Tampere University

Jimi Rantanen

CODE READABILITY IN RELATION TO COMPONENT REUSABILITY

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
May 2023

ABSTRACT

Jimi Rantanen: Code Readability in Relation to Component Reusability
M.Sc. Thesis
Tampere University
Master's Degree Programme in Software Development
May 2023

Component-based development is a model of software development, where software products are assembled from various components. To save costs and time, components are often reused in multiple products, and thus they must be developed to fulfill their role in varying systems. Developing a reusable component requires additional effort, and often, additional code, which can lead to additional complexity. As software development is often a team effort, source code is read by multiple developers, and the readability of code is of great importance. This presents an apparent conflict between component reusability and code readability, as more complex code is intuitively harder to understand. In this thesis, I analyze previous studies in both component reusability and code readability to find consensuses in their defining characteristics, and then draw parallels between both to find if they conflict with one another. Readability appears mostly related to individual features of code such as formatting and naming, whereas component reusability is more related to more abstract characteristics and design patterns. While notions of larger component size leading to more complexity exist in component reusability studies, components should be designed to be as small as possible and to only fill a single, concise role, which lessens complexity. Studies in readability have also shown little correlation between readability and complexity, alluding to reusability and readability being unrelated to each other. Further research points to the two having synergistic benefits: the more understandable a component is, the easier it is to reuse.

Keywords: Component-based development (CBD), source code readability, software component reusability

Contents

1 Introduction

Software is often a group effort, and with good reason: Collaborating with others during development leads to higher quality code with fewer bugs, more learning for everyone involved, and to an increase in both creativity and accountability [Stanic, 2022]. Teams of developers pass source code between each other through version control systems to distribute the workload and to collaborate more efficiently [Zolkifli et al., 2018]. Developers also often use source code made by the community in the form of imported packages, and sometimes build upon them - an example of a popular registry for software packages is *npm*, the Node Package Manager, used in conjunction with Javascript [npm, 2023]. In many cases, a piece of source code is read and used by multiple people, and thus, making both the readability and reusability of source code better are important parts of software development.

Metrics exist to measure and model the readability of source code. Where natural languages have readability metrics, which are based on things such as the averages of sentence length and syllables per word, code readability can be measured with some accuracy using similarly simple metrics, such as average blank lines, average line length, and the average number of identifiers [Buse & Weimer, 2009; Tashtoush et al., 2013]. Natural language readability models see large-scale use, in many places ranging from text editors to official governmental documents [Buse & Weimer, 2009]. Readability is a key thing to consider, whether writing natural language or source code, as both will be read by others. The majority of the time used on software maintenance is spent reading code, and maintenance itself is expected to consume 70 percent of all time spent on a software product [Buse & Weimer, 2009].

When developing an application or some other piece of software, it is wise to reuse components multiple times, to lessen redundancy and avoid copying and pasting the same code snippet to multiple places, which in turn leads to, among other benefits, higher productivity, a shorter development time and a smaller cost of development [Gill, 2003; Ismail et al., 2017; Jha & Mishra, 2019]. Such an approach is called *component-based development*, and according to a global survey, nearly two-thirds of the organizations within the field of software development use a component-based approach when developing software [Jha & Mishra, 2019]. However, it's often hard to predict future needs. Trying to make every component as reusable as possible from the very beginning can lead to complexity through an increase in component size [Jha & Mishra, 2019; Koteska & Velinov, 2013], and if there are no guarantees the component will be reused, such complexity can end up being unnecessary. Complexity is related to readability, even though they don't directly correlate with each other [Buse & Weimer, 2009].

According to the YAGNI principle, short for *You aren't gonna need it*, [Seitz, 2022], you shouldn't build something you only speculate will be useful in the future,

you should rather skip doing it until an actual need arises. As software development is often iterative work [Bittner & Spence, 2006], there will be opportunities to expand upon the original product in the future. Reusability often doesn't come as a by-product, it has to be integrated into software at the design and specification level [Jalender et al., 2012]. Thus, following the YAGNI principle, reusability should only be considered if there is an actual, concrete need to reuse whatever is being built.

I hypothesize readability conflicts with component reusability. Making components reusable is not a trivial task, and it comes with additional code and, in my experience, increased complexity, as possible future needs must be taken into account. Some efforts to make components more and more reusable by making them more and more generic have ended up with bloated and often moderately hard-to-understand components.

This thesis has three main research questions:

- What are the characteristics of source code readability?
- What are the characteristics of software component reusability?
- What is the relationship between source code readability and software component reusability? Are there any conflicts or synergies?

The research will be done as a literature review. I will collect studies about both code readability and component reusability as my material, and synthesize information from them by evaluating and analyzing the findings.

Preliminary research suggests, that no existing studies focus specifically on the relationship between code readability and component reusability. However, multiple studies do briefly touch on the subject, with some stating synergistic benefits, where increased readability and understandability leads to a higher degree of reusability, and some stating how more complex components are harder to understand, thus being less readable [Sharma et al., 2009; Ismail et al. 2017; Jha & Mishra, 2019].

In chapter 2, I go over background information on component-based development and the readability of code. Then, in chapters 3 and 4 I present relevant studies relating to both readability and component reusability. In chapter 5 I analyze and evaluate the material by putting together similar views and contrasting opposing ones. Finally, in chapter 6, I take a look at the analysis and draw conclusions from it, which I then summarize in chapter 7.

2 Code readability & component reusability

2.1 Code readability

Readability is a measurement of how simple something is to understand when read. With code, readability affects many things – for example, maintainability, portability, and reusability of said code. Research has been conducted on how code is maintained, and it appears most of software maintenance time is spent reading code. Some researchers have gone as far as to say a development phase should be entirely dedicated to checking the codebase for readability and making the code more readable, while others emphasize consistent guidelines to make reviewing easier. [Buse & Weimer, 2009]

Maintainability increases as readability increases, by making the code easier to understand. Maintainability can be defined as minimizing the effort it takes to make changes, either fixes or new features. Such changes are proven by research to consume the majority of software development costs, and because making changes to code requires the developer to first read and understand said code, readability affects the maintenance phase of development quite drastically. [Batool et al., 2021]

Research has been conducted on source code readability, and as it is largely a human metric, research has been conducted through surveys. One approach is to provide code snippets that survey participants have to then grade on a readability scale [Buse & Weimer, 2009] and another is to just list source code features, from structures such as loops and if-else -statements to other purely aesthetic and arbitrary parts, such as identifier names, indents, and spacing [Tashtoush et al., 2013]. In both cases, statistical techniques are used to analyze the results and determine which features and constructs result in better readability. Findings include meaningful identifier names and commented code as positives in regard to readability, while complex arithmetics and nested loops affect readability negatively [Batool et al., 2021]. Consistency also arose as a positive indicator of readability, and consistency in source code can be enforced automatically through sets of styling rules. Such processes are often called “linting”.

2.2 Linting and style guidelines

Linting refers to the process of automatically checking source code for errors, both stylistic and also programmatic, through the use of a preset guideline on how the source code should look like, and what kinds of guidelines are to be enforced.

The term itself, ”lint”, can be traced back to a UNIX utility by the same name developed in 1978. The tool was made to examine source code written in the programming language C and detect various constructs and patterns which shouldn’t be used, even if the compiler allows them. These include anything deemed wasteful or error-prone, in addition to being more strict on C’s type rules and also filtering out straight-up bugs in the code. The rationale behind the tool was to separate such careful and thor-

ough checking into another tool and to let the compiler itself be as fast and efficient as possible. [Johnson, 1978]

Nowadays linting fulfills much the same role as the namesake UNIX tool, but multiple different linters for all kinds of programming languages have been developed, which in turn have multiple possible rulesets the developer can choose from and tweak to their liking. For example, a popular ruleset for Javascript and Typescript is Airbnb's style guide [Airbnb, 2023].

Due to the big customizability factor, modern linting tools can detect a more wide variety of constructs and patterns than the original tool does. The detection rules vary not only by developer choice, as the available rulesets can be very different for different programming languages. Compiled and interpreted languages for example have a different take on linting, and as the latter has no compile phase to detect errors, the importance of linting as a way of catching errors before runtime is amplified. In addition to runtime errors, linting often enforces a consistent style in writing code, which in turn makes said code easier to maintain and review. Lastly, linters can also be set up to catch performance-reducing constructs, which leads to more performant code in the long run. In many ways, a linter can be compared to a spelling checker in a word processor. [Ogut, 2021]

2.3 Component-based development

Component-based development (CBD), also known as component-based software development (CBSD) or component-based software engineering (CBSE) is one of many models for software development processes, among others such as Waterfall and Object-Oriented Development (OOD). Processes can be split into two categories; non-technically specified ones, such as Waterfall, and ones that are more closely tied to technology, such as OOD. Non-technically specified processes can be further split into sequential and iterative models, but CBD falls into the more technical realm. Being more technically focused, CBD has more influence on the development and maintenance of applications built using it. [Crnkovic et al., 2006]

At its core, the Component-based approach is about using and reusing existing software components. In case something entirely new is needed, it should be implemented as a new component, but the primary focus is reuse. The Component-based development process can be split into two levels – the system-level development process, and the component-level development process, both of which have differing activities. The system-level process is about developing a system from reusable components, while the component-level process is about developing them, with an emphasis on reusability. To tie components together, implementation using a Component-based approach includes glue code to connect the components, and it is considered a part of the system-level process. Between the two development process levels exist bridging activities, which can be summed up as Component assessment –

the acts of finding, selecting, and verifying components to be used as parts of the system. [Crnkovic et al., 2006]

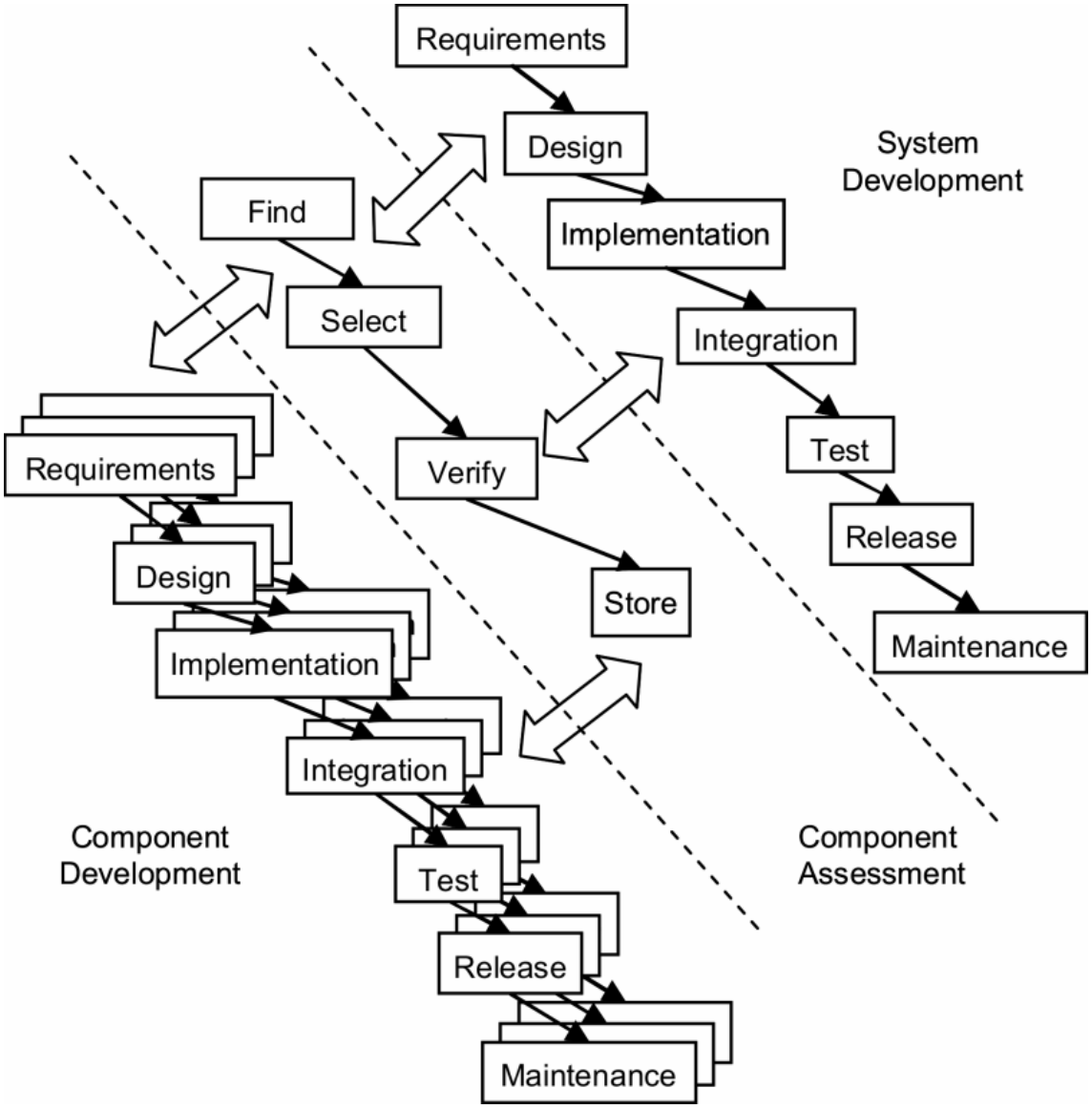

Figure 1: Processes of component development and system development with bridging activities [Crncovic et al., 2006].

2.4 Development processes in component-based development

In Figure 1, the processes for both component and system development are shown, along with their bridging activities. While the phases follow the same formula for both processes, the phases themselves differ in each process.

The requirements phase in system development is influenced by decisions made when developing components. While components are also developed in parallel, the system requirements take into account existing components. However, the requirements phase for the component development process is also influenced by the requirements of the system-level process – the requirements for components to be developed are a result of system-level specifications. In addition to the requirements at the system level, the

development of components should also look forward into the future, which is why reusability should be considered when drafting requirements for components. Reusability can be achieved through generality, which should be an explicit consideration in component requirements. Generality in software development can be simplified as not being limited to a single use case, and in the case of software components, accepting parameters through which the function of the component can vary [Návrat & Filkorn, 2005].

The analysis & design phase of system-level development is about designing the system architecture and details while keeping in mind what kinds of components already exist. A balance is to be struck between the exact desired design and compromises which have to be made if existing components are used. In the component level process, components must ultimately be designed to be part of a system, even though the said system might not even exist yet in any form. Assumptions must be made about the system in which the component might be used when designing a component. In addition, reusability also comes into play during the design phase of components. Reusability is attained through generality and adaptability, which also makes the component larger and more complex – still, the component should also be simple and efficient to fulfill specific requirements, which makes reusable component development more costly effort- and resource-wise than non-reusable single-purpose component development. [Crnkovic et al., 2006]

In an optimal scenario, the system-level implementation phase consists of little coding. System-level implementation should mostly focus on integrating components into the system, by testing, specification verification, adaptation and possibly glue-coding components together. On the component process side, implementation depends quite a bit on the technologies selected. Concepts such as interfaces are present in many object-oriented programming (OOP) languages, and many of the OOP features have synergies with component-based development. [Crnkovic et al., 2006]

The system-level integration phase is partly spread into earlier phases, as the requirements of the system are not often fulfilled by one component alone, but rather by multiple components working together – for this reason, integration of the components must be taken into account during requirements, design and implementation phases. The components themselves must be built to be integrated into systems, so at the component-level process integration must also be kept in mind all the time. [Crnkovic et al., 2006]

Testing is different in both processes. On the system level, there might not be control over the component, if it is something developed previously and/or by others. Also, testing the components, while crucial, is not enough – components can and will behave differently depending on the system. Tests are thus performed in many different ways: for the components themselves, when components are integrated into other components, and when the whole system is deployed. And as components are to be used and reused as parts of multiple different systems, testing in the component-level process is vital.

The components can be used in a multitude of different and likely unforeseen ways, so nothing should be assumed as constant from the conditions where the component is used. [Crnkovic et al., 2006]

The release of both systems and components is quite simple. In both processes, the product (either the full system or a component) is packaged for delivery. Maintenance differs - on the system level it's mostly about replacing components. By intuition, there shouldn't be a need for component replacement, as software is not subject to wear in the same way hardware is, but maintenance through component replacement is still necessary; the components themselves won't change, but everything around the software will see changes over time. Maintaining the components themselves is more problematic – components are developed and delivered to multiple systems, and it is often the system developers who encounter any possible issues. Contracts and strategies must be made to ascertain which party is responsible for updating the component, and if there is a promise of future updates for the component. Furthermore, in a multi-component system, it can be difficult to find the root cause of the issue. Something might appear to behave erratically in a component, but the real culprit might be another component. [Crnkovic et al., 2006]

3 Characteristics of code readability

In this chapter, I go over two surveys conducted on human annotators, and an acclaimed book discussing readability as the *cleanliness* of code, with a 4.7/5 average rating among over 5000 reviewers on Amazon at the time of writing. All three have significant parallels, and, to an extent, a consensus on which factors contribute to the readability of code, and conversely, which factors can be seen as negatives regarding readability.

3.1 Studying readability via a snippet-based survey

In their study, Buse & Weimer [2009] approached measuring readability by giving 120 human annotators 100 different snippets of Java code, with an average length of 7.7 lines – an example is shown in Figure 2. The annotators then had to grade the snippets on a scale of one to five, where a larger number indicated the snippet of code was more readable, and a smaller number meant it was less readable. The authors note, that their study correlates strongly with software quality metrics, such as defect density, and that readability strongly determines the quality of code. In addition, they compared their readability metrics to complexity metrics and found them to not correlate for the most part.

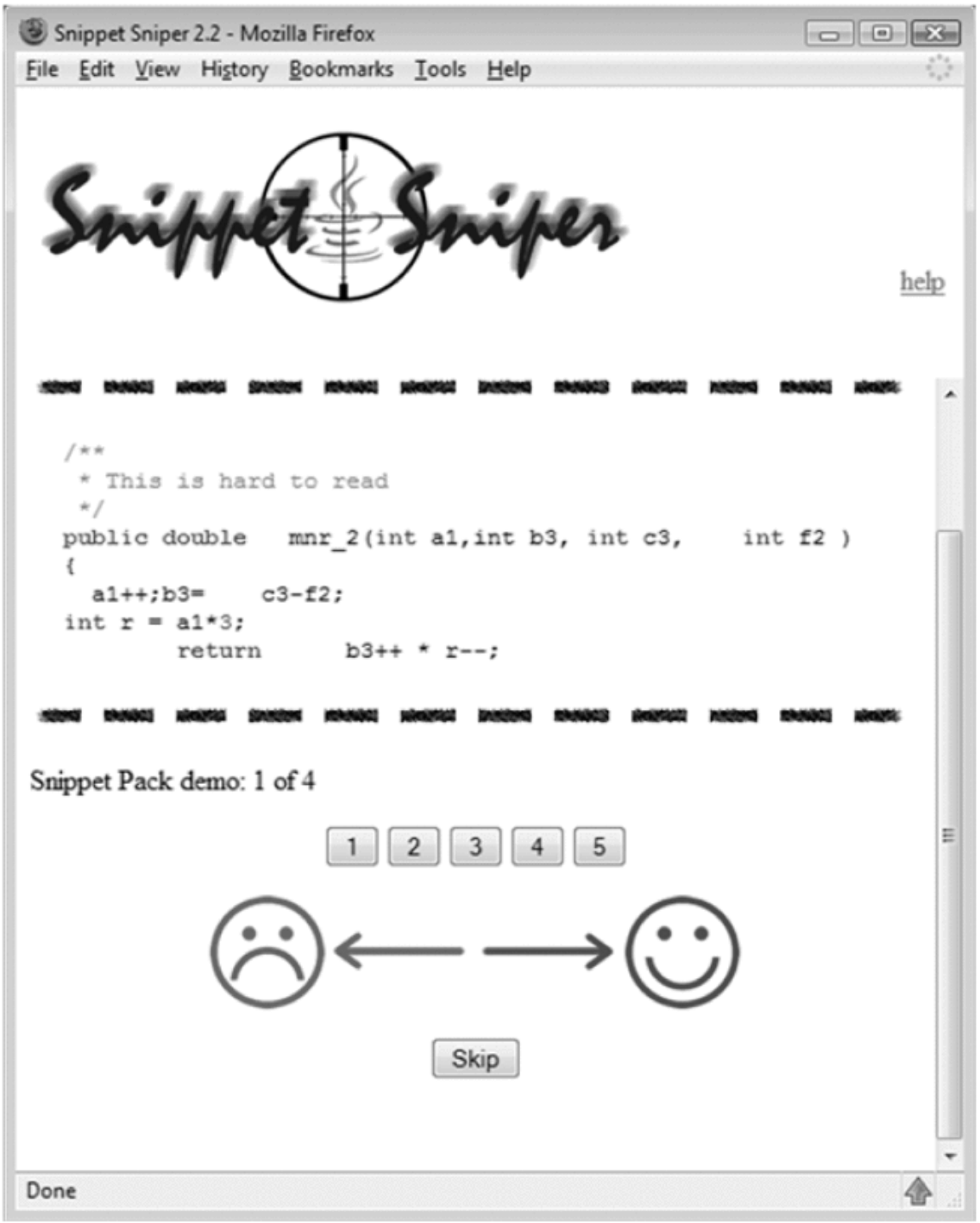

Figure 2: An example of a snippet, accompanied by the rating tool [Buse & Weimer, 2009].

The snippet selection process was not at all random – many considerations were made when choosing which snippets to show. Length was one of the most important factors - a snippet should be as short as possible, but not too short, so readability can be judged. A snippet should be logically coherent, include any comments related to it, and not include any boilerplate code, such as import statements. Such short snippets won't include any context, such as examples of how any given method is to be used. This was

on purpose – not that context is irrelevant, but the authors wanted to focus more on individual details of the code.

Due to the above considerations, a snippet is defined as such: it specifically consists of three simple statements accompanied by any possible comments and function headers, or compound statements, such as if-else, try-catch, switch, and for. A simple statement can be a field declaration, an assignment, a function call, or a keyword such as break, continue, throw, or return. In addition, a snippet won't span multiple methods or in any other way cross scope boundaries.

The results show many snippets were judged similarly by many annotators, and overall scores were weighted slightly towards more readable rather than less readable. Due to not being given specific instructions on how to grade snippets, relative scoring is considered more than absolute scoring. i.e., if a snippet was rated more highly on the readability scale by multiple annotators than another snippet, it is considered more readable, regardless of the absolute scoring.

Using the results, the authors generated a readability model with machine learning algorithms. The individual analyzed features were chosen based on the intuition, that they would likely have an effect on readability. The features could then be statically detected from the snippets. Machine learning algorithms, as opposed to simple correlation methods, were necessary, as readability judgments are suggested to stem from an interaction of features, not necessarily from individual features themselves. The algorithm received a set of features extracted from the snippets as instances, along with the snippets partitioned into two categories: less readable and more readable, with the cutoff point being at a score of 3.14. After training the model, when given instances it hasn't seen, it would give an estimate of readability, based on how likely the instance is to belong in either the less readable or the more readable class.

Testing the performance of the model, the authors experimented with different kinds of classifiers. The best of the classifiers correctly classified nearly four-fifths of the snippets. In addition, the output of the classifiers correlated rather strongly with the human scores. The authors state the metric is "in a very real sense [...] just as good as a human" [Buse & Weimer, 2009, page 551].

After establishing the metric, the authors split the original human annotators into groups based on their experience and repeated the experiment in each group. They found their metric to be closer to the human average than any of the individual groups, with a higher agreement in readability judgments in those with more experience.

Finally, the features were analyzed one-by-one, and relative predictive powers for different features were found, as shown in Figure 3.

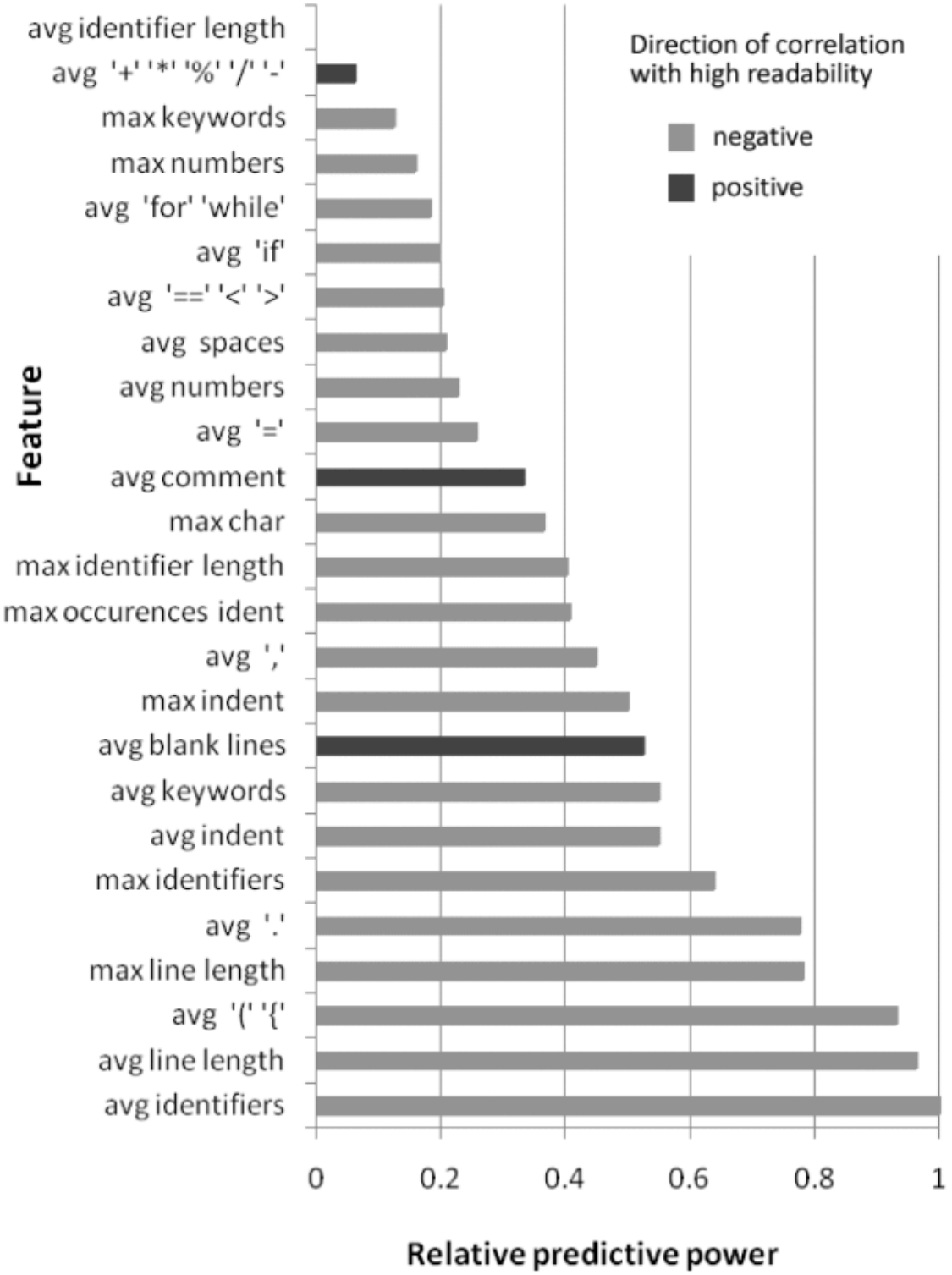

Figure 3: Chosen features and their generated predicting power [Buse & Weimer, 2009].

As such, average line length, the average number of identifiers per line, and the average number of brackets are the most descriptive features when considering readability, according to this study, while average identifier length and the average amount of arithmetic operators have little to do with readability. The authors draw a line to natural

languages, where long sentences are harder to understand than shorter ones, and the same holds in source code – shorter lines seem to be easier to understand.

After extensively going through their readability metric, the authors compare readability with cyclomatic complexity, to test their claim readability is independent of complexity. More precisely, the authors claim code readability is not related to the problem the code is trying to solve, no matter how complex it is, but rather a quality of the code itself. The selection of short snippets and the studying of code shorter, surface-level code features is partly due to this hypothesis. Multiple benchmarks exist for cyclomatic complexity, and the authors found at most a weak correlation between readability and complexity, illustrated in Figure 4. To further prove their hypothesis, the authors computed the correlation between method length and complexity, which is much higher.

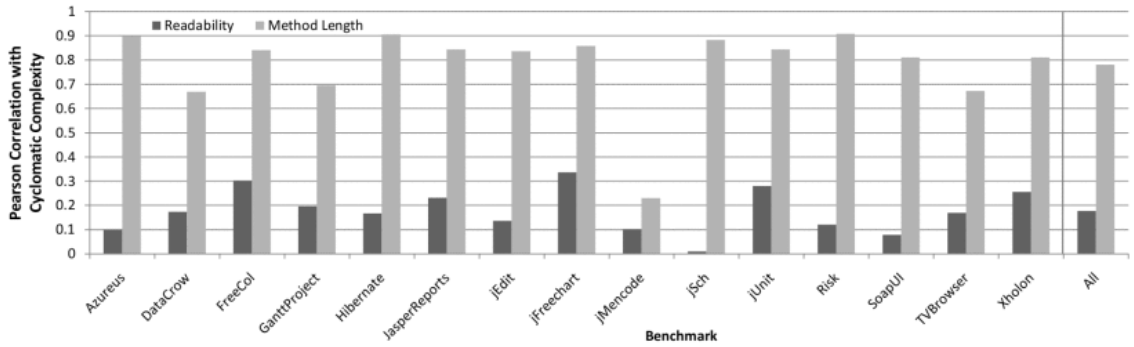

Figure 4: Correlation between readability and cyclomatic complexity, and method length and complexity.

Finally, the authors go through some additional notes. For one, they emphasize average identifier name length had next to no predictive power on readability in their metric – a finding which directly contradicts a common, intuitive notion, which they quoted as ”single-character identifiers … [make the] … maintenance task much harder” from a study by Knight & Myers [1991, page 33]. Moreover, the maximum length of identifier names could be used to predict code as less readable to an extent. The authors concede their metric cannot detect whether an identifier name contains useful information or not, but longer identifier names without such information seem to predict readability negatively.

Comments are another feature, which intuitively increases readability. However, in this metric, comments didn’t correlate with readability too well. The authors speculate this might have to do with how comments are used – less readable code might be more likely to be commented, and the comments merely cancel out features considered less readable.

3.2 Studying readability via a feature-based survey

Like Buse & Weimer [2009], Tashtoush and others [2013] begin by comparing software readability metrics to similar metrics for natural languages, citing examples such as Flesch Reading Ease and the Flesch-Kincaid [Kincaid et al., 1975], which derive their readability judgments from several individual factors, including word and sentence length. Shorter sentences and words are said to be easier to read.

The authors go on to cite multiple research papers including the one by Buse & Weimer [Buse & Weimer, 2009], to go through different metrics for readability. First, they go through research by Abbas [Abbas, 2010], where the Flesch Reading Ease test (FRES) is applied to source code. FRES calculates the ease of reading from word length and sentence length – when applying this formula to source code, keywords, and identifiers take the place of words, and sentences become statements.

A 1997 research by Chung Yung is also cited [Yung, 1997], which correlates software readability with maintainability. According to Yung, four metrics for source code readability exist: The number of unique operators, the total number of operators, the number of unique operands, and the total number of operands.

”Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability” by Wang and others [2011] is, as the title would suggest, stated to explore the automatic segmentation of code, based on various sources which state segmentation improves the readability of code. The research found such segmentation can be done automatically, and it makes for as readable code as segmentation done by a human. Finally, multiple research papers are also cited to discuss the importance of meaningful identifier names.

The authors call their approach the IPFCR approach – Impact of Programming Features on Code Readability. As such, they chose 22 different features of source code, which based on their sources relate to readability. Like Buse & Weimer [2009], Tashtoush and others [2013] rely on human annotators to get readability judgments. The annotators are chosen at random from multiple different software development companies, and their answers are given different weights based on the experience of the annotator by using an ANOVA test and attaining an F value for each feature. Figure 5 illustrates the results.

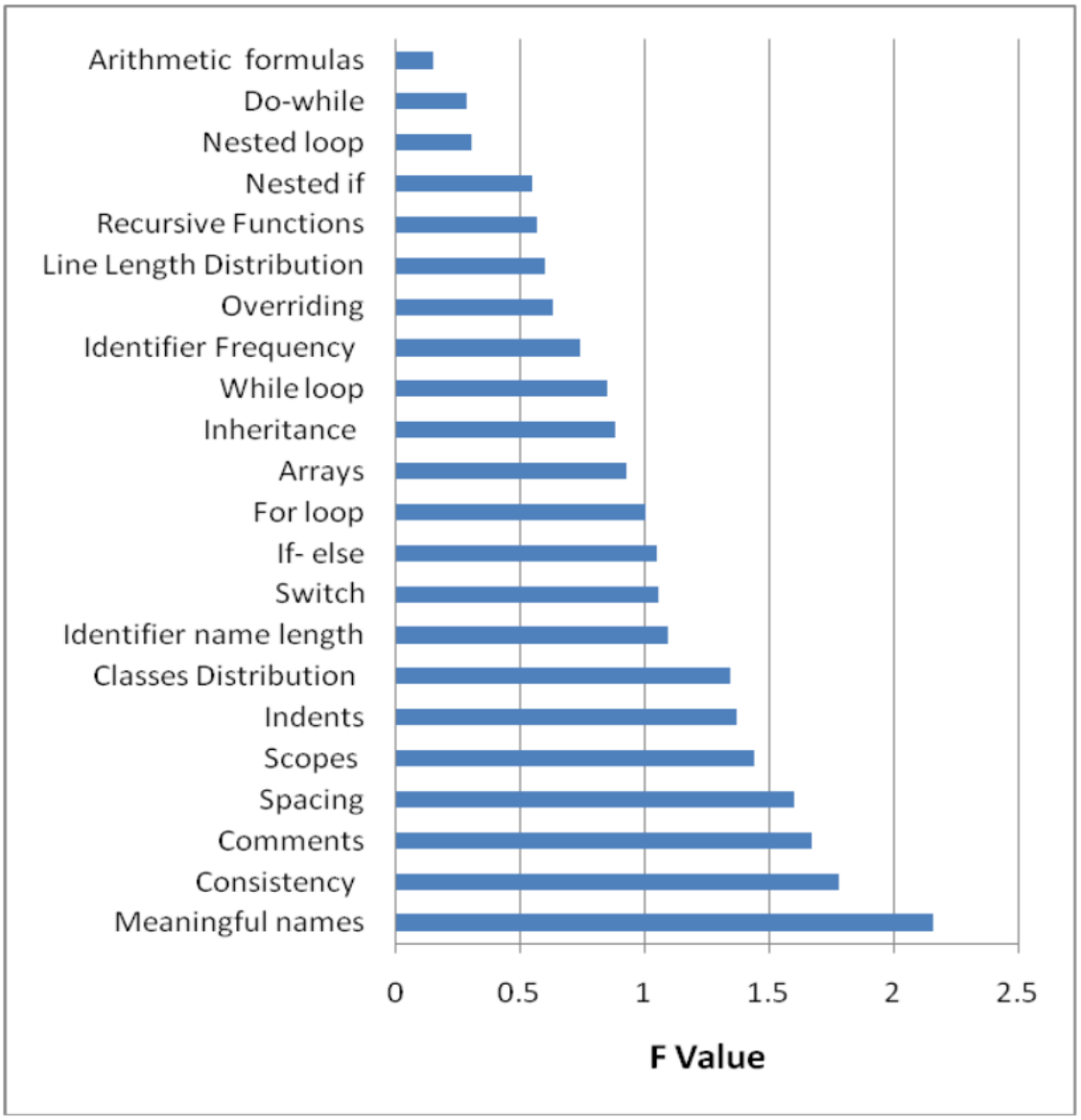

Figure 5: Chosen features and their impact on readability, weighted by job experience. Meaningful names, consistency, code lines, and comments affect readability positively, while arithmetic formulas, nested loops, and recursive functions do the opposite. [Tashtoush et al., 2013]

To clarify the most descriptive feature, the authors give some insight into the ”Meaningful names ” feature. ”Name” here refers to anything which can be named, for example, variables or classes. They state ”meaningful” in this context is partially subjective, and annotators may have conflicting opinions on what counts as meaningful. Some simple universal metrics on what is meaningful can be derived, e.g. two-letter names are rarely meaningful, but further evaluation of meaningful can prove difficult. Natural language parsers could probably be used, but even then some combinations of abbreviations might prove difficult to spot, even if such names are intuitively deemed meaningful by human annotators.

3.3 A handbook of readable code

In his book *Clean Code*, Martin [2009] discusses the readability and overall cleanliness of code in great length. An entire chapter is dedicated to meaningful names, with mentions of meaningful names also appearing in other chapters. Comments are also discussed in length, and proper formatting is deemed important. I'll go over the most relevant parts of the very detailed book.

Meaningful Names

Many parts of source code need naming – variables, functions, classes, packages, and so on. Names are a major contributing factor to readability as a whole, as naming things is so prevalent. While ”meaningful” can be a very subjective term, Martin provides some measures for the meaningfulness of names.

First and foremost, a name should communicate intention. A name should describe three things about what is being named: the reason for its existence, the purpose of it, and the way it's used. For example, a variable keeping track of a measure of time could be named *elapsedTimeInDays* or *daysSinceModification* – both communicate reason, purpose, and usage. In addition to choosing good names, it's also important to know when and if something should be named. Source code can contain unnamed variables, which in some cases could work better when named.

Many words in programming have established meanings. For example, a list often refers to a specific structure, and calling something a list should take this into account. Martin calls being mindful of cases such as this *avoiding misinformation*. Other ways misinformation can be avoided include avoiding similar-but-different names, where the difference can be hard to spot. This includes both avoiding long multi-word names which appear very similar to one another and avoiding combing similar letters such as lower-case *L* and upper-case *i*.

Decisions concerning naming should be made with readability and understandability in mind. Sometimes two things can share a name, and a distinction needs to be made – the distinction should also be meaningful, and not just an arbitrary addition to the name. For example, a reader should be able to look at two similarly named functions and know how they differ and why both exist.

Names should also be pronounceable and searchable. Both are due to human factors. Named parts of code often need to be discussed with others and others should be able to easily find the part being talked about, and the name being both pronounceable and searchable helps immensely with such interactions.

Encoding, in the context of naming, means using specific letters or patterns to signify meanings that aren't immediately apparent from the letters or patterns used. For example, a concrete class could be named *ShapeFactory* and an interface class could be named *IShapeFactory*, with the *I* being encoding for *interface*. Martin advises against encoding, as it is an extra layer of complexity, and often a needless one at that. Related

to encoding, readers shouldn't need to mentally map names to actual concepts, which is why short, single-letter names should be avoided.

When deciding on a name, what is being named must be taken into account. Class and object names should be nouns, and method names should be verbs. The nouns and verbs used should be the ones that most clearly and plainly describe what is being named, not complex synonyms or vague references. Also, multiple different nouns or verbs used for the same concept shouldn't be used in tandem, a single name should describe a concept, with distinct parts added to the name if need be.

Context should also be considered when naming. A variable called *state* could mean multiple things, but when used in the context of an address, such a name can be understood. If the context is not obvious, it must be made more obvious by making a class or an object containing variables, or using a common prefix for a group of variables. Prefixes can be overdone though, which will make the context less clear.

Finally, a name shouldn't require a comment to further describe it, the name should act as the description.

Comments

Martin approaches comments in a very two-fold way. While good comments can indeed be good for readability, bad comments can be very harmful. As stated above Martin advocates for using the code itself to describe the code, and place less emphasis on comments. He argues comments can't and won't be kept completely up to date, and over time the comments can become sources of misinformation, which is far worse than no comments at all.

Good comments come in a few varieties; Necessary comments such as legal information containing copyright statements, explanations, clarifications, warnings, and comments describing functionality not yet implemented. All of these contain information that can't be conveyed through only code: Sometimes the intent of a decision requires explanation, sometimes a part has an identified fault or a pitfall of which the reader should be aware, and sometimes the code just requires additional clarification or emphasis despite best efforts to convey meaning through the code itself.

Bad comments also exist in multiple forms, and according to Martin, in more numerous forms than good comments. Most of the described bad comments are varying appearances of simply useless or redundant or even misleading comments. If a comment doesn't add anything to what it's commenting, it's redundant.

Formatting

While formatting is often done automatically [Ogut, 2021; Airbnb, 2023], its effect on readability shouldn't be understated. Martin splits formatting into two categories: Vertical and horizontal formatting. Each has its own set of clarity rules to follow, and

Martin goes over concepts called openness, density, and distance for both, with distinct definitions.

Where vertical openness refers to using blank lines to group related lines of code together, vertical density has to do with minimizing the number of lines within a group. Both work together to split code into chunks, and vertical distance means bringing related lines to their related groups. For example, variables should be declared as close to their usage as possible, and functions which call other functions should be close to one another, with the called function being below the calling function. Groups can be formed around concepts as well – Martin [2009] explains a concept called *conceptual affinity*, where a stronger affinity should correlate with less space between lines. Affinity can come from more direct factors, such as the aforementioned function which calls another function, but there are many other sources of affinity as well. For example, Martin [2009] provides a group of four functions, which perform similar operations: *assertTrue(String message, boolean condition), assertTrue(boolean condition), assertFalse(String message, boolean condition),* and *assertFalse(boolean condition).* The conceptual affinity is strong in this case because all four functions are named similarly and perform a similar task of asserting the value of a condition, with or without a message.

Horizontally, openness and density also refer to separating unrelated parts of code and grouping related parts, but as it's done within a line, it is a bit different from vertical formatting. Spaces can be added between operators and operands to distinguish them from one another, but spaces shouldn't be used between a function name and its opening parenthesis, as they are closely related. Additionally, indentation is a crucial part of horizontal formatting, and it makes the structure and hierarchy of code visible.

Finally, Martin emphasizes consistency in formatting. A team of developers should agree upon a style, and then use tools such as linting [Ogut, 2021] to enforce the style to keep everything in line.

Classes

Martin has a whole chapter dedicated to guidelines on how to write classes. It mostly has to do with formatting and some Java-specific rules, but most interestingly, he states classes should be as small as possible. And in contrast to functions, the size of which can be counted by lines, Martin counts the size of classes by their responsibilities. He cites the Single Responsibility Principle, which defines a responsibility as a *reason to change,* and classes should only have a single one. The Single Responsibility Principle consolidates notions of *coupling, cohesion, the separation of concerns,* and *structured programming and design* into a single principle, and it's one of Martin's own works [Martin, 2014].

The Single Responsibility Principle is stated to be very important in object-oriented software design, but at the same time, it often gets overlooked. Martin attributes this to

developers not organizing code once it works, but also to developers feeling like separating a piece of software into numerous small, single-purpose classes makes it harder to understand. Martin disagrees with this notion and instead argues the amount of complexity in a project stays the same whether smaller or larger classes are used. Organizing software into small, single-purpose classes makes it easier to focus on only relevant complexity, and not have to look at unrelated parts of the software.

Adhering to the Single Responsibility Principle can be done by tracking the *cohesion* of a class. In cohesive classes, most functions and variables are related to each other directly. If clear subsets of functions and variables form within a class, it's a sign the class should probably be split into multiple different classes.

Introducing new features to a class requires modifying the class, which is error-prone. Martin instead suggests the Single Responsibility Principle should be kept in mind, and when new functionality is to be added, it should be added as a separate class instead. Another object-oriented principle is stated, the *Open-Closed Principle,* which states classes should be closed for modification while being open for extension. Extending a class through subclassing keeps it closed for modifications. The Open-Closed Principle is another one of Martin's own works [Martin, 1996].

Lastly, classes should interact with one another through interfaces. If a class depends on the concrete implementation of another class, the dependent class is in a volatile state, as the class it depends on might change. If there is an interface in between, it represents the dependency that the classes must fulfill.

4 Characteristics of component reusability

In contrast to code readability, approaches to describing component reusability differ more wildly. In this chapter, I go over eight separate studies with varying views, ranging from more vague descriptions of component reusability characteristics, to more specific guidelines on how to improve component reusability.

4.1 Assessing component reusability

Sharma and others [2009] have identified four metrics for software component reusability in their study: Customizability, interface complexity, portability, and understandability.

Customizability

Customizability helps with fitting a single component into multiple different systems, which leads to better reusability. A measurement of customizability can be made by comparing the number of writable properties to the total number of properties in the component. In Java, for example, component properties are often written to using an encapsulation method known as a *setter*, and thus the number of writable properties can be derived by counting the number of setters. Dividing the number of setter methods with the total number of properties yields a value between zero and one, where a lower value indicates less customizability and a higher one implies the component is more customizable.

Interface complexity

Applications interact with components through interfaces. The interface of a component is the primary point of contact when trying to understand and ultimately use said component. If the interface is complex, understanding the component is harder, and it's also harder to use and reuse. Less complex interfaces lead to better reusability overall.

Portability

Ideally, reusable components should be functional in multiple environments and platforms with little effort. Portability is the measurement of effort put into modifications of the component and the new environment or platform to make the component function.

Understandability

In this study, the authors only approach understandability through documentation, and not the code itself. Good documentation makes using a component less time-consuming and more efficient.

[Sharma et al., 2009]

4.2 Designing reusable components

In their study, Designing Code Level Reusable Software Components, Jalender and others [2012] discuss reuse beyond component-based development. They list four dif-

ferent levels of reuse in software development: Design level products, analysis-level products, entire applications, and finally, code-level components, which consist of modules, libraries, and the like. However, of these four levels, the authors state code-level reuse as the most commonly used one.

The lack of formal specifications is stated as a major hurdle when it comes to reusable component development. Natural language specifications are described as insufficient – specifications should be mathematically rigorous, complete, unambiguous, and understandable. Aside from specifications, the component itself should have clear boundaries, clear functionality, and clear interfaces with which to integrate with other components. While most reuse is only concerned with interfaces, some applications of reuse might need to delve further into the component, and for those an unambiguous specification of the component is essential. And finally, for both cases, documentation with information about possible reuse with examples is needed.

The authors list four ways to build reusable components at the code level: Class libraries, Function libraries, Design patterns, and Framework classes.

Class libraries

The reusability benefits of class libraries are mostly attributed to core concepts of classes: Inheritance, polymorphism, and dynamic binding. In addition, when compared to function libraries, classes are stated to provide better abstraction and adaptability. However, an issue with class libraries is overlap: multiple different libraries can have classes for concepts such as generic data structures, which makes using multiple libraries in tandem difficult.

Function libraries

Function libraries are stated to be the most common way of reusing code components. A common example of a function library is the C programming language standard library, which has functions for e.g. input/output handling and mathematical functions.

Design patterns

Design patterns are a slightly more high-level concept when compared to libraries. A design pattern has to do with how code is structured, and using an established design pattern can simplify maintenance and avoid architectural drift. Using an established pattern also improves communication between developers, and allows less experienced developers to have a widely accepted basis to build off of. Examples of design patterns include Model/View/Controller (MVC) and Client/Server. In MVC specifically, the benefits arise from the separation of concerns between the input, processing, and output of an application.

Framework classes

Frameworks are collections of classes, which are designed to work together as a complete set, as opposed to components of class libraries which can serve as individual,

stand-alone parts. Reusing a framework class allows the developer to build upon a ready-made overall design of an application. The main advantage of framework classes over library classes is stated to be the communication between components. The components of a framework class have built-in support for intercommunication between its components, while a class library often does not.

4.3 Genericity and reusability

Bigot & Pérez studied reusability through genericity and identified three concerns which software components bundle up: A component's behavior, it's data-types, and its targeted execution resources. According to the authors, breaking up this bundle of concerns would greatly increase reusability. The authors approach their study by presenting an example of a component, to which they then apply measures to increase its genericity, and evaluate how well the more generic component can be used to implement the initial example.

The example used was a *task farm skeleton*. A task farm is a type of algorithmic skeleton for parallel computing, with three distinct roles illustrated in Figure 6: A *dispatcher*, any number of *workers,* and a *collector.* Algorithmic skeletons are a part of parallel programming, where many different algorithms can be identified to derive from a shared pattern [Cole, 2004]. Other such patterns, in addition to the task farm, include *divide & conquer, pipeline,* and *map and reduce* [Bigot & Pérez, 2009].

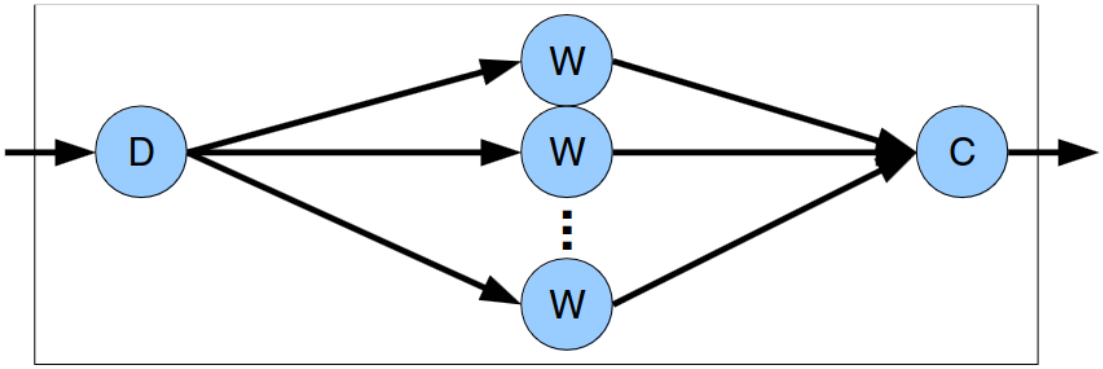

Figure 6: The 'task farm' algorithmic skeleton. D stands for dispatcher, W stands for worker and C stands for collector. [Bigot & Pérez, 2009]

It's stated a component-based implementation of a task farm will often contain three components, one for each role. For increased reusability through genericity, the *composite*, the combined component of all three sub-components, should allow for multiple aspects to be passed in as parameters: The type of the data, the number of workers, and the implementations of all three roles, with default values existing for the dispatcher and the collector.

4.4 Determining the characteristics reusable components

Ismail and others [2017] begin by stating that component-based development decreases the cost and time of software development and improves the quality of software, by reusing existing components and thus lessening the need to write everything from scratch. They compare component-based development to building structures using Lego blocks – like Lego, component-based development is about providing building blocks, which can then be used to build larger wholes.

Researching component-based development, the authors found four different approaches to component evaluation: Product line components, original components, quality components, and reusable components, all of which have their characteristics for evaluating a component. While there is some overlap, only the reusable component approach and its component characteristics were considered for this study.

Four different characteristics were found for evaluating reusable components: Understandability, adaptability, portability, and confidence While the authors do not expand on why they found these characteristics to be vital for reusability or provide further explanations of them, each of the characteristics can be split into multiple sub-characteristics. The characteristics and their sub-characteristics can be organized into a list as follows [Koteska & Velinov, 2017, tables 2-6]:

- Understandability
  - Documentation level
  - Observability
  - Complexity
- Adaptability
  - Customizability
  - Modularity
  - Generality
- Portability
  - External dependency
  - Compliance
  - Deployability
  - Replaceability
- Confidence
  - Maturity
  - Error tolerance
  - Observed reliability

Finally, the authors conducted a small survey with 18 respondents, whom they considered expert users of software components. While the sample size is small, the survey shows the respondents valued understandability the most and confidence the least when considering reusability. Additionally, the respondents' opinions of the sub-characteristics varied more than those of the main characteristics: Documentation level

and customizability were especially highly valued, while generality and replaceability were among the least valued.

4.5 Guidelines for component reusability

Jha & Mishra [2019] set out to research general guidelines for improving the reusability of a component in component-based development, noting the benefits reuse has in productivity, profitability, and decreased development cost. They go over some challenges related to component-based development, one of which is outlined as such: A component that is large in size has challenges with reusability, as size brings complexity.

The authors also note how not everything can be taken into account, the development of systems from reusable components needs to take the constraints of the project into account. They outline three different goals and constraints:

- Product characteristic constraints: functionality and quality.
- Development process constraints: the schedule and the cost.
- Maintenance process constraints: maintenance cost.

Reviewing reuse, the authors suggest eight different guidelines for improving component reusability [Jha & Mishra, 2019]:

- Language platform independency
  - Components should be developed using platform-independent languages.
- Small code structure
  - The structure of code in a component shouldn't be too large, and it should be easily testable.
- Robust documentation
  - The component and its use should be documented during development.
- Certification
  - Component certification should be considered.
- Optimized design
  - Component design should be optimized by keeping the next implementation in mind.
- Portability
  - New components should be developed with portability across various platforms in mind.
- Applied reuse metrics
  - Suitable reuse metrics should be applied to further understand reusable components.
- Standardization
  - The process of component retrieval should be standardized.

Finally, the authors talk briefly about component repositories. Components are retrieved from various repositories, and as repositories grow in size, retrieving compo-

nents becomes more and more difficult and time-consuming. Special care must be taken when placing components into repositories, so they are properly classified, cataloged, and certified, so finding the most suitable component and retrieving it is as efficient as possible.

4.6 A model for component reusability

Koteska & Velinov [2013] researched how reusability could be measured and calculated in component-based development and found existing metrics and models for calculating reusability lacking. No unified model exists, which leads to challenges in discovering how reusable a component is. In their study, the authors review existing reusability metrics, and because they found the existing metrics incomplete, their goal is to enrich them with additional attributes, which can then be considered when measuring reusability. Finally, the authors propose a unified model for measuring readability.

The authors go over both *black-box* and *white-box* reuse, where black-box reuse refers to a situation where the inner workings of the component to be reused are not known. White-box reuse, however, refers to a situation where the component and its inner parts can be observed in full. I will focus on the part about white-box reuse, as black-box reuse has little to do with readability, as no code can be read when black-box reusing a component. In addition to black-box and white-box reuse, *glass-box* reuse also exists, where the inner workings of a component can be seen, but they can not be altered in any way, whereas white-box, by definition, allows for modification of the component [Sharma et al., 2007].

After going over black-box reusability metrics, the authors present two formalized metrics proposed by Bi and others [2009]. The first metric is based on the relative amounts of new code and reuse, where a reusability value is derived from a relationship between total lines of code and lines of new code – the more new code is needed, the less reusable a component is. The second metric is based on software complexity, and there a readability value is derived from comparing a complexity value to a domain-specific reuse coefficient – the more complex a component is, the less reusable it is. Complexity, in turn, can be calculated via a multitude of methods, the authors cite the McGabe [1976] and Halstead [1977] methods as examples. To simplify, McGabe's [1976] method builds a graph out of possible paths through a program and derives complexity from the number of nodes and arcs. Halstead's [1977] method derives complexity from the number of operators and operands in the source code.

The rest of the study mostly concerns black-box reusability. However, the authors do list some general attributes which are considered important regarding reusability from other studies [Koteska & Velinov, 2013]:

- Understandability
- Usability
- Complexity

- Portability
- Adaptability
- Confidence
- Customizability
- Reliability
- Utilizability
- Maintainability
- Functionality

In addition, the authors propose two new attributes to add to the list: security and installability.

4.7 Issues in component reusability

In his study, Gill [2003] researched reusability issues in component-based development, noting that while component-based development has benefits in the reliability, stability, productivity, development time, and cost of software projects, it still has issues that merit consideration. He approaches component reusability through general software reusability, citing general software reusability guidelines as outlined by Poulin [1994], which are stated to often be similar to software quality guidelines:

- Ease of understanding
- Functional completeness
- Reliability
- Good error and exception handling
- Information hiding
- High cohesion and low coupling
- Portability
- Modularity

As these are more generic and not specific to component-based development, the author suggests the following five more specific guidelines when it comes to component-based development: Conducting software reuse assessment, performing cost-benefit analysis for reuse, adoption of standards for components, selecting pilot project(s) for wide deployment of reuse, and identifying reuse metrics. While all of these proposed guidelines have more to do with ensuring reuse is approached in the right way, and less to do with what makes a component reusable, some of them are still related to component-level reusability characteristics.

Reuse assessment should be the very first stage of reuse. A software organization considering reuse should thoroughly assess goals, strategies, targets, and plans around reuse – what is there to gain from reuse, how it should be approached, where it should be implemented, and a concrete plan of action. This is by no means a trivial undertaking, and it has both technical and organizational issues: On the technical side, the organization should identify the core business objects which can be made reusable, and define

clear standards and guidelines, structures, and classification schemes for reuse. On the organizational side, training programs are needed, as well as personnel support for the core reusable components, and an infrastructure of reuse metrics, measurements, policies, and incentives.

While a cost-benefit analysis alone shouldn't dissuade organizations from employing reuse, it's still an important consideration. Costs associated with reuse include the following: Increased documentation, maintenance of reusable components and their associated documentation, costs associated with a component repository, and the training of personnel.

4.8 Reuse challenges in component-based development

Crnkovic & Larsson [2002] studied the challenges of component-based development. While the challenges go far beyond reusability, a large part of the study concerns the challenges with reuse specifically. The challenges with reuse arise from multiple sources, but a big part of it is how everything around the component keeps changing and evolving. Requirements evolve, the system around the component evolves, environments change and the component might have to be migrated between platforms. Amid all this shifting, the component should be generic enough to be reusable and compatible with various systems, environments, and platforms.

The authors note, how a reusable component must fill two somewhat contradictory requirements: It should be generic enough to cover all different usage aspects, but at the same time, it should be simple enough to efficiently fulfill a single type of usage. Citing previous research, the authors go over how reusable component development needs three to four times more development resources than a component not made to be reusable, and additional complexity arises from incomplete and poorly understood requirements regarding reusable components.

The ability to adapt to change in multiple aspects surrounding a reusable component is crucial in developing a reusable component. Everything evolves, from the requirements of the system the reusable component is being developed for, to technologies used, and from organizational changes to societal changes. All of this has an impact on the product life cycle, whether it be by a simple change in system requirements, or even by a change in the currency used.

The requirements of software products change during their lifecycle, and for products built out of individual components, this means the requirements target the individual components. If a component is very reusable, chances are it will be used in a multitude of contexts, which in turn means differing requirements originating from very different software products. Changing requirements is more of an issue for components at the early stages of their lifecycle according to the authors, as early on the components will be less general and can't adapt to changing requirements without being changed themselves. Later on, the adaptability and generality of a component should increase,

which leads to less pressure to change the component due to product requirement changes. Finally, components become obsolete, as the changes and evolution of systems and technologies become too much for the component to keep up. Also, component cohesion is at risk of degrading when changes are made due to changing requirements.

As components are updated, new revisions of the component will need to be released. An important factor concerning reuse is the compatibility between different versions. While maintenance and improvement-related revisions usually lead to fewer compatibility issues, sometimes changes are necessary, and some changes can make it difficult for different versions to be compatible with each other.

Maintainability is also emphasized. The maintenance process in component-based development can be complex, as errors encountered in a product often originate from the individual components it consists of, and a new version of a component will also need to be tested in other products it’s being used in.

5 Analysis

The material appears to show clear patterns regarding readability. The results of annotator studies differ somewhat, most likely because of their different methods of arriving at the chosen readability features – Buse & Weimer [2009] asked annotators to grade code snippets, whereas Tashtoush and others [2013] asked annotators to grade individual features which intuitively and according to their sources had an impact on readability. Meaningful names and consistency are the most descriptive features regarding readability according to Tashtoush and others [2013] both of which are quite subjective and couldn't be extracted by Buse & Weimer [2009]. However, many features do appear in both studies. While Tashtoush and others [2013] do not state in their study whether the impact of a feature was negative or positive in regards to readability, except for a few features, comparing the results to the results of Buse & Weimer's [2009] study makes it quite clear.

Comments, identifier name length, indents, the amount of if-else statements, the amount of for- and while-loops, identifier frequency, and arithmetic formulas appear directly in both result lists, with just about the same predictive power regarding readability. Additionally, spacing, one of the most descriptive features according to Tashtoush and others [2013], is very similar to the *avg blank lines* metric from Buse & Weimer [2009]. Interestingly, while Tashtoush and others [2013] note arithmetic formulas as a negative regarding readability, Buse & Weimer's [2009] study shows the opposite. However, in both cases, the effect on readability is very small.

Incidentally, many similarities between the studies and Martin's book can also be found. Martin expands on the more subjective measures, such as meaningful names, comment quality, and consistency, as well as confirming the effect of formatting in the form of spacing and indentation. Choosing a meaningful name comes with many aspects, many of which weren't considered in the annotator studies. Assuming identifier name length as a negative descriptor in the study of Tashtoush and others [2013], as it is not outright stated, both annotator studies found lengthy identifier names to negatively affect readability. However, according to Martin's book, a name's effect on readability cannot be determined by length alone, the name itself is much more important. If the name adequately describes why the identifier exists, how it's used, and what it does, the name can be longer.

In addition to names, Martin demonstrates comments are also a more nuanced aspect of code, and their effect on readability cannot be discerned by simple metrics alone, such as their amount or their length. Both annotator studies list comments as a positive descriptor, while Martin is much more critical of them. While he does think comments can be good, Martin gives more examples of bad comments than good ones.

Formatting appears in both annotator studies in multiple forms: Line length, indents, spacing, the number of spaces, and blank lines. Formatting appears to lie somewhere between the most subjective features, such as meaningful names, and the least subjec-

tive ones, such as the number of various structures and keywords. The quality of formatting can be measured numerically through the metrics present in the annotator studies somewhat effectively, but they don't tell the whole story, which Martin expands upon. Namely, he touches upon subjects such as grouping related parts of code and emphasizes consistent formatting, which ties into consistency being one of the most descriptive features regarding readability according to Tashtoush and others [2013].

Martin also appears to agree with the notion proposed by Buse & Weimer [2009], where complexity is independent of readability and an inherent quality of software. He instead proposes designing classes in such a way, that the reader of the code is in contact with the least complexity possible, which in turn leads to better readability.

While multiple obvious parallels can be drawn between the works regarding readability, works regarding reusability seem to have less of a consensus. Koteska & Velinov [2013] even state outright, that existing reusability measures are lacking, and no unified model exists. However, like in the readability materials, complexity also appears in multiple reusability studies. Sharma and others [2009], Ismail and others [2017], Jha & Mishra [2019], and Koteska & Velinov [2013] all discuss complexity as a characteristic of reusability to various extents. Sharma and others [2009] discuss the complexity of interfaces, Ismail and others [2019] bundle complexity up as a part of understandability, Jha & Mishra [2019] mention how a large size makes a component more complex and Koteska & Velinov [2013] go so far as to suggest reusability could be calculated using software complexity metrics. All four of these studies agree with the notion that complexity is a negative descriptor regarding reusability.

Apart from complexity, the studies seem to differ somewhat, as the level at which the studies approach reusability differs - Sharma and others [2009], Ismail and others [2017], and Koteska & Velinov [2013] list characteristic-based guidelines for developing reusable components, while Jalender and others [2012] focus on more concrete design patterns, Gill [2003] goes over organizational guidelines, and Jha & Mishra [2019] go over more concrete actions to take during development. Still, both Jalender and others [2012] and Bigot & Pérez [2009] mention the separation of concerns, both Jha & Mishra [2019] and Gill [2003] emphasize component repositories and some other commonly discussed concepts can be found – multiple studies [Jalender et al., 2012; Sharma et al., 2009; Ismail et al., 2017] talk about good documentation, and Gill [2003] also notes the increased cost associated with increased documentation needs in component-based development. Ismail and others [2017] group characteristics under one another as sub-characteristics, and they put good documentation under understandability.

Interestingly, Buse & Weimer [2009] define readability as how easy something is to understand when read, which would indicate synergistic benefits between readability and component reusability. Additionally, Buse & Weimer [2009] state their study strongly correlates with software quality metrics, while Gill [2003] states general software reusability attributes also correlate with software quality.

In addition to splitting understandability into sub-characteristics, Ismail and others [2017] mention adaptability and split it into three: Customizability, modularity, and generality. Both Sharma and others [2009] and Koteska & Velinov [2013] mention customizability, while both Bigot & Pérez [2009] and Crnkovic & Larsson [2002] talk about generality/genericity, with the latter also explicitly mentioning, and even emphasizing, adaptability. Also, while the survey conducted by Ismail and others [2017] had a small sample size of 18, documentation level and customizability were found to be the most important sub-characteristics of reusability among annotators.

Portability and confidence, the last of the main characteristics mentioned by Ismail and others [2012] also appear in other studies. Portability especially is mentioned many times, and from the study by Crnkovic & Larsson [2002] it's clear portability is closely related to adaptability – in the case of both attributes, the main point is to fit into multiple environments and to keep up with changing platforms, technologies, and requirements. Requirements are also mentioned by Jalender and others [2012], who greatly emphasize their effect.

Overall, while parallels can be drawn between both readability and component reusability studies, readability appears mostly related to specific, code-level features, while reusability seems to tie in more with broader design patterns, concepts, and overall characteristics, which are harder to specifically define and quantify. However, Martin talks about classes in length, and about how designing classes correctly leads to cleaner code. He goes on to cite the Single Responsibility Principle, where each class should only have a single reason to change and emphasizes the importance of organizing object-oriented code into very small, single-purpose classes. While classes and objects aren't the same concept-wise, they share many qualities, and the motivation behind both approaches is the same – breaking down a larger whole into smaller, organized, and more manageable parts. [O'Reilly, 2023]. The usage of components does differ from class usage somewhat: Using components often doesn't require the user to be familiar with the inner workings of a component, as components can be used interchangeably and independently [O'Reilly, 2023]. However, as the focus of this thesis is to find parallels between readability and reusability, the possibility of using components without reading them isn't relevant. Thus, applying Martin's guidelines on clean class design to component design seems reasonable.

Three distinct guidelines on class design can be found in Martin's text: First, classes should be as small as possible and only have a singular responsibility. Second, classes should be cohesive, and if not, breaking them down into smaller classes should be considered. Third, classes should be decoupled from one another through the use of interfaces to avoid volatile dependencies. The third guideline doesn't apply to components, as components themselves define an interface through which it interacts with other components and parts of an application. The first two can be applied to component design, but even this doesn't reveal any conflicts between readability and reusability – a

component can be easily split into smaller sub-components if need be, and even a small component designed to fulfill a very specific purpose can be customizable, and even generic when it comes to its implementation.

6 Discussion

As stated before, the initial hypothesis of this thesis is that the readability of code and the reusability of software components conflict with one another. The study was done as a literature review, and information from three readability studies was synthesized together, along with eight reusability studies of varying levels. The studies in readability correlate with each other quite well, and a consensus can be found: The characteristics of code readability can be found in specific, code-level features, such as formatting, naming, and comments. The studies in reusability offer less of a consensus, but parallels exist: The studies suggest reusability is related to more abstract concepts than reusability, including design patterns, organizational guidelines, actions to take during development, and the main focus of this thesis; characteristics of the components themselves, such as understandability and adaptability.

While trying to make a component as reusable as possible appears to lead to readability problems through increased size and a larger number of lines, the root cause might not be in reusability per se – it might be in making the component fit too many roles, a thought supported by Martin's [2014] class design guidelines when applied to component design. This is likely to be the key finding of this thesis, and the reason why reusability appears to cause readability problems. While a large, jack-of-all-trades type of component might be very reusable in a very literal sense of the word, such a component goes against the idea of splitting software into smaller parts with their distinct purposes. Special care should be taken to keep components focused on their core purpose, and if need be, splitting components into smaller sub-components if they appear to become bloated and harder to read. An obvious synergy arises between readability and component reusability – A component that is easier to understand, is also easier to reuse.

It was harder to find a consensus on the measures of component reusability – as Koteska & Velinov [2013] said, no unified model exists for reusability, and existing reusability metrics are incomplete. However, all studies regarding readability seem to support one another very well, probably because programmers have an intuitive notion of readability [Buse & Weimer, 2009]. Readability as a concept is also attributed to discernible features and parts of source code, which are independent of any overarching design patterns or architecture, which could make it easier to quantify readability. Reusability, being seemingly more attributed to design and other such more abstract concepts, appears to be quite a difficult thing to measure concretely.

Interestingly, while no absolute universal consensus regarding the characteristics of reusability could be found, the most often repeated themes also had synergistic ties to readability. Complexity and understandability appear in both fields, but Buse & Weimer [2009] show in their study, that complexity and readability are only weakly correlated. They point out readability is more a product of simpler, surface-level features of source code, rather than relating to the actual function of the code. Sharma and others [2009] seem to contradict this notion somewhat when they state interface complexity as a

measure for component reuse: ”Complex interfaces will lead to the high efforts for understanding and customizing the components” [Sharma et al., 2009, page 4]. They recognize higher complexity as a factor, which makes a component harder to understand. A point could be made, where understandability is not the same as readability, but Buse & Weimer [2009] defined readability in their study as such: ”We define readability as human judgment of how easy a text is to understand” [Buse & Weimer, 2009, page 546]. It seems readability and complexity do correlate somewhat, but as Buse & Weimer [2009] show, the correlation is rather weak, especially when compared to more descriptive measures of complexity, which would explain the contradiction somewhat – as increased complexity doesn’t in itself lessen readability very much, it also doesn’t make a component too much harder to reuse.

7 Conclusion

In this thesis, I studied both the reusability of software components and the readability of source code. Readability comes from how code is written, not necessarily the length of it – formatting and naming go a long way in making code readable. Component reusability, however, is not as easily measured. Characteristics of reusable components exist and studies show some consensus, but pinpointing reusability in individual features is not simple. Reusability comes from more abstract concepts, such as adaptability and understandability, the latter of which can be tied to readability.

While readability is not in direct conflict with component reusability, poorly executed reusable component design can lead to issues with readability, which can obscure the true source of the issues. Keeping components clean and splitting components into sub-components whenever the component is fulfilling too many purposes at once will increase readability while maintaining reusability.

The findings emphasize the importance of design and maintenance in component-based development. Components should be designed to concisely fit a single role, and doing multiple unrelated things with a single component should be avoided. The code used to write said components should follow established guidelines for readability. Thus, reusability and readability can both be improved at the same time.

8 References

Abbas, N. (2010). Properties of “good” java examples. In Proceedings of the Umea’s 13th Student Conference in Computer Science. 66 pages.

Airbnb. (2023). Airbnb/javascript: JavaScript Style Guide. GitHub. Available at: https://github.com/airbnb/javascript. Accessed April 2023.

Batool, A., Bashir M. B., Babar, M., Sohail, A., & Ejaz, N. (2021). Effect or Program Constructs on Code Readability and Predicting Code Readability Using Statistical Modeling. Foundations of computing and decision sciences, vol. 46, no. 2, pp. 127–145.

Bi, S., Dong, X., & Xue, S. (2009). A Measurement Model of Reusability for Evaluating Component. In Proceedings of the 2009 First International Conference on Information Science and Engineering, pp. 20-22. IEEE.

Bigot, J., & Pérez, C. (2009). Increasing reuse in component models through genericity. In Formal Foundations of Reuse and Domain Engineering: Proceedings of the 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009, pp. 21-30. Springer Berlin Heidelberg.

Bittner, K., & Spence, I. (2006). Managing iterative software development projects. Addison-Wesley Professional.

Buse, R. P. L., & Weimer, W. R. (2010). Learning a Metric for Code Readability. IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 546-558.

Cole, M. (2004). Bringing Skeletons Out of the Closet: a Pragmatic Manifesto for Skeletal Parallel Programming. Parallel computing vol. 30, no. 3, pp. 389–406.

Crnkovic, I., & Larsson, M. (2002). Challenges of component-based development. Journal of Systems and Software, vol. 61, no. 3, pp. 201-212.

Crnkovic, I., Chaudron, M., & Larsson, S. (2006). Component-Based Development Process and Component Lifecycle. In Proceedings of the 2006 International Conference on Software Engineering Advances (ICSEA'06), Tahiti, French Polynesia, p. 44.

Gill, N. S. (2003). Reusability issues in component-based development. ACM SIGSOFT Software Engineering Notes, vol. 28, no. 4, p. 4.

Halstead, M. H. (1977). Elements of software science. Elsevier.

Ismail, S., Wan Kadir, W. M. N., Noor, N. M. M., & Mohd, F. (2017). Determining characteristics of the software components reusability for component based software development. Journal of Telecommunication, Electronic and Computer Engineering, vol. 9, nos. 3-5, pp. 213–216.

Jalender, B., Govardhan, A., & Premchand, P. (2012). Designing code level reusable software components. International Journal of Software Engineering & Applications, vol. 3, no. 1, p. 219.

Jha, S. K., & Mishra, R. K. (2019). A review on reusability of component based software development. Reliability: Theory & Applications, vol. 14, no. 4, pp. 32-36.

Johnson, S. C. (1977). Lint, a C program checker. Murray Hill: Bell Telephone Laboratories.

Kincaid, J. P., Fishburne Jr, R. P., Rogers, R. L., & Chissom, B. S. (1975). Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Naval Technical Training Command Millington TN Research Branch.

Knight, J. C., & Myers, E. A. (1991). Phased inspections and their implementation. Software Engineering Notes, vol. 16, no. 3, pp. 29–35.

Koteska, B., & Velinov, G. (2013). Component-Based Development: A Unified Model of Reusability Metrics. In ICT Innovations 2012 (pp. 335–344). Springer Berlin Heidelberg.

Martin, R. C. (1996). The open-closed principle. More C++ gems, vol. 19, no. 96, p. 9.

Martin, R. C., & Feathers, M. C. (2009). Clean Code : a Handbook of Agile Software Craftsmanship. Upper Saddle River, N.J: Prentice Hall.

Martin, R. C. (2014). The Single Responsibility Principle. Blog. Available at: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html. Accessed April 2023.

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, no. 4, pp. 308-320.

Návrat, P., & Filkorn, R. (2005). A note on the role of abstraction and generality in software development. Journal of Computer Science, vol. 1, no. 1, pp. 98-102.

npm (2023). About npm. Node Package Manager documentation. Available at: https://docs.npmjs.com/about-npm. Accessed April 2023.

Ogut, C. (2021). What is Linting? How does a linter work? cogut.medium.com blog. Available at: https://cogut.medium.com/what-is-linting-how-does-a-linter-work-49381f28fc60. Accessed Feb 2023.

O'Reilly (2023). Component-Oriented Versus Object-Oriented Programming. Available at: https://learning.oreilly.com/library/view/programming-net-components/0596102070/ch01s02.html. Accessed April 2023.

Poulin, J. S. (1994). Measuring software reusability. In Proceedings of the 1994 3rd International Conference on Software Reuse, pp. 126-138. IEEE.

Pressman, R. S. (2001). Software engineering : a practitioner's approach (5. ed.). McGraw-Hill.

Seitz, B. (2022). YAGNI. Available at: http://webseitz.fluxent.com/wiki/YAGNI. Accessed April 2023.

Sharma, A., Kumar, R., & Grover, P. S. (2007). A critical survey of reusability aspects for component-based systems. International Journal of Industrial and Manufacturing Engineering, vol. 1, no. 9, pp. 420-424.

Sharma, A., Grover, P. S., & Kumar, R. (2009). Reusability Assessment for Software Components. Software engineering notes vol. 34, no. 2, pp. 1–6.

Stanic, M. (2022). Benefits of teamwork and collaboration for software development teams shakebugs.com blog. Available at: https://www.shakebugs.com/blog/collaboration-in-software-development/. Accessed April 2023.

Tashtoush, Y., Odat, Z., Alsmadi, M. I., & Yatim, M. (2013). Impact of Programming Features on Code Readability. International Journal of Software Engineering and Its Applications vol. 7, no. 6, pp. 441–458.

Wang, X., Pollock, L., & Vijay-Shanker, K. (2011). Automatic segmentation of method code into meaningful blocks to improve readability. In Proceedings of the 18th Working Conference on Reverse Engineering, pp. 35-44. IEEE.

Yung, C. (1997). Simplified readability metrics.

Zolkifli, N. N., Ngah, A., & Deraman, A. (2018). Version control system: A review. Procedia Computer Science, vol. 135, pp. 408-415.