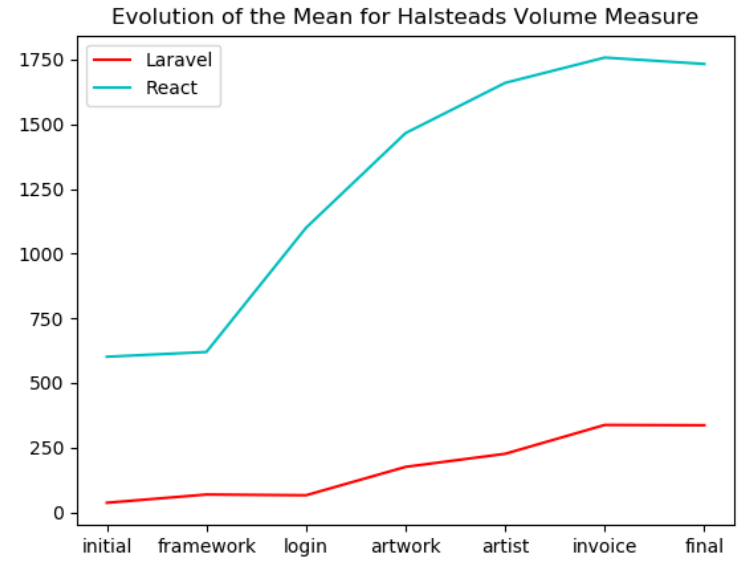
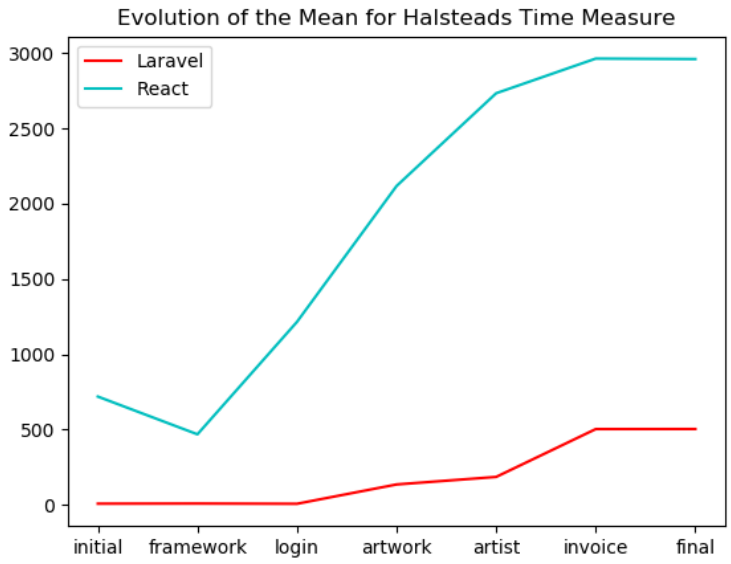


Figure A.1.: Evolution of the arithmetic mean of Halsteads Effort and Volume measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the arithmetic mean after the completion of the sprint shown on the X-axis.



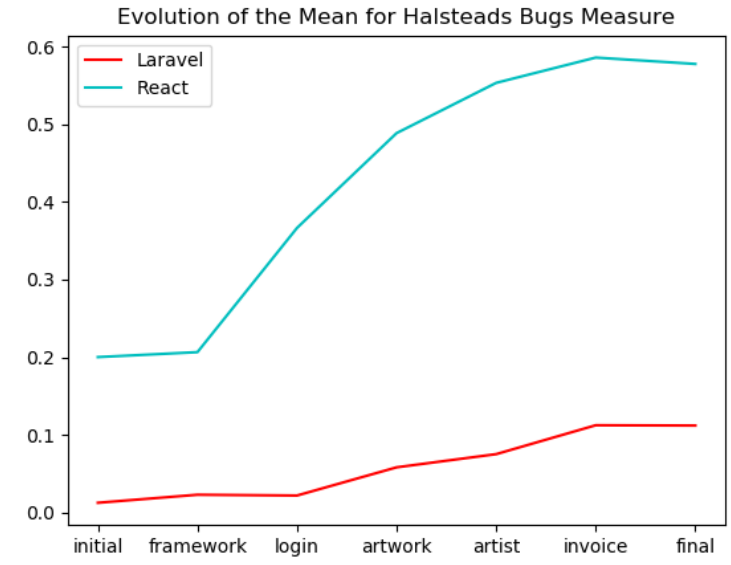


Figure A.2.: Evolution of the arithmetic mean of Halsteads Time and Bugs measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the arithmetic mean after the completion of the sprint shown on the X-axis.

Appendix B.

Evolution of Measures based on the Median

For any measurement it is necessary to know which scale the measurement is operating on. Without knowledge of the scale we do not know which mathematical operations are meaningful on the results of the measurement. For most of the software measurements used during this experiment it is not known if it is justified to assume that they operate on an interval or ratio scale. Therefore, for most measurements used we have to assume that they operate on an ordinal scale, which means that statistical operations such as the calculation of the arithmetic mean are not meaningful. An alternative to the arithmetic mean which works on ordinal scales is the median.

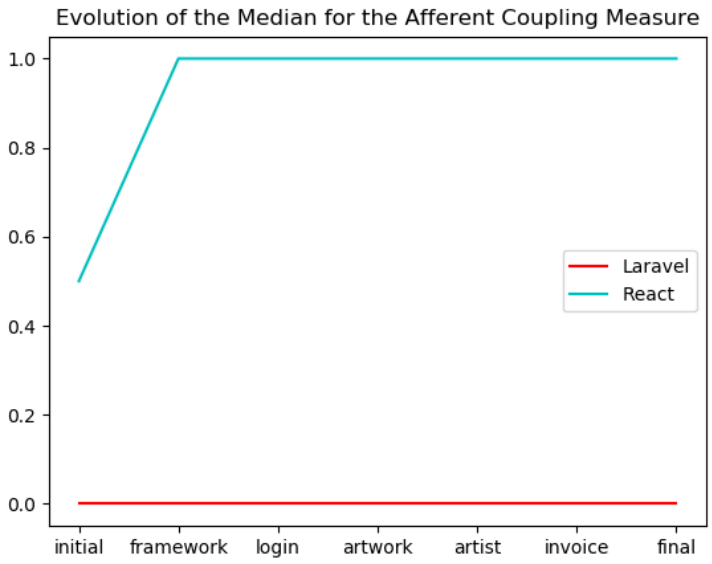
Ordinal scales establish a certain order between the elements measured. There is no additional information about the intervals between each of the elements, we just know the order. Any mapping from an ordinal scale to another ordinal scale which preserves the order of the elements is a valid measurement. Since the median does solely depend on the order of the elements measured, it can be reliably used with ordinal scales.

However, as the results shown below demonstrate the median for the Laravel side often shows almost no change at all for our experiment. Since a main goal of this thesis was to investigate the evolution of measurements, we decided to focus on the evolution of the arithmetic mean instead of the evolution of the median. We would like to point out that even though most evolutions are not particularly useful, the

evolution of the median of the Maintainability Index follows the expectations stated in subsection 5.3.3. Contrary to the results received by investigating the arithmetic mean of the Maintainability Index, the median shows the expected positive impact of ReactJS's general high degree of separation of concerns on the Maintainability Index.

Measurement	Laravel	React
Maintainability Index [18]	42.1	42.945
Cyclomatic Complexity [44]	1	3
Afferent-Coupling [43]	0	1
Efferent-Coupling [43]	2	4
Instability [43]	1	0.75
Halstead Effort [32]	38.055	12532.13
Halstead Volume [32]	40.02	856.073
Halstead Length [32]	12.5	149
Halstead Time [32]	2	696.229
Halstead Bugs [32]	0.01	0.285
Halstead Difficulty [32]	1	13.712
Halstead Vocabulary [32]	8	55
Logical Lines of Code	16	43

Table B.1.: Overview of the results received for the different measurements used. The columns show the different measurements while the respective row for each framework contains the values of the median for each of the measurements. The values shown are the result of the measurements after the `final` sprint.



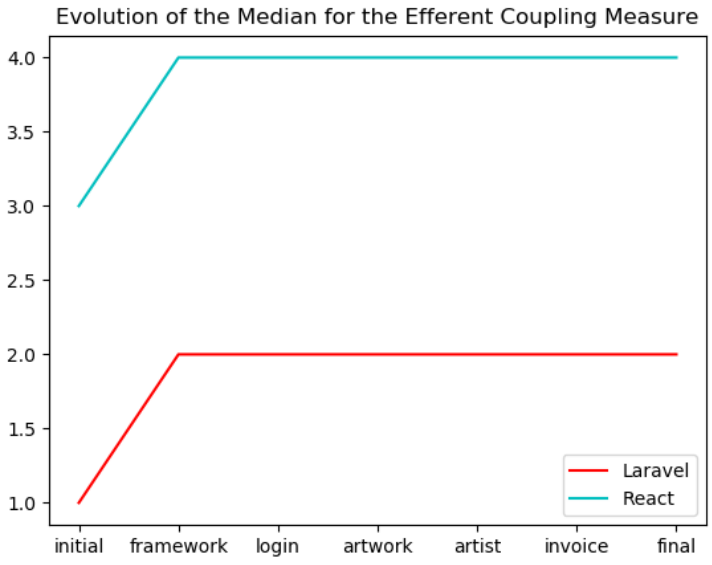


Figure B.1.: Evolution of the median of the Afferent- and Efferent-Coupling measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the median after the completion of the sprint shown on the X-axis.

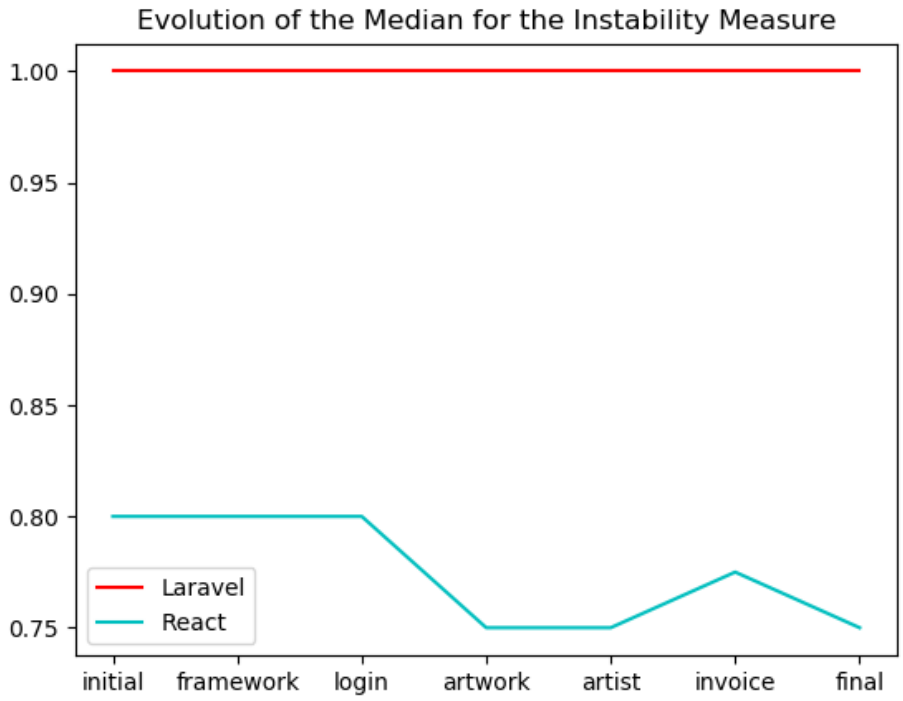
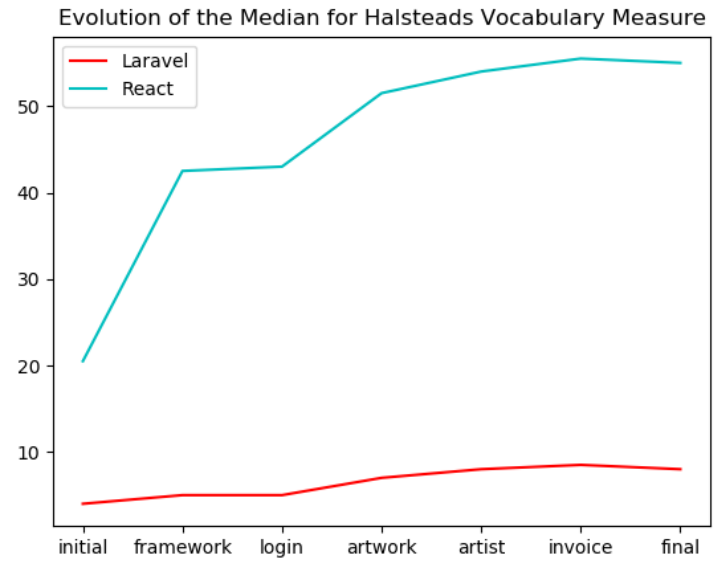


Figure B.2.: Evolution of the median of the Instability measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the median after the completion of the sprint shown on the X-axis.



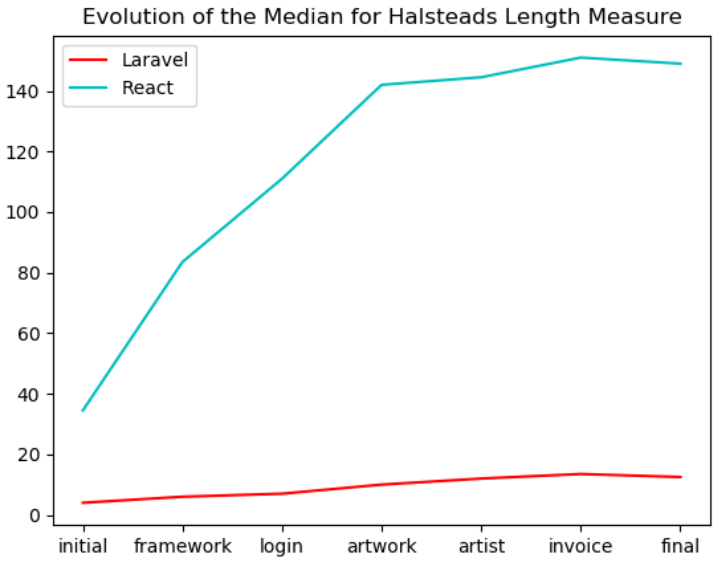
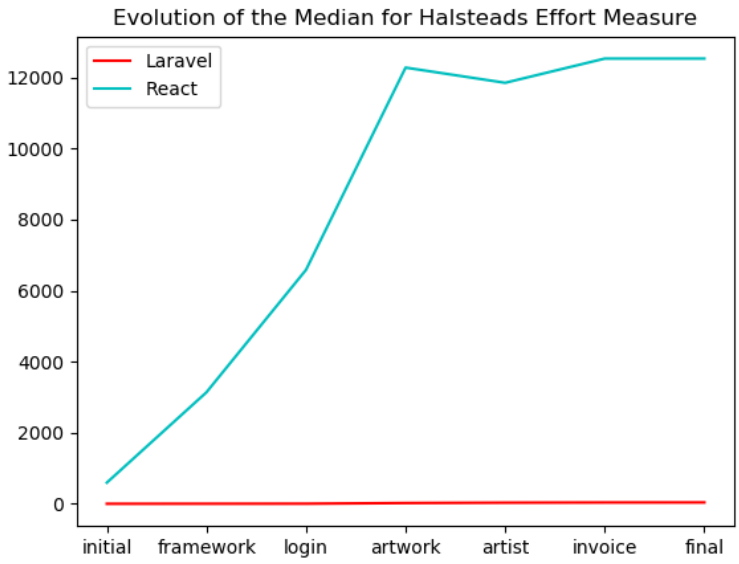


Figure B.3.: Evolution of the median of Halsteads Vocabulary and Length measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the median after the completion of the sprint shown on the X-axis.



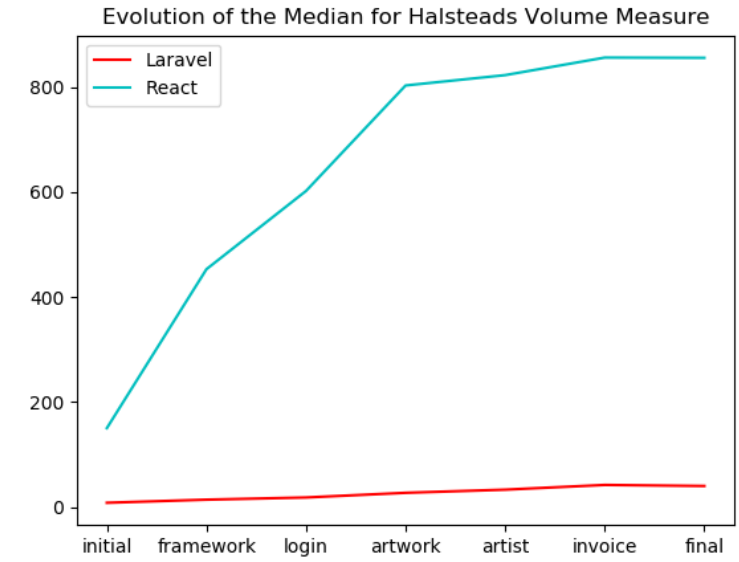


Figure B.4.: Evolution of the median of Halsteads Effort and Volume measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the median after the completion of the sprint shown on the X-axis.

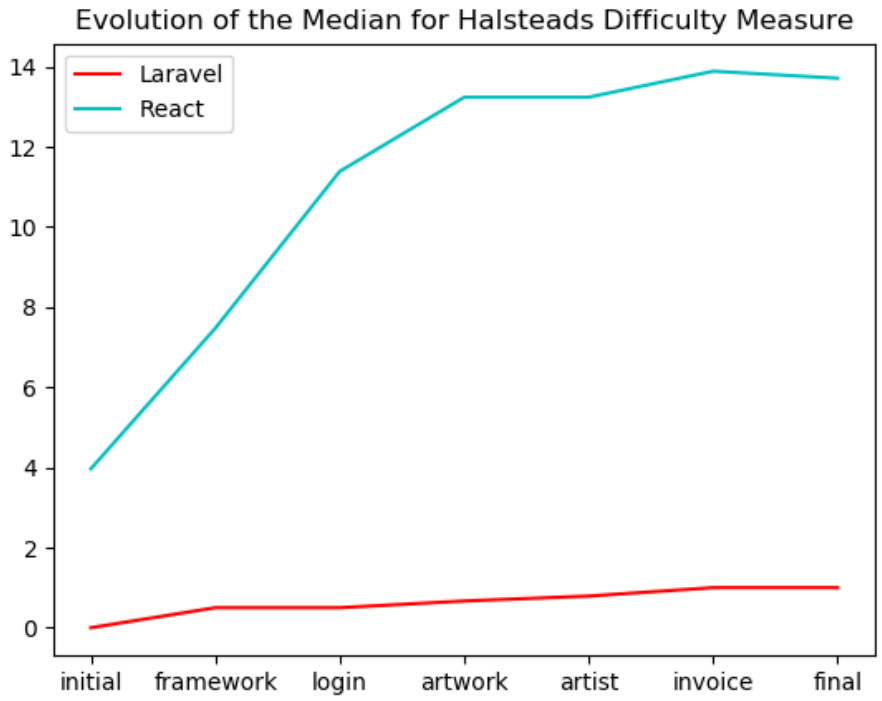
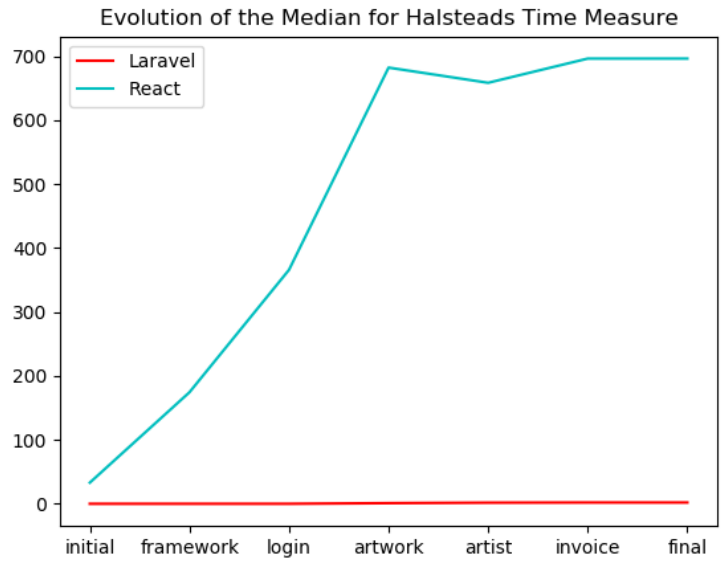


Figure B.5.: Evolution of the median of Halsteads Difficulty measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the median after the completion of the sprint shown on the X-axis.



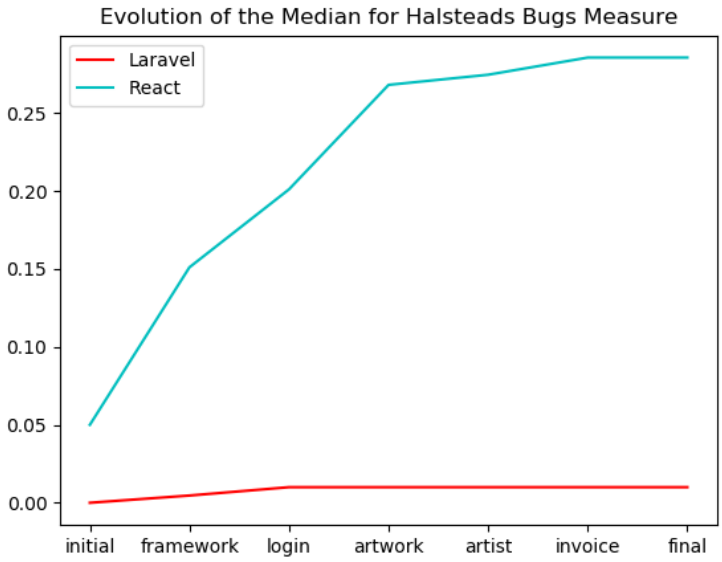


Figure B.6.: Evolution of the median of Halsteads Time and Bugs measure. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the median after the completion of the sprint shown on the X-axis.

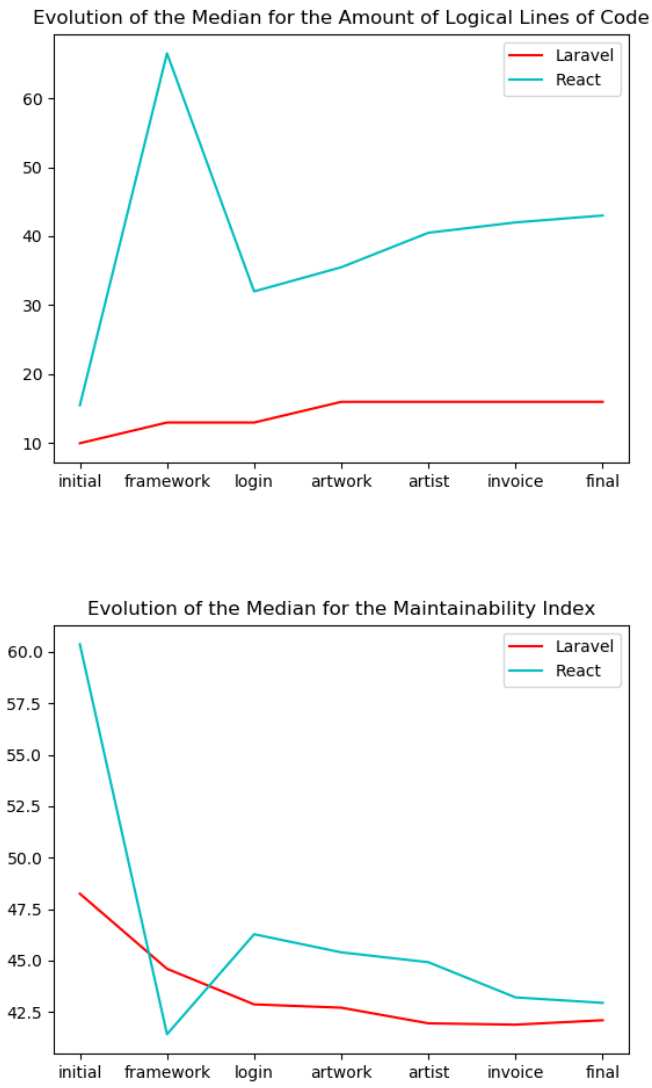


Figure B.7.: Evolution of the median of the logical lines of code and the Maintainability Index. The X-axis shows the different sprints which mark specific points in time over the course of the development. The Y-axis shows the value of the median after the completion of the sprint shown on the X-axis. Contrary to the results received by investigating the arithmetic mean of the Maintainability Index, the median shows the expected positive impact of ReactJS's general high degree of separation of concerns on the Maintainability Index.