Software quality and dependability concepts, terminology - Software Complexity and Maintainability

Gonçalo São Marcos
2018284198
Universidade de Coimbra
gmarcos@student.dei.uc.pt

José Esperança
2018278596
Universidade de Coimbra
esperanca@student.dei.uc.pt

April 1, 2022

Abstract

Discussion focused on the main concepts of software quality and dependability as well as highlighting the importance of software maintainability in the current industry of software development. Closer look at how code complexity affects software maintainability, especially in large systems, and how that reflects on costs, ending with how this results could be applied to the industry and what needs to be researched further.

1 Introduction

Software is all around us and, nowadays, we certainly could not live without all of the systems we interact with every day. From the calculator application on our phone to the banking system that makes sure all our bills get payed without issues, we are completely surrounded by systems we depend on and that are crucial for very important tasks. The banking system is a perfect example of a piece, or, in fact, thousands of pieces, of software that must work as intended or else the whole world economy would crumble.

When tackling software development challenges we must always keep in mind the quality of the software that is being produced.With that comes the added challenge of making that software maintainable in a cost and time effective manner, in order to keep up with the requirements that are ever changing.

2 Overview on the main concepts of Software quality and dependability

Firstly we have to discuss the definition of software quality. IEEE Std. 730-2014 [68314] defines it as ”The degree to which a software product meets established requirements; however, quality depends upon the degree to which established requirements accurately represent stakeholder needs, wants and expectations”. The exact definition of the concept touches upon two fundamental ideas: the software does what it is required to do and matches the stakeholders' needs, wants and expectations. With this in mind we can affirm that to build high quality software we must focus on designing a system that does what the stakeholders' want it to do while keeping track and implementing all of the usually very dynamic requirements.

The requirements of software are always one of the trickiest parts since usually the source is the client, who most of the time isn't a specialist in software development, wants to implement a certain feature and the developers must follow those orders. We can look at requirements from two perspectives: Functional or Non-Functional.

Starting with Functional Requirements, the more direct and trivial type to explain and comply with, these focus on what the software does and the quality attribute in this view is related to how well the implemented functionalities match with the user's needs and expectations.

On the other hand, Non-Functional Requirements (also referred to as Quality Attributes) focus on how the system does it in regards to features such as performance, security, reliability, availability, usability, maintainability, etc.. The quality in the perspective of these types of requirements is much harder to measure directly. These Quality Attributes (or Non-Functional Requirements) come up at the very beginning of the development process and they vary depending on the priorities that are established by the stakeholders and by the nature of the system itself. For example, a banking system must set a very high priority in regards to security and reliability but, at the same time, a piece of software that is created purely for entertaining might set reliability as a lower priority attribute than usability since that is its main goal. Priorities of such attributes might also change during the development process or even while the software is already deployed (e.g. a newly added store service might force the whole system to be more secure than originally planned).

A software product must also be dependable. We should be able to trust that the service won't fail when it is needed and that those failures are not frequent or severe. Dependability encompasses multiple system attributes such as: availability, confidentiality, integrity, reliability, safety and maintainability [MFC21]. From a dependability point of view, when a fault occurs in the system it leads to an error (i.e. wrong change in the state of the system), which consequently can lead to failure, that is, an incorrect system response. As a consequence of that, faults are a huge threat to the dependability of a system and we must implement mechanisms that try to prevent, tolerate, remove or even forecast those faults.

3 Software maintainability

Looking back at the main attributes of dependability we find one that, at first glance, might go unnoticed and that is: Maintainability. This Quality Attribute is defined by the IEEE as "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment". This attribute is, in fact, crucial to the quality of the software, since virtually no system is deployed and not tinkered with after the fact.

Maintainability has a lot of components associated with it, with the most relevant being:

- Modifiability
- Understandability
- Readability
- Ease of integration

Messy, uncommented and confusing code will certainly cause an headache to the person trying to modify it, which, in turn, will cost more time and money. These factors also influence the ability to easily integrate new code, such as new features or upgrades, or insert restructured and rewritten code when a module is completely substituted by a newly coded version.

If we look at dependability's threats we can easily see the benefits of better maintainability since it is essential to remove faults and assure the quality of the software itself, specially in a long term perspective. The harder a system is to maintain, the more time, money and general resources will be spent during maintenance tasks. It's impossible to keep up with the needs of the clients without maintaining software, that is, maintainability is essential to all other Quality Attributes.

This all escalates when we observe the actual costs of a software project during the entirety of it's life cycle. Everyone understands that there are a lot of costs associated with software maintenance, especially when there is a Author Daniel D. Galorath estimates that 75% of all project's costs are associated with maintenance [CAS$^+$17], while Jussi Koskinen predicts that those could go up to 90% and confirms that this percentage has been rising throughout the years [Kos15]. It is, therefore, imperative that maintainability is a core attribute of all systems, especially those built with long term goals.

4 Code complexity

The complexity of a piece of software is something that has been heavily studied throughout the past decades. IEEE Std. 730-2014 [68314] defines it as "the degree to which a system or component has a design or implementation that is difficult to understand and verify". It is a highly valuable characteristic of the quality of code, given that it can be used as a quality metric of all the code built by a company. Simpler, less complex code is obviously easier to maintain and analyse, providing companies an easier way to make sure everything works correctly. Considering this utility, it is important to the whole industry that it is possible to measure and use code complexity correctly.

4.1 Measuring code complexity

Even though code complexity is relatively easy to define it is really hard to measure accurately. The definition is somewhat intuitive - the harder something is to understand, the more complex it is. But with it comes along a more bigger challenge, trying to measure it in a objective, non-biased way. The problem comes, as almost all do within the software development industry, from people. There has to be someone interacting with code for it to have its complexity. But everyone is a different individual and everyone learns, interprets and thinks in different speeds and ways. Therefore, it is impossible to make a one-size fits all metric, as something being difficult is subjective to the individual that is interacting with it. Nonetheless, it hasn't stopped people from trying it, as there have been developed many metrics to estimate the complexity of the code.

The first and the most basic one is Lines of Code, or LOC. It's probably the way most novices look at their code, but it has some value associated with it. A stretch of code that is longer than it needs to be causes confusion and fatigue to the people reading it, leading to worse readability and understandability.

The second one we'll be talking about is McCabe's Cyclomatic Complexity. It's used to determine the number of independent paths in the control flow graph of a program and it's one of the most popular code complexity measures.

Based on the number of function invocations, there are the Fan-in and Fan-out metrics [Hos15]. The former represents the number of functions that call the function we're measuring and the latter represents the number of functions calls made by the function we're evaluating.

There's Buse and Weimer's metric [Pin], which was made to evaluate the readability of small code snippets, using an original tool developed by the metrics' authors, which is based on simple features like word and sentence length.

Considering change measures, we have the number of revisions of a file and the number of developers who worked on a file, both being relatively self-explanatory.

Finally there are Halstead's metrics [IBM21]. These are 7 metrics based on the number of operators and their instances, which are plugged in mathematical formulas to calculate the output of each measurement.

Considering all these metrics, which one of them is the best? All of them focus on specific areas, bringing with them their pros and cons, so the answer is probably "it depends". There are also cases of the best results being obtained using models based in multiple metrics [ASH$^+$14].

4.2 Factors that affect code complexity

As mentioned in the previous subsection, there are countless factors that affect the complexity of a certain system's code. Authors have researched this theme for years and we found a very interesting take by Banker et al., 1989 [BDZ89].

First of all we can look at what causes complete software. A system and its software can be complex simply because its functions are to complete complex tasks, even so, it is very possible to poorly code a complex or even simple task, making it more complex than it optimally should be. The initial design decisions associated with the code, caused by poor programming practices might also influence dramatically the final complexity of the program. Modifications to an already existing system can also affect how complex the code gets after poorly planned modifications during maintenance tasks.

Nowadays we have access to many software development tools that force programmers to adopt ways of coding that will eventually help reduce the complexity of badly designed code result of poor programming practices. This also allows us to easily infer that older code is usually more complex because it has undergone many maintenance years and changes which might have had a negative effect on its complexity (such as adding modules that were never planned to be added at the start). Older systems were also written at a time where software maintainability was not a priority factor and there wasn't much awareness about those concerns. We will talk more in depth about how code complexity affects maintainability in the next section. The authors of this paper suggest that the complexity of a certain software can be estimated as a function of six distinct factors:

- Software Tools → Using the appropriate software tools and development methodologies can help to write less complex and more maintainable code
- Volatility → If a system undergoes more frequent maintenance it can cause those tasks to be more rushed and less planned which can in turn degrade its complexity over time
- Functionality → If the system itself if more ambitious in terms of functionality it will most likely require more ambitious code which might make it harder to maintain good levels of software complexity
- Age → The sophistication of programming methods at the time of writing the code influences the overall complexity
- Operating Requirements → Imposition of operating requirements may force programmers to pay less attention to maintainability and code complexity concerns
- Error Rates → Directly affects the complexity of software

The combination of all these factors can help estimate the overall complexity of a system's software just by analysing them.

Antinyan et al., 2017 [ASS17] go a step further and define eleven different (and distinct from Banker et al., 1989) code characteristics in a survey, which the authors used to question 100 participants. The results of their work can be seen in **Figure 1**. The authors also conclude that the study shows that the two main sources of complexity in software - lack of structure and nesting depth - might be related to accidental complexity. This accidental complexity is not related to the complexity inherent to the problem but is caused by a direct consequence of poor design decisions and/or non-optimal methods of programming. The graph also shows the different levels of influence on complexity of the different defined code characteristics.

This study also concluded that these complexity increasing factors do not vary with neither the job nor the experience of the developer, which means that they are relevant for every company and developer.

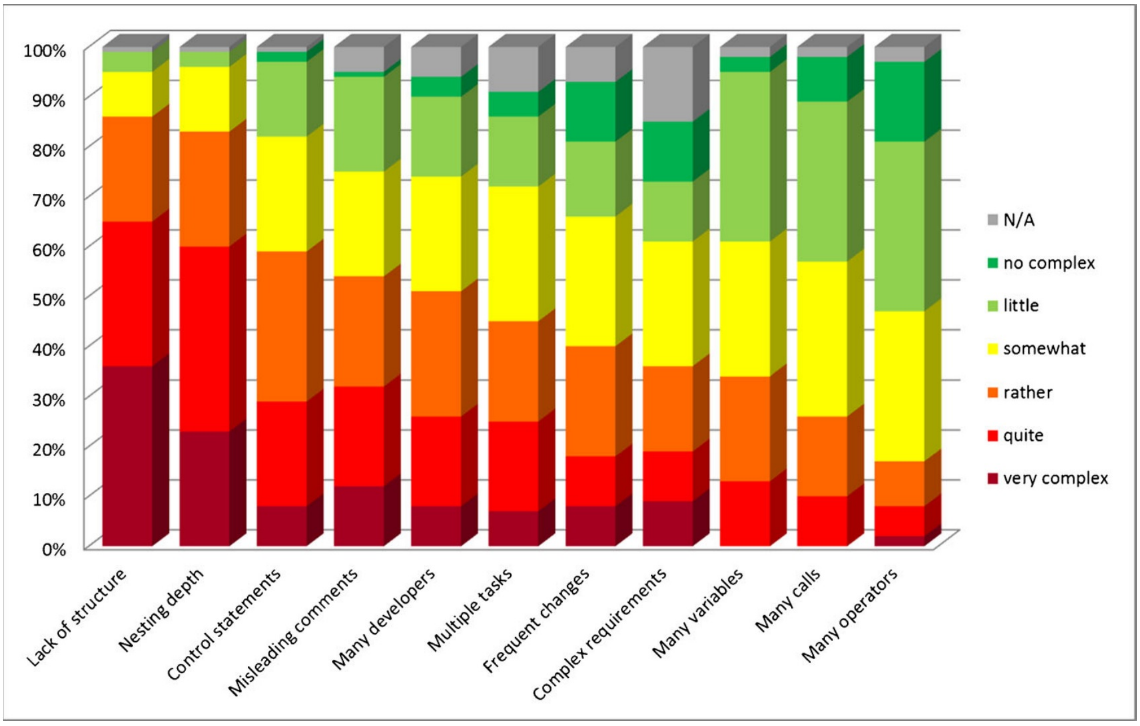


Figure 1: Influence of code characteristics on complexity [ASS17]

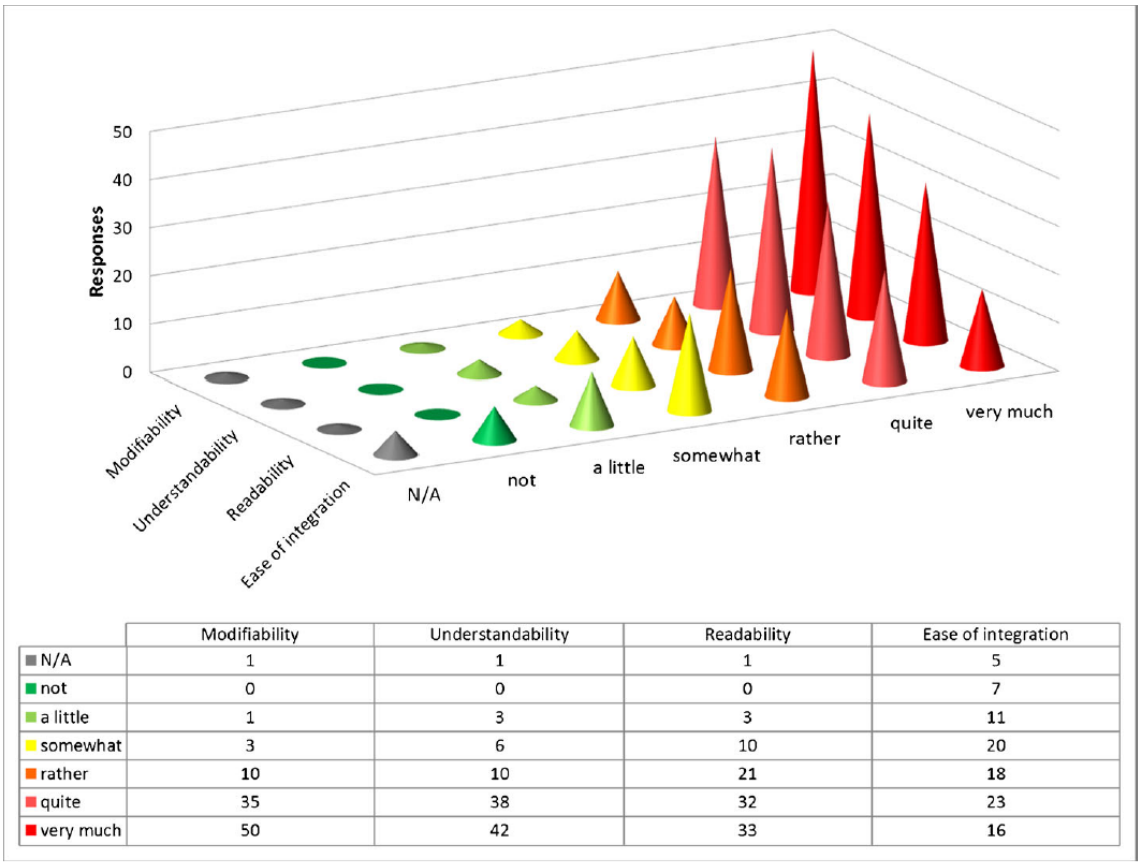
5 How is maintainability affected by code complexity

One of the main causes for maintenance is often closely related to error rates in the system's software. If an error is detected in a piece of software it can lead to erroneous behavior of the system and that must be fixed through maintenance procedures. Well, a system that is very complex is more prone to higher error rates since the code might be more difficult to comprehend and consequently programming errors are more likely to go unnoticed. More complex software will also be harder to test and that can also lead to a decreased likeliness of catching said errors.

Banker et al., 1989 [BDZ89] suggest that, through questionnaires, the mean error rate could be estimated based on five factors: software volatility, developer, application experience, primary user and software complexity. Software volatility can be seen as the frequency at which changes are made to the application software, since more frequent changes usually mean more chance to introduce new errors into the software. The developer that is assigned to a certain task also influences the error rate since, for example, outsourced developers are less familiar to the maintainers and the company structure. The experience, or lack thereof, on the application itself can also cause some errors to go unnoticed. The primary user of the application is also a deciding factor because more thought and care will be put into software that is intended for clients than, for example, into software that is intended to be used in another department in the same company by people who are more familiar with the process. Finally, the software complexity itself influences the error rates since more complex software is more likely to generate errors than simpler, easier to read code.

We can start to see how code complexity will affect maintainability directly: more complex code will generally mean the software will have to be maintained more and it also causes that same software to be harder to maintain. There's an apparent vicious cycle that developers must keep in mind when in the developing phase of the software to avoid the unnecessary escalation of maintenance costs.

In Antinyan et al., 2017 [ASS17], the authors conclude from a study that the respondents agree that complexity has a huge influence on readability, understandability and



	Modifiability	Understandability	Readability	Ease of integration
■ N/A	1	1	1	5
■ not	0	0	0	7
■ a little	1	3	3	11
■ somewhat	3	6	10	20
■ rather	10	10	21	18
■ quite	35	38	32	23
■ very much	50	42	33	16

Figure 2: Influence of code complexity on internal code quality attributes [ASS17]

modifiability (**Figure 2**). Well, modifiability is a very important pillar of code maintainability and it was found that complexity affects it the most.

They also go further into the influence of software complexity on maintenance time (**Figure 3**), arguing that if complexity were to have a small effect on maintenance time, then we wouldn't need to spend so much resources and put so much effort into trying to limit that complexity. In **Figure 3** we can clearly observe that the inquired people strongly agree complexity is a major influence on maintenance time of code. Around 27% of the inquired people, the modal value of the survey, agree that the influence is in the order of 250-500% of additional time required for maintaining complex code compared with a simple code of the same size.

Having studied the author's survey results and other articles, we agree that the more complex a piece of software is, the more difficult it will be to perform maintenance tasks on said software and more likely it will be that new errors are introduced. When approaching maintainability it is always wise to look at it from a financial point of view, i.e. maintenance is a very expensive part of running a system (as discussed briefly in the introduction). Generally, more hours of labor will be put into maintenance if the code is more complex so that easily means the overall cost of a more complex system will be higher during its life cycle than that of a more simple system.

Banker et al., 1989 [BDZ89] model the estimate of the amount of label hours as a function of seven factors:

- Task Magnitude → If more code is to be modified, the more work will be required
- Task Complexity → Some tasks are just more difficult and demanding than others
- Programmer Skill → Better rated programmers work better and more efficiently. This can be obtained through corporate personnel files

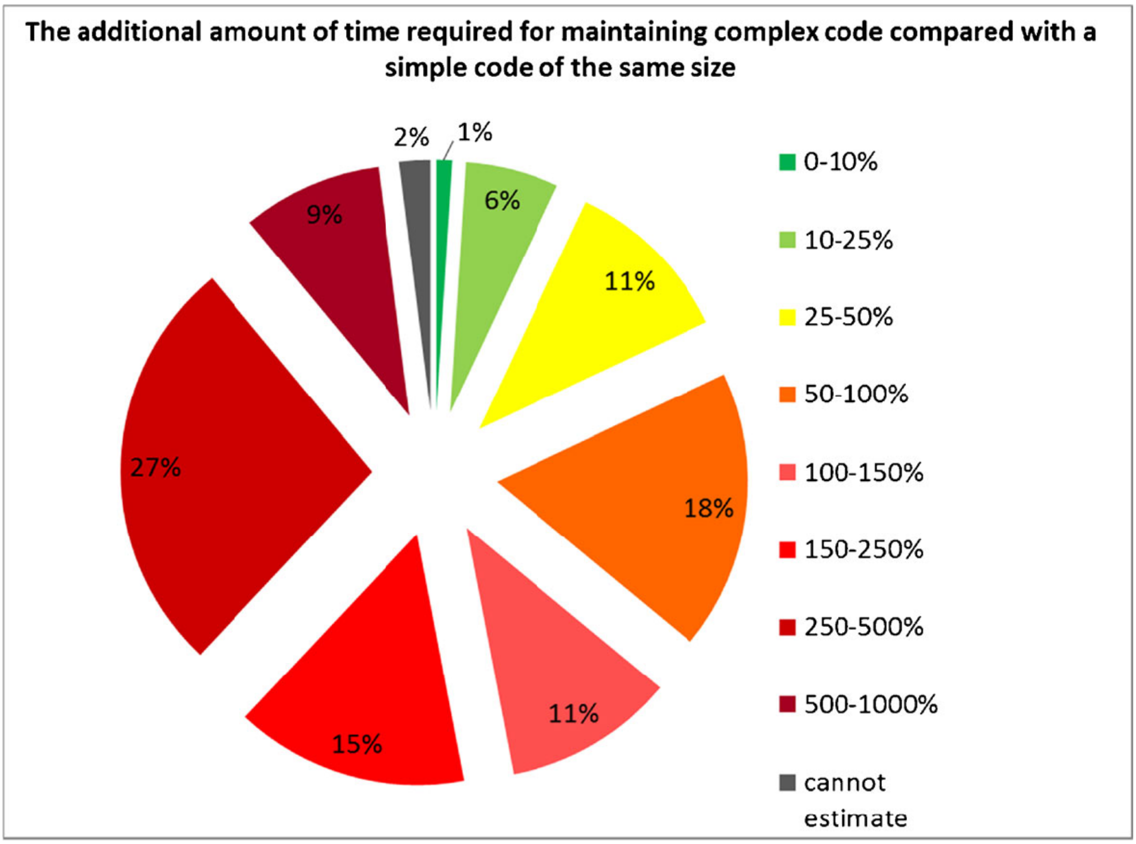


Figure 3: Influence of complexity on maintenance time of code [ASS17]

- Programmer Experience → "Even a good programmer is at a disadvantage when faced with an unfamiliar system". This information can also be obtained through corporate personnel files

- User Skill and Involvement → Requests for modifications usually come from user's needs. A more unsophisticated user will usually generate more confusing requirements which in turn makes the workload more difficult for the programmer

- Software Tools → There exists tools designed to increase programmer productivity which will help with maintenance costs

- Software Complexity → A programmer will need more time to analyse and maintain a more complex piece of software due to the ability to understand the code

If it is found that system complexity is influencing and increasing maintenance costs it should be looked into. If the cost impacts are small, managers may opt to leave it be. It may also be possible to identify the key components that are too complex and just modify those accordingly. A system might be so complex it could be easier and cheaper to rewrite it and replace it than to modify it [BDZ89].

6 Use of complexity measuring in the industry

In Zhang et al. [ZZG07], the authors propose a complexity based method for predicting defective components in a system. Through thorough testing and validation they prove the model works well with real world sets of data. The model in question puts software complexity at the centre and measure it in three different manners: Lines of Code (LOC), McCabe's Cyclomatic Complexity ($V(g)$) and Halstead's Volume (V) which are static code attributes commonly used for measuring code complexity. More detail about these

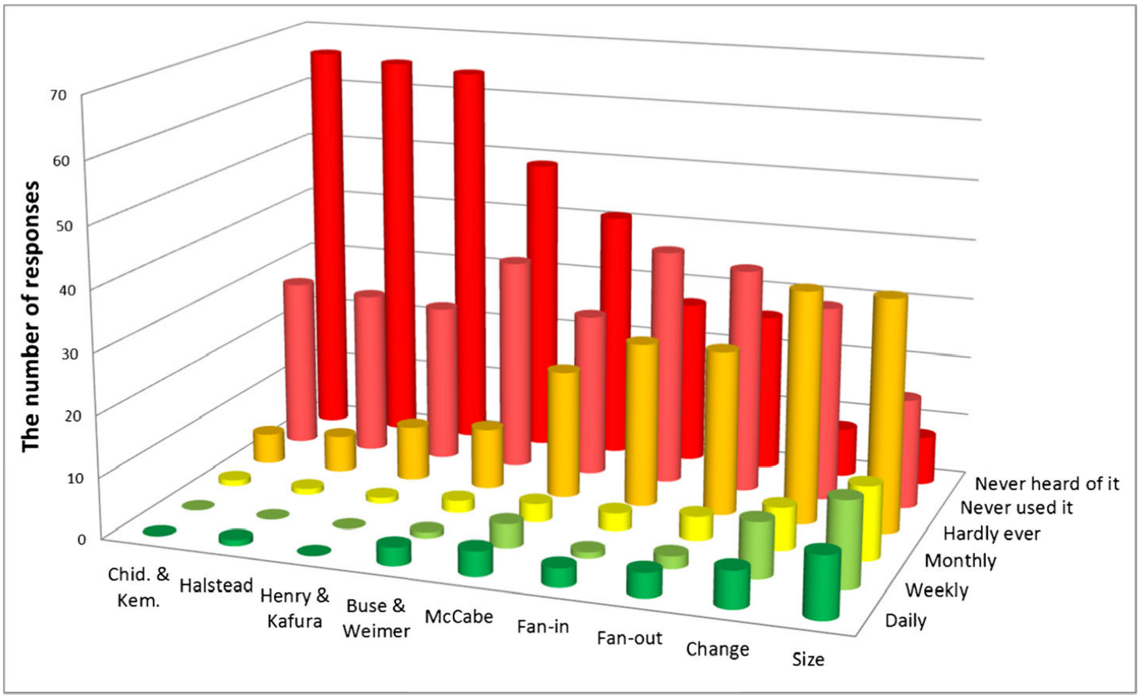


Figure 4: Use of complexity measures in Industry [ASS17]

measurement types are discussed in Section 4.1. These metrics are used in the model as defect predictors and the model itself classifies the tested components in two different classes: defective or non-defective. The results observed by the authors are satisfactory and they affirm that "static code complexity measures can be useful indicators of component quality and that using classification techniques, we are able to build effective defect prediction models based on the code complexity measures." [ZZG07].

With a working model that can correctly predict defective components, managers of systems can more efficiently plan and execute maintenance operations in a way that is less costly. Although pretty effective models like the one proposed by Zhang et al. [ZZG07] exist, most companies still choose not to adopt such measuring of complexity for various reasons that are briefly touched upon by Antinyan et al., 2017 [ASS17]. The reasons listed by the authors state that some company regulations don't even consider those measures or they already implement another set of measures, developers don't believe in the use of such measures since it could take even longer than not using it, some believe that the measures are just not a good indicator of complexity or that they don't really help understanding the code and finally some believe that the tool support is not satisfactory.

In **Figure 4** we can see the results of a survey that registers the use of complexity measures in the industry in terms of how frequently those measures are used and we can clearly observe that the measures are still not widely adopted in the industry.

7 Future Work

During the making of this paper, we noticed some areas that need further exploring in order to fully implement code complexity into the software development workspace.

In the first place, it is really important that the complexity metric chosen is highly accurate at measuring the user's perception of complexity. This is, there are a substantial amount of cases where the measured complexity of a problem decreased, but the perceived complexity was not reduced or, in some cases, was even increased. We assume this is because, as all things in life, the real solution is usually a mix of ideas, so it is needed to create a new measurement taking into account all these influences found, similar to the models talked about in section 6. There should be more experiences in this area in order to find the optimal combination.

We also believe it could be really interesting to repeat the study done on the influence of code characteristics on complexity [ASS17], but with an increased number of factors, being that they could be missing an important factor that was not tested for.

8 Conclusions

In summary we believe that software complexity can be a very powerful tool that might allow companies to better organize and execute maintenance in their systems. These models and measures can help increase time and cost effectiveness of maintenance tasks that are one of the biggest costs of software development: maintaining it to keep it dependable. The measures of code complexity described are useful to estimate how the maintainability of the system might be affected and, thus, giving the companies ways to better project maintenance tasks and predict faulty modules of their software. Although we believe these models and the study of the complexity of software are very useful, we also understand that it is still not a big priority in the industry and we agree it should be more widely adopted by the industry as a way to better create and maintain systems so they can be relevant and less expensive over their lifecycle.

References

[68314] Ieee standard for software quality assurance processes. *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, pages 1–138, 2014.

[ASH+14] Vard Antinyan, Miroslaw Staron, Jörgen Hansson, Wilhelm Meding, Per Österström, and Anders Henriksson. Monitoring evolution of code complexity and magnitude of changes. *Acta Cybernetica*, 21:367–382, 01 2014.

[ASS17] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, 22(6):3057–3087, Dec 2017.

[BDZ89] Rajiv Banker, Srikant Datar, and Dani Zweig. Software complexity and maintainability. pages 247–255, 01 1989.

[CAS+17] Celia Chen, Reem Alfayez, Kamonphop Srisopha, Barry Boehm, and Lin Shi. Why is it important to measure maintainability and what are the best ways to do it? pages 377–378, 2017.

[Hos15] Muhammad Iqbal Hossain. How do i measure fan-out/fan-in? https://www.researchgate.net/post/How-do-I-measure-FAN-OUT-FAN-IN, 2015. [Online; accessed 31-march-2022].

[IBM21] IBM. Halstead metrics. https://www.ibm.com/docs/en/rtr/8.0.0?topic=SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm, 2021. [Online; accessed 31-march-2022].

[Kos15] Jussi Koskinen. Software maintenance costs. https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf?version=2&modificztionDate=1430404596000&api=v2, 2015.

[MFC21] Henrique Madeira, João Fernandes, and Frederico Cerveira. Slides software quality and dependability (mei) (qualidade e confiabilidade de software). 2021.

[Pin] Cláudio Pinto. Buse weimer metric. https://cdtpinto.github.io/pages/bw.html. [Online; accessed 31-march-2022].

[ZZG07] Hongyu Zhang, Xiuzhen Zhang, and Ming Gu. Predicting defective software components from code complexity measures. pages 93–96, 2007.