Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Towards scientific software improvement - Proposing a methodology

Author: Antonios Orestis Roussos (2574589)

1st supervisor: Adam Belloum
2nd reader: Zhiming Zhao

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

May 8, 2019

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

An immense amount of research software is constantly being developed, which quite often ends up dumped in the organisations' archives once it's served its purpose. Such software products, would be quite useful if they were adopted by the open source software community, as they have been born as tools for research and therefore progress. As they remain unused, the need for software quality in the field arises even higher. This leads to the need to answer our research question, "How can the software quality of scientific software be improved?". This report aims to identify and address important aspects of software quality research that could be improved during the software products life-cycle, and proposes a methodology to do so. The important goal we aim to achieve, is to contribute in converting/transforming/evolving such products into self-sustained software through the proposed methodology. The methodology involves several continuous software measurements to provide insight and control over the developed software, along with NASA's Technology Readiness Levels, in order to provide concrete evidence and build not only on technical, but also social aspects of the software.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Preface

1.1 Introduction

Nowadays, software is being produced rapidly, both by organisations in the commercial market as well as for research purposes. Software systems are being continuously developed, aiming to achieve their corresponding goals. With all this available software, which grows in number and size day by day, there are several points of interest arising. In order to take a deeper look at this topic (along with defining the purpose of the present report), we are dividing the software systems mentioned into two big groups, that of Commercial Software, and the Open Source Software one. Such specific groups are chosen, as they enable us to address the concerns that belong in each of them, individually.

Even though the two groups are clearly distinguishable there is a crucial concern addressing every software system, that of software quality. As we mentioned above, the groups are defined based on their needs and goals, and even though software quality is a crucial attribute for both, it is important/required for different purposes. In computer science, software quality can directly affect financial profit, while in Open Source Software (OSS) it directly affects directly other related software aspects that are crucial for the open source community. Such attributes[1] are: reliability, stability, scalability and so on, depending on the type of the software system.

[1]Several definitions that describe software quality are available, and we are going to elaborate more later on.

1.2 Motivation - Research question

While discussing software systems and quality attributes, we would like to become a bit more specific in order to demonstrate the reason that motivated us for this piece of work. Firstly, the field that draws our attention is that of science/research, as we want to investigate several situations that occur within the specified domain. Computer science, both practical and theoretical, evolves day by day. Undoubtedly, a great amount of research facilities and universities are successfully contributing in that. Our attention is drawn by situations that often take place in such environments, as several problems can potentially emerge during production.

Scientific approaches, quite often, result into new pieces of software that are being created. That's in order to tackle the specific needs of each (research) topic, or to demonstrate/act as a proof of concept. Usually, such software components or even whole systems follow the *quick and dirty* approach, even sometimes unintentionally. The project resources (man hours, time & budget) are usually limited and therefore proper code quality is sacrificed in order to give its place to other tasks. Although it is something understandable, it is crucial to point out the way it affects the community, to a margin that is not necessarily perceived by it.

The current situation as is being described is the main catalyst for our research question, which is defined as "How can the software quality of scientific software be improved?". In addition to that, we are considering of exploring ways that we can achieve even more, in terms of further putting the produced software to a good use.

Quick and dirty approaches, likely result into poorly designed and developed code in the manner of coding & development standards, as well as close to non-existent documentation[1]. Such situations, deeply affect the project on many levels/phases, as they imply limited time spent on preparation, design documentation, development & testing which would be the cause of underlying issues that influence the overall software quality. As Zhi et. al. (1) studied such influence by focusing on the financial sector of software projects, which concludes that documentation can increase profit, as the final software quality is improved. Similar study by Alla et. al.(2) in health care systems, shows such importance as well. There are vulnerable design decisions[2] that are inherited in the later phases without receiving proper treatment due to the fact that they might not have been visible at the time of creation or due to limited resources. Obviously, such situations have impact

[1]These claims are meant to illustrate the importance of such situations and definitely not to criticise abilities or skills of any developer

[2]Decisions that are prone to errors, bugs, security issues, etc.

on the software's quality, or in other terms, on the project's software attributes. Poor software attributes result in low overall quality, which is the topic we are addressing in this report. Having the resulting project with low code quality and -as mentioned before- probably poor documentation, directly impacts several quantity & quality attributes of the software system/component. This is threatening to the goals we set, as the project's overall quality is something mandatory in order to have a promising beginning 1. Out of the several attributes that are affected, the ones that are crucial to us and are under the spotlight, are maintainability and reusability.

Taking into account the highlighted software quality attributes, we'd like to discuss their impact on research and open source software projects as we aim to increase the chances of the system to be reused, extended or maintained. Of course, this has direct impact on the pace and progress of further research in tightly correlated fields or approaches that may interact with the one held responsible for developing the software under discussion. Furthermore, software created that way, (in many cases) is likely to be archived and forgotten. Therefore, upcoming research on the topic will demand partial (or even complete) re-implementation of the archived software as it, most likely, won't be accessible by other organisations easily.

Academia does not emphasise in software quality outcome as much as industry, and therefore the produced software is not competitive to the global open source software community, in terms popularity and reliability. Our opinion and even our expectation on the topic is to address the issues mentioned above, in order to improve the overall code quality of situations as those described, and hopefully to the point that the community can reach some of the goals that motivate us. Those goals are based on the idea of *"Open source-ing"* software that is being created for the purpose of research, and furthermore aim to satisfy the expected requirements by the *open source software community*. Throughout our thinking process, we acknowledge the rapid software development in industry along with several of their quality standards and we let such factors influence our approach on identifying possible ways to improve open source software quality. Naturally, commercial software is usually more appealing to society than open source software, as more resources are invested into its design which aims to accomplish that result, but also in several aspects of software quality. This aspect is currently not so strong in open source software, which is an aspect that needs improving. To do so, commercial software development methods and procedures can act as a guideline towards that goal (3).

1.2.1 Research Goals

In this section, our goals are being introduced, along with the situations we aim to address. Overall software quality is one issue we address, but at the same time we want to increase the chances of a software system being adopted by the online community. For the latter there are external factors that heavily affect the difficulty of such task, as it depends on users. In order to do so, we investigate the user requirements as summarised and discussed in (4), while considering the *online developer community* as the *end-users* of the software in question. Those levels vary from -1 to level 4 (the higher the level, the more satisfied the user is), which try to capture different aspects of user satisfaction, from *trust* and *cynical satisfaction*, to the level that the software exceeds the user's expectations. The target group of users in such case is the open source software community and to address it we take **GitHub** as an example community, because it's widely known and probably the biggest open source software platform/community at the moment of writing.

1. **Popularity & audience:** GitHub's repositories trust factor is based on the reputation and popularity of the project. That is expressed by domain specific mechanics i.e. amount of people *watching*, marking the repository with a *star*[1], and by taking action on active pull requests (Approve/decline, give feedback for improving the pull request and such). Research projects usually lack popularity and therefore trust, as they tend to implement really specific and possibly complex implementations which are highly correlated with the research field. Therefore, non-researchers are less likely to discover and support such software as it's less likely to fill their needs. Having a limited audience makes it harder to increase popularity and therefore the trust factor of the project. That is one of the most common ways to discard which project to use and which not to, and thus we identify the need to compensate for the difficulty in gaining popularity with another approach. What if there is a way to evaluate or even guarantee the code quality of a given project? That could change things, and earn the trust needed for the project to be included in other work and more importantly, give it a chance in the community. (Addressing user levels -1, 0, 1 (4))

2. **Maintained:** As most of research projects become inactive, once they have been "handed in" or have been used as a *proof of concept*, they are no longer being maintained. That's where the interaction with the community comes in. Having a piece of software created for the purpose of research, it's most likely that there is no other

[1] Referring to Git-Hub's popularity numbers

software designed for the specific problem (or if there is any, it might be unpopular for the reasons described in goal 1). Now, given that the software meets certain quality criteria, it is able to be adopted and maintained for further research, which will hopefully allow it to stand on its own. Taking into account the user levels described in (4), the aim of this goal is to briefly capture the corresponding levels 2, 3 and 4, having their requirements slightly adapted to fit the needs of the specified community in our case.

3. **Further development & evolution:** On top of having the software being maintained, additional components can be added to it through the feedback of the community, or directly by its contribution, in order to broaden the issues the initial software would address. Thus the software can be evolved through such a procedure. (Taking into account that the software stands up to user level 4, it is considered and perceived capable for further development.)

4. **Software Recycling:** Going back to the initial issue, that the projects are more likely to be *"thrown away"* after they have achieved their purpose, we make a metaphor, that of correlating the piece of software to an actual item that has been produced. Now that the item has served its purpose, we can throw it away (archiving the project somewhere where it lies dormant), or preferably put it up for recycling. Of course one could argue that we are addressing reusability, but the difference here is that we are not talking about the *ability* of software to be reused, but for the idea on which developers[1] should build upon.

[1]Currently referring specifically to the ones developing software for research purposes.

1.3 Glossary

Term	Acronym	Definition
Software product	-	Every software system, application or algorithm which is the result of software development.
Open source software	OSS	Publicly accessible and redistributed software.
Closed source software	CSS	Privately developed software, usually from companies that is meant for commercial and/or industrial use.
Object oriented (programming)	OO(P)	-
Software metrics	-	Measurement units defined specifically for software. Can either measure quantity or quality.
Software metric values	-	Result of software measurements. Usually, but not always, take values in the ordinal scale.
Technology readiness level	TRL	Levels that represent to what point a technological product is ready for release/use
Lines of code	LOC	Common software metric for measuring code volume.
Appendix A	-	Research that has been conducted and handed in as a separate deliverable (literature study). Not included in this document version that is destined for review.

2

Software Quality

2.1 What is software quality?

With the rapid growth and evolution of technology, both in software and hardware, users' expectations on quality increase accordingly (5). Improving the state-of-the-art in everyday life equipment and software applications, contributes to such result. As mentioned in (6) §5, no matter how things change as time goes by, striving for high quality will always come in conflict with the time factor of the software development[1]. Software quality has received several definitions so far, and important attempts to capture its essence are described here:

> "Quality comprises all characteristics and significant features of a product or an activity which relate to the satisfying of given requirements".
>
> *German Industry Standard DIN 55350 Part 11*

> "Quality is the totality of features and characteristics of a product or a service that bears on its ability to satisfy the given needs".
>
> *ANSI Standard (ANSI/ASQC A3/1978)*

> "a) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs: for example, conform to specifications.
>
> b) The degree to which software possesses a desired combination of attributes.
>
> c) The degree to which a customer or user perceives that software meets his or her composite expectations.

[1] including all phases of the software's life-cycle, from design/architecture, to testing and delivery.

> d) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer."
>
> *IEEE Standard (IEEE Std 729-1983)*

Although the verbal definitions might be precise in theory, it is needed to identify those individual characteristics that are responsible for capturing software quality. There has been several attempts to achieve that, as the definition needs to adapt to the present day in order to precisely outline the important attributes according to the technological evolution. Some universally accepted definitions are describing such attributes, like ISO 9126-3 as *Functionality, Reliability, Usability, Efficiency, Maintainability, Portability*, R. Fitzpatrick (1996) (7) and the more recent (Gillies 3rd edition 2011) (8) describe it as *Integrity, Reliability, Usability, Accuracy, Efficiency, Maintainability, Testability, Flexibility, Interface facility (Interoperability), Re-usability and Transferability (Portability)* or the ISO 25010 (9) in the same year, as *Functional Suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability.* Adnan et. al.(10) have gathered the most popular software quality models and provide some insight along with discussion on how to choose the most appropriate model depending on the needs of the product.

2.2 Measuring Software

Evaluating to what extent a software product fulfils certain requirements, we are obliged to conduct measurements. That being said, we need to discuss the different types of measurements that are beneficial in individual cases. Firstly, we divide the characteristics (namely attributes) that act as factors that affect quality, between those that externally influence the software product, listed as *External Quality Characteristics: Correctness, Usability, Efficiency, Reliability, Integrity, Adaptability, Accuracy, and Robustness*, and those that can be internally identified as *Internal Quality Characteristics: Maintainability, Flexibility, Portability, Re-usability, Readability, Testability, and Understandability* described by McConnell's (11). Measuring software can be performed by using qualitative or quantitative means or a mix of both in more complex cases. In both situations, for each characteristic in question, there is a set of measurable attributes, called software metrics, that is capable of expressing the associated and correlated characteristics in a software piece/product or system.

2.3 What's in the spotlight?

Padhy et. al.(12) share the same concern regarding software quality of software created both in industry and in academia, while their focus is mainly to improving a specific attribute (reusability), our focus is to propose a methodology. In order to improve the resulting quality of a software system, we need to improve several of its aspects, and thus control them. Continuing on DeMarco's fundamental idea, we identify some of the major quality aspects along with the state-of-the-art in order to measure them. Building on top of software quality's importance, to work our way towards the goals mentioned in 1.2.1, the need for identifying the most relevant and crucial factors arises. Discussing the goals themselves, we see that if we could characterise the resulting software from this procedure, it should be described at least as: maintainable, sustainable and reusable [1]. Additionally, such attributes have already been proven of high importance, as for example maintenance tends to have significant impact at almost all stages (60-80%) of the product's life-cycle (13), and reusability is an important sub-characteristic of maintainability that will be addressed directly.

Another attribute with severe impact on quality is *sustainability*. Sustainability can be viewed from several dimensions, which are: environmental, economic, individual, social and technical (14) (15) (16). The reasons we discussed, lead us to explicitly investigate the aspects mentioned above, *maintainability*, *sustainability* and *reusability*. Maintainability is the most time consuming procedure in the the software's life-cycle(17), and this is tightly coupled/correlated with the defined goals 2. On top of that, as described in 3, we aim to improve the state of the software regarding reusability in order to also increase its longevity. Sustainability on the other hand, apart from the technical aspects, has also a social dimension that is correlated with users' trust, an important goal (goal 1) for us, and that dimension has also been addressed by Becker et. al. (14) and Lago et. al(15).

In this section we are investigating the correlation between maintainability and reusability with software quality, the interaction within the software's life-cycle as well as the possible ways to control and manipulate them. Their definitions within our context and additional details are explained in this document, while some more information is imported directly from Appendix A. Appendix A, titled *How important is the maintainability software quality attribute for software systems, and how can it be improved?*, is the literature

[1] Important to mention here, that as the software under development is being implemented for the needs of study, we expect that it is being correct and tested (both as in software quality attributes). In other terms, is has achieved its purpose for the needs of the study/research

review which was conducted prior to this master thesis report. What additionally motivated us to go with this approach, is a statement that T. DeMarco (18) argued about, back in 1982 and inspired a lot of researchers:

"You cannot control what you cannot measure."

To conduct software measurements, we are making use of the most suitable software metrics in each case. This topic has been under research for quite some time, and many proposals are being made constantly in the effort of precisely capturing the essence of software quality attributes and sub-characteristics. Interesting theories come to practice and they are distinguishable in two main categories, in order to address their main focus separately, as defined by Misra (2005) (17). The first category contains those that propose new metrics, while the second one those that validate existing or their own newly proposed metrics, or investigate the relationships between quality metrics and between quality attributes (and possibly their influence on maintainability) (Appendix A)

2.4 Maintainability

2.4.1 Definition

Maintainability is the software quality metric which expresses how "maintainable" a system is. The higher the value of maintainability, the easier to maintain the system, both in terms of maintenance effort as well as managing maintenance resources.

2.4.2 Measuring & Controlling Maintainability

Several maintainability quality models have been proposed in order to evaluate maintainability of software systems, with the *Maintainability Index* (MI) to be the most popular metric to conduct such measurements. MI has been through refinements and adaptations as the software standards change, until 2009 that Riaz et al. proposed the model that became the most accepted and is the state-of-the-art as of now. It is defined in the equation 2.1.

$$MI = 171 - (5.2\ln(aveVol) - 0.23aveV(g') - 16.2\ln(aveLOC) + 50\sin(\sqrt{2.46perCM})) \tag{2.1}$$

As mentioned earlier, maintainability is a composite metric, and its sub-characteristics are being adapted accordingly, depending on the problem of each case/application. Adapting them to the software's needs, is not always a defined situation and can be under further discussion. For example Al-Kilidar et al. (19)(2005) define them as: changeability, analysability and understandability, while Upadhyay et al. (13) as: changeability, testability and customizability. For the purpose of this research, we use the definition of maintainability as given by ISO 25010 (9): modularity, reusability, analysability, modifiability and testability. In order to provide a clearer picture and show the significant impact regarding maintainability being an important metric that needs attention, suffice it to say that 40-80% of a software product's expenses are spent on maintenance (Glass 2002) (20). Among its sub-characteristics, our attention is especially drawn to two of them: *reusablity* and *analysability*. The reason is that such aspects are important for achieving our goals, as analysability is useful throughout the code analysis in (3.1), while having high reusability increases the chances of software to be adopted by the community.

In the following table we present a summary of maintainability metrics, gathered and discussed by (17) and covered by appendix A.

Predictive metric	Acronym	Definition
Average class size	ACLOC	Average class size in terms of the number of lines per class
Attribute inheritance factor	AIF	The percentage of class attributes that are inherited. It is calculated by summing the inherited attribute for all classes from its super-classes in a project. Average method size AMLOC Average method size in terms of number of lines per method (Lorenz and Kidd, 1994)
Average depth of paths	AVPATHS	The average depth of paths from methods that have paths at all.
Control density	CDENS	Represents the percentage of control statements in the code
Coupling factor	COF	Coupling Factor measures the extent of communication between client and supplier classes (Lorenz and Kidd, 1994). It is measured for the entire project as the fraction of the total possible class coupling. Its value ranges between 0 and 1. Lower values are better than higher ones.
Weighted methods in classes	WMC	Reflects the complexity of the classes and is the sum of the cyclomatic complexities of all methods in the classes (Chidamber and Kemerer, 1994). The WMC metric is obtained by summing the values of McCabe's Cyclomatic Complexity of all local methods.
Depth of inheritance tree	DIT	Depth of inheritance tree (Chidamber and Kemerer, 1991) measures the position of a class in the inheritance tree. It corresponds to the level of a class in the inheritance hierarchy. The root class has DIT value of zero.
Lack of cohesion in methods	LOCM	LOCM metrics calculates the degree of communication between the methods and member variables of a class (Chidamber and Kemerer, 1991). It is obtained by calculating a list of member variables and the number of references to each variable of all methods in that class. Secondly, the sum of the ratios of the usage divided by the total number of methods is obtained. Finally, LOCM is calculated as the quotient of the sum of ratios by the total number of attributes.
Method hiding factor	MHF	Helps to measure the visibility or invisibility of each method with respect to other classes in the project (Brite e Abreau and Carapuca, 1994). The visibility is calculated as follows: private = 1, public = 0, protected = Size of the Inheritance Tree divided by the Number of Classes.

Percentage public/protected members	PPPC	Calculates the percentage of public and protected members of a class with respect to the other members of the class. Response for classes RFC Measures the cardinality of the response set of a class (Chidamber and Kemerer, 1994). Response for class is the number of methods in a class, plus the number of distinct methods called by those methods. Since the principal mode of communication between objects is through message passing, an object can be made to act in a certain way through a particular way of method invocation.
Method inheritance factor	MIF	Obtained by dividing the total number of inherited methods by the total number of methods. The total number of inherited methods, on the other hand, is obtained by summing the number of operations that a class has inherited from its super- classes.
Program length	N	Measures the total number of operators and operands in a program (Halstead, 1997).
Program vocabulary	n	Measures the total number of unique operators and unique operands in a program (Halstead, 1997).
Number of Classes	NCLASS	Calculates the total number of classes in a system.
Number of methods	NMETH	Calculates the total number of methods in a system.
Polymorphism factor	POF	Helps to measure the degree to which classes within a system are polymorphic (Brite e Abreau and Carapuca, 1994). Polymorphism is used in object oriented programming to perform run-time binding to one class among several other classes in the same hierarchy of classes. Polymorphism helps in processing instances of classes according to their data type or class.
Source lines of code	SLOC	Calculates the number of source lines in the project. However, this excludes lines with white-spaces and comments.

Table 2.1: Maintainability Predictive metrics

2.5 Reusability

As explained before, there are several reasons for investigating maintainability, but as it's a composite metric there is the need of investigating even further into an important sub-characteristic, reusability. Not only in the context of software quality but, specifically, in our case, targeting and improving the reusability of software components is crucial as well. Research software, usually is designed to produce really specific outcomes depending on the situation. Throughout this process, many components are being developed in order to be combined and reach the final outcome. Such components, are most likely to be required by other projects, especially those in the same field of study.

2.5.1 Definition

Reusability shows the degree to which an asset can be used in more than one system, or in building other assets (21). It is the quality attribute which questions to what extent different system components can be reused, either within the same or in another system. Reused in the sense that the components are not being *reworked*[1] or *refactored*[2], but they are used exactly as they are.

> "Reusability is usually taken to mean designing a system so that the system's structure or some of its components can be reused again in future applications." (Bass et. al., 1998, p. 84)

2.5.2 Measuring reusability

There is no definite method to measure reusability, although several metrics have been established in order to make an estimation of reusability. Mainly Chidamber and Kemerer have proposed metrics(22) and a metric suite(23) containing six metrics called CK metrics & CK metrics suite. CK metrics have had a significant impact on software engineering research and measurement, becoming fundamental and used broadly. For instance, by Goyal and Gupta (24) and Padhy(12) among others.

Calculating reusability based on metrics has lead to approaches like Goyal's, describing their methodology like in 2.5.2 , while Guptal and Dashore (25) proposed a composite metric expressing the reusability value of a given class, as described in equation 2.5.2.

[1] Rework is a known vicious circle in software development since it plays a central role in the generation of delays, extra costs and diverse risks introduced after software delivery. It eventually triggers a negative impact on the quality of the software developed.

[2] Code refactoring is the process of restructuring existing computer code, without changing its external behaviour. Refactoring improves nonfunctional attributes of the software.

1. First, analyse ck matrix values.
2. Second, calculate reusability of object oriented software model using neural network based SOM technique.
3. Comparing the obtained reusability value using SOM technique with reusability value based on ck matrix analysis.

$$ReusabilityOfClass = a * (DIT) + b * (NOC) - c * (CBO) \tag{2.2}$$

where a, b, c are empirical constants.

For convenience, we take a = b =1 and c = 0.5. Reusability of any OO code is the same as the class having the highest reusability.

Choosing the most suitable reusability model for each case requires taking several aspects into consideration, as there is no *golden rule* yet. For example, there are aspects that can be further improved, like the resulting measurement precision. Additional metrics have been proposed in order to move towards a more complete model, like the one proposed by Padhy et.at. (12) namely *reusability rank*, which aims to increase the precision of such estimations.

2.6 Sustainability

Controlling a software system, requires actions in every step of its life-cycle. All software attributes are meant to be monitored and taken care of in every phase, although some of them need more attention in specific levels, like design or code level. Sustainability is an attribute that is of high importance to be in the spotlight during the architecture/design phase (16). An example we can use the study of Carillo et. al. (26), which defines 7 criteria of industrial AK models and then suggests software quality attibutes for each criterion, in order to express the correlation between sustainability and software quality. Venters et. al. (16) identify two viewpoints of sustainability as it is perceived, *sustainable software* and that of *software engineering for sustainability (SE4S)*. In our case we care about the latter one.

2.6.1 Definition

Before software sustainability can be measured, it must be understood (Seacord et al., 2003 (27)). In modern English, sustainability refers to the 'capacity' of a system 'to endure' (Oxford 2010). The term's Latin origin *sustinere* was used as both endure and as up-hold, furnish [something] with means of support. This suggests that longevity as an expression of time and the ability to maintain are key factors at the heart of understanding sustainability (16).

2.6.2 Measuring & Controlling Sustainability

Sustainability is a multidimensional quality attribute, that is still under research and therefore no state-of-the-art measurements are available. Although, refined and popular approaches try to capture the essence of sustainability, while taking into account the broad spectrum of the attribute's complexity. Quality attributes need to be taken into account from the *design* phase, and sustainability is not an exception. In order to produce sustainable (and maintainable) software, we need to build the system's architecture around that idea. Zdun et. al. (28) have proposed a minimalistic meta model in order to approach such goals. What we would like to highlight here, is the importance of documenting design decisions rationales along with the decisions themselves[1], as it provides transparency to the software, not only for controlling the process of design and development, but for making the (future) understanding of the software easier [2]. Lago et.al. (15) discussed about

[1] Having Zdun et. al. (28) proposing a template in their work.

[2] Both for improved teamwork and for future/new developers as well

the individual substances of sustainability (*social, environmental, technical and economic*) and their interactions/interrelations providing great insight by identifying interdependence relations.

2.7 Quality integration with TRL

At this point we have discussed the software quality of a product, and explained several ways to approach specific qualities of a given software. Reaching our goals 1.2.1 and justifying a software product's quality, requires more than just obtaining *"good"*[1] measurement results. For that reason, we decided to take steps towards more concrete evidence, that can support and represent the software quality of the product. To achieve that, we are investigating and proposing a method for proving the quality of a software product, according to international standards. Initially by identifying and discussing the characteristics of importance to our study in detail, followed by measuring the desired aspects by using software code analysis tools 3.1 which conduct such measurements by using open source code evaluation libraries (8. Appendix B). On top of the code analysis outcome, we are mapping and correlating the obtained results with specific *Technology Readiness Levels* (TRL) (29). TRL is a universal acknowledged rating, that acts as a proof of technological quality as its name states. Additionally, within the context of this research, there is the requirement by the EU commission [2] , that needs to be fulfilled by the research departments of universities as a mandatory specification for receiving the Commission's support and funding. The mapping that is part of our proposed methodology, aims to automate the procedure of extracting a software product's TRL.

[1] Meaning that the outcome of the metrics used, show positive results regarding the factors that are measured.

[2] Links accessed on 1/12/2018:

1) https://enspire.science/trl-scale-horizon-2020-erc-explained/

2) https://ec.europa.eu/research/participants/portal/desktop/en/support/faqs/faq-859.html

3

Software Code Analysis & Evaluation

Measuring software products can be accomplished by adopting several procedures, that vary depending on the types of the measurements. Software metrics are either quantifiable or qualitative, and the procedure of measuring is either manual, semi-automated (supervised) or completely automated. In the case of our research, we are making use of tools in order to conduct the measurements with complete automation.

3.1 Software Quality Analysis

3.1.1 Software Quality Measurements

For this procedure we are making use of software analysis tools, that provide the possibility of fine grained measurements, tailored to project specific needs. We came across several tools that could be used, but we are are going to analyse only 2 (or 3) of them for the means of the present report. The mentioned tools are the ones that we preferred to use for our research and investigation.

- SonarQube https://www.sonarqube.org
- Codacy https://app.codacy.com
- Codefactor https://www.codefactor.io

During our use cases we draw some (totally) subjective opinion regarding the used tools, as we were provided with enough information to comment on several aspects, as well as sharing our comparison information. When investigating software systems, it's common to discuss mostly the technical aspects of the software, using the corresponding terminology. However the following elaboration is meant to consider and address a bigger audience than just the developers and engineers; *"The degree to which a software can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context of use"* (30) (usability) need to be taken into account.

- SonarQube(31):
  - Pros: Lists a nice overview of the projects, along with the most important analysis results. Additionally, the possibility of applying filters in order to provide refined results if you'd like to have a summary of a more specific aspect of the projects. Furthermore, an overview page of the active rules is provided, where you can gain insight regarding all the rules in detail (code patterns) that are available by the system. By combining these rules, the user is able to build quality profiles. Quality profiles are sets of language-specific rules, that are being used for project analysis. Those rules are being grouped into categories, according to the type of metric they represent (cyclomatic complexity (CC), security vulnerabilities etc.). Finally, the outcome of the measurements can be used in a functionality named *Quality Gate*. A quality gate allows you to define metric-specific set of rules, which express whether a project satisfies the quality

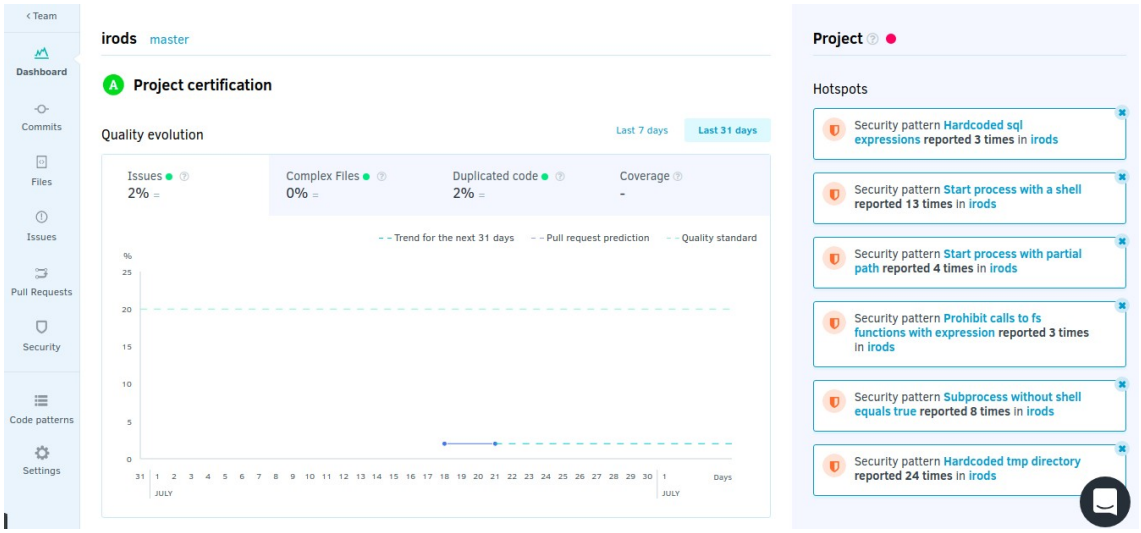

Figure 3.1: Project Overview.

criteria defined for each metric. E.g. we can set CC to be less than 15% and security vulnerabilities between 5% and 15%. Having defined a quality gate for such specific criteria, we can apply it to the projects that interest us in such measurements during their current development stage.

- Cons: The tool requires some time to set up, along with calibrating for the desired results. On top of that, it requires project specific "properties" file, which on one hand is great as you can define the parts of the project to be analysed, but on the other it requires additional manual effort. Another major flaw, is that there is no way to run analysis for more than one language for each project. Lastly, in order to have a project analysed, the compiled build is required to run the analysis, which means that you are forced to deploy the project to the same machine that you run sonarqube in order to produce all the required files.

- Codacy

  - Pros: Requesting project analysis happens online, so there is no need for local setup of the tools, nor project deployment. Provides great project management functionality, by allowing the creation of organisations, inviting members as well as assigning tasks to members, which improves cooperation and coordination in a team. Provides a rich projects' overview (figure 3.1), by drawing graphs

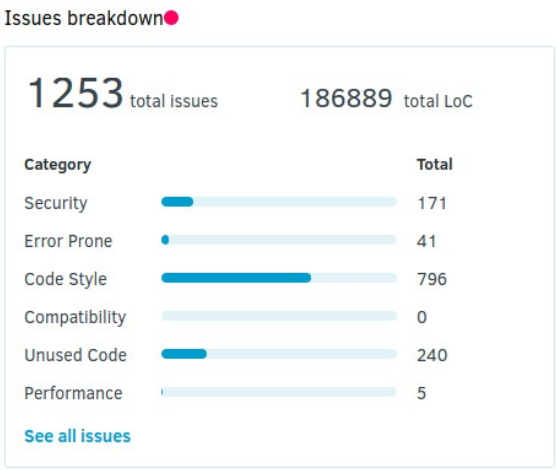

Figure 3.2: Issues breakdown.

and diagrams expressing the summarised information regarding the main categories of the metrics. On project specific level, the overview shows a really good summary of strong and weak points, giving great insight on what needs improvement. Issues overview is simple end self-explanatory (figure 3.2). Filters can be applied to refine the "issues" overview according to "Language", "Category", "Severity level" and "Pattern". Finally, on the "pattern" overview, the project specific patterns that have been applied are shown, along with detailed information about them. They are grouped according to the language and the metric category they are investigating, and one level deeper, containing the list of all the patterns that they include, along with their status (enabled/disabled) (figure 3.3). Patterns can be tweaked in order to decide which of them will be applied to the project.

- Cons: Minimalistic (and kind of empty) interface, which results in a less user friendly environment, since in order to figure out what the buttons and the symbols mean, the user has to either mouse-over or click them through. Settings or a way to change/select the rules to be applied in order to adjust the analysis to each project or preference, was not found. Less in depth analysis probably as there are no comments regarding vulnerabilities or security for example. Also grade generation is kind of black box.

- Codefactor

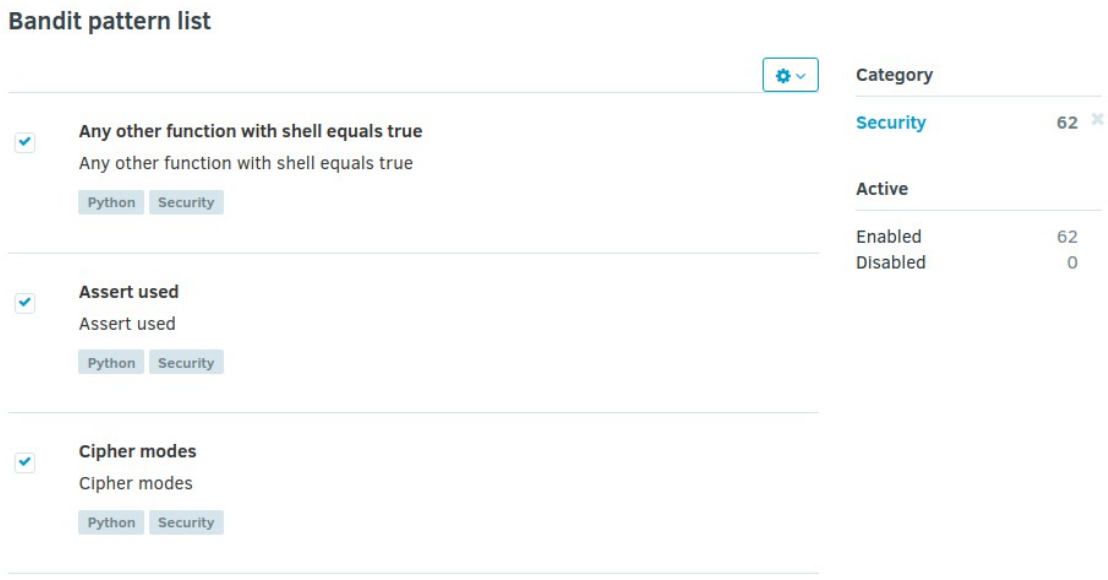

Figure 3.3: Enabled/Disabled patterns

- Pros: Seemingly more accurate grading compared to the two other tool, as it is using "+" and "-" symbols in addition to "A" through "F" grading, to capture a more accurate representative score.
- Cons: Simplistic and kind of "clunky" interface. Projects' overview illustrates most of the important information, but the navigation tabs are not providing a lot of options. This results into slightly hard navigation and sometimes wasted effort as you might be looking for information that is actually absent, like the total commits of a repository.

3.1.2 Grading Breakdown

Sonarqube's list of metrics (32):

- Duplication: Density of duplication = Duplicated lines / Lines * 100
- Maintainability:
  - A=[0-0.05], B=[0.06-0.1], C=[0.11-0.20], D=[0.21-0.5], E=[0.51-1]

    The Maintainability Rating scale can be alternately stated by saying that if the outstanding remediation cost is:

Figure 3.4: Issue categories

  - less than 5% of the time that has already gone into the application, the rating is A
  - between 6 to 10% the rating is a B
  - between 11 to 20% the rating is a C
  - between 21 to 50% the rating is a D
  - anything over 50% is an E

- Reliability:
  - A = 0 Bugs
  - B = at least 1 Minor Bug
  - C = at least 1 Major Bug
  - D = at least 1 Critical Bug
  - E = at least 1 Blocker Bug

- Security:
  - A = 0 Vulnerability
  - B = at least 1 Minor Vulnerability
  - C = at least 1 Major Vulnerability
  - D = at least 1 Critical Vulnerability
  - E = at least 1 Blocker Vulnerability

Codacy(33):

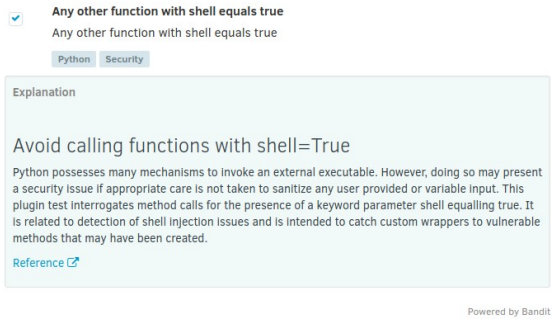

Figure 3.5: Pattern description

- Grades range from A to F, with A being the best grade (34).
- The grades are calculated with the number of issues for each thousand lines of code (KLOC). Steve McConnell, on his book 'Code Complete' (11), wrote about average bugs per LOC and quoting:

  > (a) Industry Average: "about 15 - 50 errors per 1000 lines of delivered code." He further says this is usually representative of code that has some level of structured programming behind it, but probably includes a mix of coding techniques.

Codefactor(35):

- Codefactor uses a modified metric that used letter indicators combined with 1-10 numeric range (36):

By investigating the grading criteria of each individual analysis tool, we make comments on the strong and weak points as seen throughout our use cases. Sonarqube is pretty well documented and appears to have a legitimate evaluation process. Codacy is using criteria from the book (11), which offers confidence in their grading system. Codefactor, provides the list of open source analysis libraries used for reviewing, and expresses the results in a scale A-F, with A being the highest/best score.

Table 3.1: Codefactor grading scale

Performance	Grade (letter)	Numerical range
Excellent	A	9.4-10
Excellent	A-	9.0-9.3
Good	B+	8.7-8.9
Good	B	8.3-8.6
Good	B-	8.0-8.2
Satisfactory	C+	7.7-7.9
Satisfactory	C	7.3-7.6
Satisfactory	C-	7.0-7.2
Poor	D+	6.7-6.9
Poor	D	6.3-6.6
Poor	D-	6.0-6.2
Failing	F	0.0-5.9

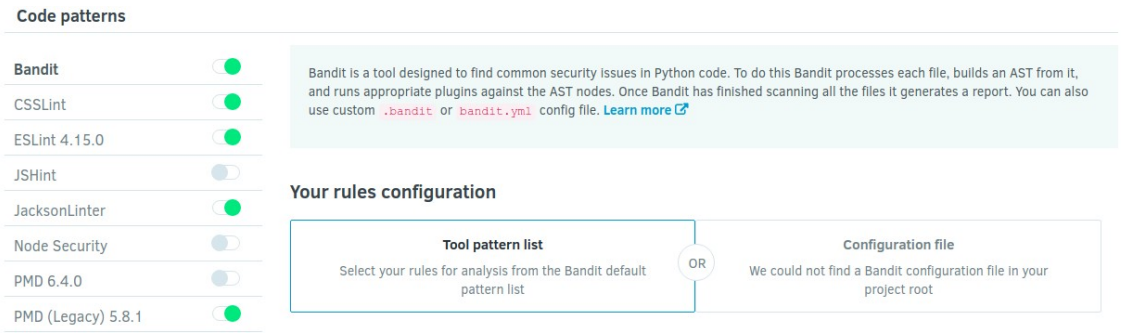

Figure 3.6: Analysis patterns.

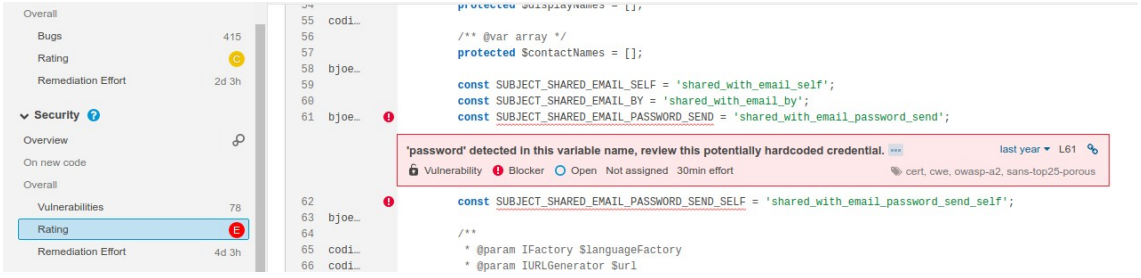

Figure 3.7: Faulty code explanation.

4

Technology Readiness Level

4.1 Introduction to TRL

Discussing the importance of software quality for every software system, it becomes essential to be able to draw some representative conclusions or to obtain insight regarding a software system. In this report we would like to take a look into the ways that one can evaluate software systems and determine their quality, minimising the required resources to do so, while allowing one to judge whether the software is ready to be exposed to the market/industry (Closed Source Software / CSS) or be published to the community (OSS).

For that purpose, we'd like to present/include the Technology Readiness Levels (TRL) in our discussion. TRL is a methodology introduced by NASA in 1974, which was not formally defined until 1989. The idea behind the TRL is to evaluate technology maturity and was initially designed for NASA aeronautics containing 7 technology readiness levels, in order to determine whether a piece of technology that is being developed, is ready to be integrated with an existing system. In addition to NASA's aeronautics were TRLs were used to determine the technology used for space missions, several governmental organisations also adopted the scale and tailored it to their specific needs, such as the Department of Defense (DoD). Later on, two additional levels were added in order to adapt the TRLs to the commercial market.

Of course TRL, originally, is meant to express technology maturity, which in other terms implies hardware development as well. Up to this day, TRLs are being applied to more than just hardware with the proper adjustment for each party using the scale. Additionally, there have been many proposals of related scales that are derivatives of TRL application to specific needs researches. Some of them are:

- Integration Readiness Levels (IRL)

- System Readiness Levels (SRL)
- Research and Development Degree of Difficulty (RD3)
- Manufacturing Readiness Levels (MRL)

While we reference some of the TRL derivatives, some of which could be alternatives that can be chosen to support this piece of research in similar approaches, it is important to highlight the reason behind staying on the "original/base" TRL methodology. That is due to the criteria set by the European Commission, for university research funding. As the commission states, the need of proof of quality expressed in TRL is required for such qualification, and we were already investigating (semi-) automated ways of assessing software code quality, we decided to dive into possible ways of automating such an evaluation process.

4.2 Application of TRL on Software

Taking into account that the TRL is mostly meant for hardware, one could argue that if that is the case, then why do we want to apply it on software systems, and how could it be helpful. As it was initially designed for evaluating complete (including hardware or only hardware) systems[1], it is true that it might not give as much insight in order to draw conclusions if applied on software. On the other hand, the need for obtaining such valuable information regarding software systems exists, as discussed in the previous section, and therefore if we can adjust and apply it on such system, some insight is much better than no insight at all.

Therefore, several proposals have been made in order to interpret the original TRL to Software TRL. Hereby are the adapted definitions for software, which are numbered according to the corresponding TRL (level) as found on NASA's website[2]:

Level	Description
1	Scientific knowledge generated underpinning basic properties of software architecture and mathematical formulation.
2	Practical application is identified but is speculative, no experimental proof or detailed analysis is available to support the conjecture. Basic properties of algorithms, representations and concepts defined. Basic principles coded. Experiments performed with synthetic data.
3	Development of limited functionality to validate critical properties and predictions using non-integrated software components.
4	Key, functionally critical, software components are integrated, and functionally validated, to establish interoperability and begin architecture development. Relevant Environments defined and performance in this environment predicted.
5	End-to-end software elements implemented and interfaced with existing systems/simulations conforming to target environment. End-to-end software system, tested in relevant environment, meeting predicted performance. Operational environment performance predicted. Prototype implementations developed.
6	Prototype implementations of the software demonstrated on full-scale realistic problems. Partially integrate with existing hardware/software systems. Limited documentation available. Engineering feasibility fully demonstrated.
7	Prototype software exists having all key functionality available for demonstration and test. Well integrated with operational hardware/software systems demonstrating operational feasibility. Most software bugs removed. Limited documentation available.

[1] https://goo.gl/4o6JYC (Last accessed on 27/12/2018)
[2] https://goo.gl/XykPfq (Last accessed on 27/12/2018)

8	All software has been thoroughly debugged and fully integrated with all operational hardware and software systems. All user documentation, training documentation, and maintenance documentation completed. All functionality successfully demonstrated in simulated operational scenarios. Verification and Validation (V&V) completed.
9	All software has been thoroughly debugged and fully integrated with all operational hardware/software systems. All documentation has been completed. Sustaining software engineering support is in place. System has been successfully operated in the operational environment.

1. Scientific knowledge generated underpinning basic properties of software architecture and mathematical formulation.

2. Practical application is identified but is speculative, no experimental proof or detailed analysis is available to support the conjecture. Basic properties of algorithms, representations and concepts defined. Basic principles coded. Experiments performed with synthetic data.

3. Development of limited functionality to validate critical properties and predictions using non-integrated software components.

4. Key, functionally critical, software components are integrated, and functionally validated, to establish interoperability and begin architecture development. Relevant Environments defined and performance in this environment predicted.

5. End-to-end software elements implemented and interfaced with existing systems/simulations conforming to target environment. End-to-end software system, tested in relevant environment, meeting predicted performance. Operational environment performance predicted. Prototype implementations developed.

6. Prototype implementations of the software demonstrated on full-scale realistic problems. Partially integrate with existing hardware/software systems. Limited documentation available. Engineering feasibility fully demonstrated.

7. Prototype software exists having all key functionality available for demonstration and test. Well integrated with operational hardware/software systems demonstrating operational feasibility. Most software bugs removed. Limited documentation available.

8. All software has been thoroughly debugged and fully integrated with all operational hardware and software systems. All user documentation, training documentation, and maintenance documentation completed. All functionality successfully demonstrated in simulated operational scenarios. Verification and Validation (V&V) completed.

9. All software has been thoroughly debugged and fully integrated with all operational hardware/software systems. All documentation has been completed. Sustaining software engineering support is in place. System has been successfully operated in the operational environment.

By examining the definitions above, we see that it is more than viable to evaluate a software system's TRL by investigating which points are being fulfilled. Although the application of the above is a bit abstract, since there is no defined way of correlating a system to the TRL, except by filling in a questionnaire or more like a spreadsheet (37).

4.3 Software TRL calculation idea and proposal

As it is probably clear by now, being able to determine the TRL of a software system can be really useful. It provides insight about the status of the system and its development stage, and it can potentially be used in order to simplify the communication between technical and non-technical parties of an organisation or a third party. Also it can act as proof of the system's quality when it's in production (CSS) or published to the community (OSS). Although for the latter, we can argue about how dependable taking the TRL of a software product as proof of quality is, since it is calculated with a spreadsheet? That means that a (group of) person(s) is responsible for filling in the accurate response to the spreadsheet, and therefore we can't determine for sure whether the outcome of the calculation is true and valid, as the means by which it was being calculated are not dependable (i.e. the person making the calculation could have false insight of the system, or be biased). Of course there are several publications that propose calculations (equations), but usually they are too specific trying to address a given type of software.

As the current procedure of calculating the TRL of software systems has several flaws, it cannot be used as the dependable proof we described, and so we identify the need of calculating the TRL in such a way that can be valid, have minimised flaws and also act as a proof of the software quality of the system under examination. In order to do so, the procedure that is going to be used, needs to fulfil the same requirements on its turn.

In order to tackle the described problem, we argue that if actual metric measurements are involved to the TRL calculation, it would better support the validity of the outcome. Software attribute measurements can be a difficult but valuable task, as it provides great insight of several aspects of the software system. At this point, we would like to discuss the proposed approach for calculating the TRL. It is really important to highlight that such procedure is, most likely impossible to be automated, considering that until now it is being done by the use of spreadsheets. Additionally, the proposal also gives space for fine-grained adjustment, precisely as it is desired for [1]. First comes the software quality measurement, which is complex and important as it is the basis for the idea. Once we are able to actually make measurements, we need to be able to understand what do they mean about software quality (maturity) as it is expressed by the TRL scale, and therefore we need to approximately interpret and correlate the TRL levels with the software quality measurements, in order to be able to extract the information required, and even more, argue about the achieved TRL.

[1] Different types of systems require different measurements and also same metrics can have different impact depending on the type as well

TRL	Measurement Interpretation
Level 4 Level 5	• coding standards • style • compatibility
Level 6 Level 7	Solving/Reducing amount of bugs & error prone issues.
Level 8 Level 9	Exterminating remaining issues, striving towards higher or perfect grades.

Table 4.2: TRL to Measurements Correlation

4.4 TRL Correspondence

The software code analysis tools are able to interact with systems having TRL 4 and above (since only then, actual software code is being produced). Mapping of the metrics that give insight in the system to specific TRL happens on an abstract level, as the TRL model needs to be slightly adapted to the project's specific needs. The correlation is illustrated in table 4.4, and each higher TRL includes all the previous requirement as well.

Each of the systems described in section 3.1, are using slightly different ways of delivering grades to software code, but one thing is common among all, that of the quality borderline. In each case, a way of representing the *Grade "A" Quality* is defined, and this is our focus. In order to consider that the criteria in table 4.4 are met, achieving the highest grade available is required, regardless of the tool chosen for the evaluation.

The higher the level, the more attention is needed on the severity of the issues. Errors should be kept as low as possible at all times, but of course, if not possible, they need to be solved in order to reach level 7-8 since at that point we are reaching the final stages of the system's development and "production"[1] deployment. It is worth mentioning that the level 8 was initially the highest, and 9 was added later on, so we consider it a refinement level.

[1] We are referring to production environment in order to describe the significance of the desired software quality and the attention needed towards details.

4.5 Importance of correlation with TRL

Being able to understand and express the TRL in more technological terms, provides several benefits which are tightly relevant to a software system's development. Of course, as mentioned before, being able to gain insight regarding a system is crucial as the system's health can be monitored at any given snapshot in time. Such information allows us to intervene immediately and adjust decisions that have been made, in order to provide suitable solutions that will address any concerning issue, before it is too late[1]. Additionally, we gain awareness of possible vulnerabilities and error prone issues that can be avoided in the later stages, by taking the analysis' results into account when making design decisions and plans about future development.

Furthermore, not necessarily beneficial to the development procedure of a single project, keeping proper documentation and records/log regarding several projects that might be implemented by different teams, can be proven useful. Doing so, aids in identifying patterns or pitfalls during development that have negative impact on the system, and therefore can be dealt with in advance or even be avoided.

Up until now we only discussed about using the proposal suggested by this report, with the purpose of "monitoring" and "controlling" a software system's development, by analysing the system itself. However, apart from gaining insight regarding the system under discussion, it is crucial to mention that the same correlation can be extracted for a candidate third party module (or service) to be included in the system. Pointing the discussion towards handling plugins that are being involved with the system, we can point out several important benefits that can be earned by applying the same process on them. Keep in mind, that TRL is meant to be used this way, judging whether a piece of technology is mature enough to be introduced to the system under development. As the external libraries are being analysed, and while taking into account that for each system we can set our own desired qualities to be matched from the analysis, we can apply the same rules to them. Therefore, the outcome of the analysis makes it possible to apply a TRL on each individual module. For instance, using only external libraries that receive a level 7 or above, it means that it's up to the development team to retain level 7 or higher. Last but not least, TRL examine the capability of a module to be introduced to the system, also in terms of compatibility (TRL 4), which can be automated directly by using one of the tools in this proposal.

[1]Meaning the project is further in development, which will create heavier dependencies on the existing decisions.

A problem that arises with the current TRL calculation method, is that it is not really trustworthy. That's a result of the current flawed process of evaluation. As of now, the TRL evaluation of a project, as we described before, requires to fill in an *.xls* spreadsheet. To make this statement clear, we provide the following example:

A project manager fills in the spreadsheet with project information, in order to receive the corresponding TRL. Once the software is "evaluated", he wants to distribute the developed piece of software to the open source/research community. In order to promote it, the manager is making the claim that the provided software is a TRL 7, so that the community can trust that the system is going to deliver everything it's required to, within a research environment. The problem arises, with the realisation that the awarded TRL may involve human error[1]. Humans tend to be biased, especially when they want to support their own creations in terms of quality and meaningfulness. Combining this with the previous paragraphs of this section, it is possible to explore even more benefits that are generated as an outcome by this summarised information. We mentioned that we can apply analysis to any candidate module that is going to be included, which -apparently- gives us a good overview of the quality of the specific piece of software. We can test if it satisfies our self-defined qualifications and decide whether it's suitable or not. Taking this one step further, we can get rid of human error in the TRL calculation by applying the proposed analysis. We can argue that our software system is of a specific TRL, while pattern configuration details/files act as proof, by distributing them to any other party that would like to confirm or make use of it. It is worth mentioning, that while this method provides transparency of the software system and includes every detail to support reproducibility and repeatability, in this kind of situation, the awarded TRL is bound to a specific set of pattern configurations and qualifications.

[1]Or just human judgement, which in this case we consider it having the same impact.

5

Guideline

5.1 Introduction

Improving the resulting software quality is what we seek to answer of our research question and is quite an important task, and of course, a continuous process. It is affected by all development phases, during the whole product's life-cycle. We argue that in order to control quality we need to control the crucial components that influence it, and we do that by conducting measurements that provide us insight and transparency. Although, while important, the measurements are not sufficient on their own to achieve such goal. It is necessary to pay attention to other significant aspects, like documentation, architecture and design. Such topics though cannot be covered completely just by this research, as specifications and details are necessary. We are constructing the basis towards a software implementation that will support much more than only our goals 1.2.1. Concluding with everything discussed till this point, and taking into account the aim of this research, we'd like to develop a guideline that will be used as a support handbook for the purposes mentioned so far. We find it quite important to assist developers in the research community, by providing insight on improving their software quality. In this section, we summarise the information gathered by this research, in order to compose and propose a *guideline*, that is meant to accompany academic software developers throughout the implementation of their systems. That will be done by discussing several steps of the implementation, while providing references for elaborated material and greater insight. It is crucial to point out that given the available resources for each project, as well as the level of implementation difficulty, they should be divided accordingly and this isn't something that can be predefined. In academia is always the case that many and complicated systems have to be implemented in small amount of time, therefore proper time management is crucial

as some additional tasks are being involved with our proposal. There is no silver lining on such approach, but the recommendations made in this chapter are meant to support and instruct the developers with important aspects of software engineering throughout the process.

5.2 The Guideline

The procedure being described here, is based on a generic software life-cycle model, in the form of: *Requirementanalysis* → *Design* → *Implementation* → *Testing* → *Maintenance/Evolution*. To adapt the model to our research, we need an extra phase (which is repeated before finishing almost every step in the software's development cycle, as explained in this section), which is the *Code Analysis* phase, that has to be executed during several steps, and the resulting model should look like this (Figure 5.1): *Requirement Analysis* → *Design* → *Implementation* → *Code Analysis/Evaluation* → *Testing* → *Code Analysis/Evaluation* → *Maintenance/Evolution* → *Code Analysis/Evaluation*. During the requirement analysis, it is important to identify the aims of the software, not only in terms of output, but also in desired quality aspects [1]. Identifying such aspects in this phase, is important as they are crucial for later stages.

Another issue that is worth addressing, is the importance of documentation [2]. Design phase is the one containing most of the design decisions that are made throughout the project. At this point, we need to mention the importance of properly documenting[3] design decisions, for the reasons mentioned in 2.6.

By injecting the *Code Analysis / Evaluation* phase to the life-cycle, we aim to continuously monitor the software's status regarding the desired attributes identified in *Requirement analysis* phase, and of course take action when needed[4]. That way, we are able to have continuous integration in our methodology and guideline process. It is important to try and keep the evaluation results high enough, especially in the early phases, as we want to contain/keep (change) entropy (39) of the software for as much as possible, as a "snowball effect" is the least needed within our context. Sadly, entropy, is not only generated by the code that is developed throughout the process, but it is highly affected

[1] Depending on the software's purpose, being reliable might be more important than being secure and so on

[2] It is quite common for software developers to reduce the amount of documentation they provide under time pressure (both internal and external documentation) (38)

[3] Both internally and externally when needed.

[4] Discussing in detail on how to tackle such situations is not one of the objectives of this document.

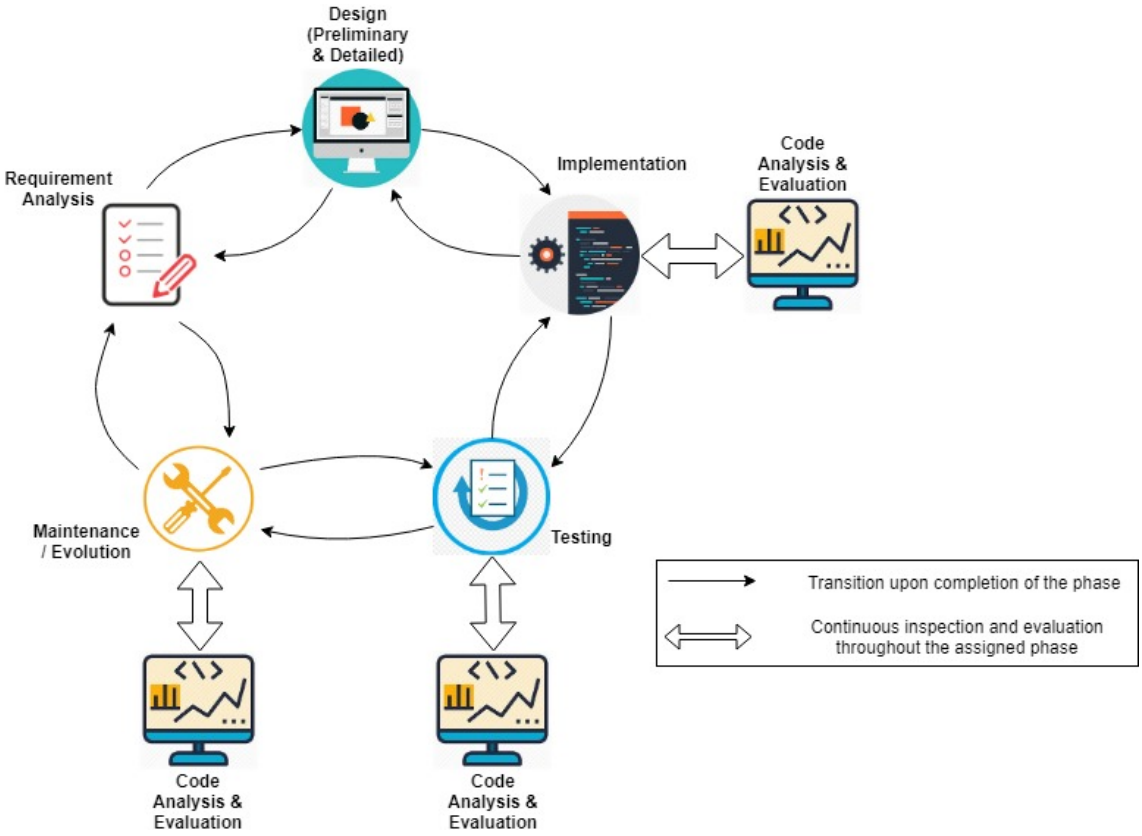

Figure 5.1: Suggested development cycle

by the abstraction level of the libraries that are used as well. Dealing with such case is hard, as more attention is needed regarding the selection of appropriate libraries, with high code abstraction (low entropy), and it has been discussed by Capra et.al (2012) (40). Although their concern is regarding energy efficiency, the means of their approach can be adapted and used here as well. They proposed an indicator for measuring the entropy level of code fragments (that can be used for filtering external libraries as well) which they call *Framework entropy*.

Is good to mention that we are making use of the TRL correlation in order to gain trust for the product, while working our way towards satisfying the goals (1.2.1) set in the introduction section. As discussed, being able to correlate a software system to a universal accepted scale, is something that supports the trust in the system as well as its overall quality.

Keeping the code evaluation results high, means that we work towards the final grading criteria, which is to achieve the desired TRL for the developed product. Although, as noted in 4.4, sometimes not all measurements are important for each phase in the life-cycle, we

should highlight that paying attention to possible side effects of the ongoing development might affect other measurements. Of course, such task is not easily achievable due to the complex relations and interactions between the quality attributes. There have been several studies like Haggander et. all(41)[1], Aldekoa et. al.(42), ISO/IEC 25000:2014(9), regarding the impact of quality attributes to each other, but in the end the trade-offs are always project-specific. A nice summary and case study has been made by Berander et.al. (2005) (43).

At the project's conclusion, there is a little additional information that needs to be accompany the software itself. A list with the adapted *filters/code patterns* that were used[2], the final scores as well as the table 4.4 for reference. The gathered information can act as a solid proof of the software's TRL at the moment, being supported by the present research.

[1]Maintainability and performance relation.

[2]Those that differ from the default, while making that explicit.

6

Analysis & Results

6.1 Experimentation

For the purpose of this research, a total of 10 projects were selected and analysed in order to extract information and data in order to investigate our approach further. For the test cases we selected open source software projects, of different sizes and purposes. All of the test cases that were chosen, had to be open source as that aspect is the main focus we are addressing. They range from 1000 lines of code (LOC) to a bit less than 450.000 LOC, while some of them are open source and free to use, some offer explicit services online while using the same code base, while others have been converted to private for further development and commercial use. Different sizes and purposes allow us to obtain a broader set of observations that can be used for further testing in order to improve software quality aspects.

6.2 Test results

In this section we list all the measurements obtained by the analysis, split according to the tool used. The measurements provided here are representing the overall issues that each code analysis tool identified. The first thing to note is the difference in the amount of projects analysed by SonarQube. The reason behind that is discussed in the next section as it was due to some limitations that could not be avoided and finding a proper workaround was really time consuming.

Table 6.1: Codacy results

Project	Total issues identified	Grade
Cookery	59	B
Pumpkin	508	C
Weevilscout	4276	B
Sonarqube	19960	B
Lobcder	3302	B
OwnCloud	42416	B
OneData	485	B
NextCloud	6699	A
dCache	7825	B
iRods	1253	A

Observing the results of each tool, there are several factors that need discussion; the assigned grade, and the total identified issues on project level. Firstly, we discuss about outstanding differences in the identified issues of the projects. We observe that Codacy (Table 6.1) identifies way many more issues that Codefactor (Table 6.2) on Sonarqube, Lobcder and dCache, while their grades are quite similar. Investigating the reason behind that, we find that Codacy's increased issue report number, lies in *warning* severity level, that does not really impact the final grade[1]. Now, regarding the difference in grade, *Pumpkin* and *OwnCloud* received and F from Codefactor, while they received a C and a B respectively from Codacy. Investigating further, we discover that such difference in the grades, lies in the severity and impact of the identified issues. Each analyser uses different evaluation methods, which explains the differences in grades as well. Particularly, both tools identify a significant amount of security vulnerabilities (26 by Codefactor and 68 by Codacy), but the severity of the issues vary. All of the Codefactor's security issues are

[1]For example in *Lobcder* is 2232, while in codefactor just 252.

Table 6.2: Codefactor results

Project	Total issues identified	Grade
Cookery	31	C+
Pumpkin	710	F
Weevilscout	3580	D-
Sonarqube	1524	A
Lobcder	945	B-
OwnCloud	56556	F
OneData	1076	C
NextCloud	4415	B
dCache	2900	B
iRods	3252	C-

Table 6.3: Sonarqube results

Project	Total issues identified	Grade
Cookery	42	Passed
Pumpkin	1366	Failed
Lobcder	19218	Passed
OwnCloud	8669	Passed

evaluated as *major* level of significant, on the contrary, Codacy marks them as *middle* and low severity. However, there is another factor that we need to take into account, and that is of the analysis tools' (technological) ongoing & further development. The tools themselves are software systems that continue to be developed and improved, and of course such changes affect several aspects, having the grading be one of them. During our research we kept monitoring all of the projects above along with the grading differences due to each new *commit* to their repository. This observation is important, as we kept monitoring and verifying the provided grades in this paper even at the time of writing, until we noticed this interesting fact.

The snapshot of the results provided here, is from December 2018, where all the project grades had small to no changes in their grade. Of course, there were several changes on the issues identified, as in number or nature of new issue, while some old issues were resolved, the grade remained approximately the same. That was until a point that during the validation of the grades, we noticed that Codefactor changed OwnCloud's grade from F to C while Codacy's remained exactly the same. That makes us draw two conclusions that

influence the discussion on this section. Firstly, that due to Codefactor's development (or of one of its dependencies - grading library) the severity of issues can change and therefore affect the overall result and grade. Secondly, that it is not wise to have an in-depth analysis or comparison of the provided grades at the moment, as, based on the earlier conclusion, the automated analysis process is not so concrete yet[1].

[1]Automated code analysis tools were introduced recently, around 2010. Therefore, they have to establish a solid ground for their techniques as well as catch up with further programming languages development.

6.3 Limitations

Throughout the experimentation with the software code analysis tools, we came across several difficulties that are worth mentioning to provide further insight on the obstacles that might occur.

The few obstacles we faced were when experimenting with sonarqube. At the analysis part, evaluating with sonarqube can be proven a bit more complicated. Apart from putting together the configuration file, the input for analysis, heavily depends on the project's makefile as well. This can indeed cause frustration in case the makefile is outdated or in similar situations, taking into account that anyone might want to analyse the project, without the aid of a developer. Therefore, producing the compiled project to be analysed, might require additional -not wanted- effort. Furthermore, once everything is set for analysis, we have to deal with the limitations of sonarqube itself. Each tool differentiates the services it provides according to several aspects. In case of sonarqube, that is different packages, that include additional tool/language regarding the analysis and some extra services (which of course come with pricing). The obstacle arose as C++ is not supported in Sonarqube's free package, and it was required in order to analyse several of our projects. Upon contacting Sonarqube regarding this issue, they suggested to use the "Developer package", which includes several additional languages such as C++, but comes with a price we were not willing to pay. Codacy and codefactor on the other hand, differentiate their service based on the privacy setting of a project's repository, which is free for the public repositories and priced for private ones.

Another issue that we should discuss, is that of validating the proposed correlation between the code analysis grades 4.4 and the TRL scale. The reason behind this, is that we lack time & expertise[1] resources, which is explained in "Future work" 7.1.

[1] Specifically regarding the use cases.

7

Conclusion & Discussion

We had a deep look into measuring software and its importance on software quality. Quite a few information have been based upon one of the earliest approaches to software measurement by DeMarco (1982) (18), therefore there are a few things we would like to draw the reader's attention to. Firstly, we need to consider that when DeMarco published his thoughts, the topic of software measurement was still a new one, with many aspects to explore and many changes through time as well. Working our way towards the purpose of this research, we propose a methodology that is capable of answering the research question that we've set. That is, by altering the software's development cycle to improve the scientific software quality. Although, this approach should act as a guideline and not as absolute truth. If we account for his book as a whole, one can derive that the more the metrics discovered and used, the better the resulting software quality, which is indeed not true. Both the control and the development of a product require adequate resource investment, and we need to clarify that there has to be a limit to how focused someone is on measurements and control, in order to keep them in balance with the software product itself. Additional thoughts and self reflection of DeMarco regarding his book can be found in (44). As he said and we quote:

> "... the more you focus on control, the more likely you're working on a project that's striving to deliver something of relatively minor value. To my mind, the question that's much more important than how to control a software project is, why on earth are we doing so many projects that deliver such marginal value?"
>
> "Can I really be saying that it's OK to run projects without control or with relatively little control? Almost. I'm suggesting, first we need to select projects where precise control won't matter so much. Then we need to reduce our

> expectations for exactly how much we're going to be able to control them, no matter how assiduously we apply ourselves to control."

Therefore we would like to explicitly express/remind that this document's purpose is to act as a guideline and not to be followed universally regardless the case that it is being applied to.

7.1 Future Work

First thing in line, is to validate the suggested correspondence of the analysis grading, to the TRL scale[1]. The planned experiment for such validation is to pick several projects as use cases, and for each use case, a person with good insight is required. The projects' *"experts"* are called to calculate the TRL for their use case, by using the official TRL excel calculator (37). We draw conclusions on the TRL of each project, based on the received grades of the code analysis while keeping into account the proposed correspondence, and compare it to the TRLs received from the excel sheet calculator. Once the comparisons are finished, we need to discuss and draw conclusions regarding expected differences, provide rationales and proceed to adjusting the correspondence where needed.

Later on, additional software quality attributes need to be evaluated, analysed and included in the procedure we discussed in 4.3. From those identified in (9) and (8), our candidates are: *functionality, reliability and correctness.*

[1]The reason why this is not included in the present study, is explained in the limitations6.3

Bibliography

[1] Junji Zhi, Vahid Garousi-YusifoÄ§lu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. **Cost, benefits and quality of software development documentation: A systematic mapping**. Journal of Systems and Software, **99***:175 – 198, 2015. 3*

[2] Sujatha Alla, Pilar Pazos, and Rolando Delaguila. **The impact of requirements management documentation on software project outcomes in health care**. *05 2017. 3*

[3] B. Tratz-Ryan. **Sustainability Innovation Key Initiative Overview**. *https://www.gartner.com/doc/2516916/sustainability-innovation-key-initiative-overview, 6 2013. 4*

[4] **The profession of IT: Software Quality**. *Technical Report 25, COMMUNICATIONS OF THE ACM, 9 2016. 5, 6*

[5] Salesforce. **Salesforce chapter: User Expectations**. *https://www.salesforce.com/research/customer-expectations/. [Online; last accessed 2-February-2019]. 9*

[6] Gerard OâĂŹRegan. Introduction to Software Quality. *Springer, Cham, 2014. 9*

[7] Ronan Fitzpatrick. **Software Quality:Definitions and Strategic Issues**. *4 1996. 10*

[8] Alan Gillies. *3 edition, 2011. 10, 51*

[9] **ISO 25000 Software Product Quality**. *https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, 2011. [Online; last accessed 18-October-2018]. 10, 13, 42, 51*

[10] Bassem Matalkah Adnan Rawashdeh. **A New Software Quality Model for Evaluating COTS Components**. *2006. 10*

BIBLIOGRAPHY

[11] Steve McConnell. Code Complete. *1993. 10, 27*

[12] Suresh Chandra Satapathy Neelamdhab Padhy, R.P. Singh. **Software reusability metrics estimation: Algorithms, models and optimization techniques**. *2017. 11, 16, 17*

[13] Nitin Upadhyay, Bharat M. Deshpande, and Vishnu P. Agarwal. **Developing maintainability index of a software component: a digraph and matrix approach**. ACM SIGSOFT Software Engineering Notes, **35***:1–11, 2010. 11, 13*

[14] C. Becker, R. Chitchyan, L. Duboc, S. Easterbrook, B. Penzenstadler, N. Seyff, and C. C. Venters. **Sustainability Design and Software: The Karlskrona Manifesto**. *In* 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, **2**, *pages 467–476, May 2015. 11*

[15] Patricia Lago, Sedef Akinli Koçak, Ivica Crnkovic, and Birgit Penzenstadler. **Framing Sustainability As a Property of Software Quality**. Commun. ACM, **58***(10):70–78, September 2015. 11, 18*

[16] Colin Venters, Rafael Capilla, Stefanie Betz, Birgit Penzenstadler, Tom Crick, Steve Crouch, Elisa Yumi Nakagawa, Christoph Becker, and Carlos Carrillo. **Software Sustainability: Research and Practice from a Software Architecture Viewpoint**. Journal of Systems and Software, **138***:174–188, April 2018. 11, 18*

[17] Subhas Chandra Misra. **Modeling Design/Coding Factors That Drive Maintainability of Software Systems**. Software Quality Journal, **13***(3):297–320, Sep 2005. 11, 12, 13*

[18] Tom DeMarco. **Controlling Software Projects: Management, Measurement and Estimation**. Jourdon Press, *1982. 12, 49*

[19] K. Cox H. Al-Kilidar and B. Kitchenham. The use and usefulness ofthe ISO/IEC 9126 quality standard. *11 2005. 13*

[20] Robert L. Glass. **Facts and Fallacies of Software Engineering**. *2002. 13*

[21] **Software Product Quality**. *Standard, International Organization for Standardization, March 2011. 16*

[22] Shyam R. Chidamber and Chris F. Kemerer. **Towards a Metrics Suite for Object Oriented Design**. SIGPLAN Not., **26***(11):197–211, November 1991. 16*

[23] Shyam R. Chidamber and Chris F. Kemerer. **A metrics suite for object oriented design**. *1994. 16*

[24] Er. Deepali Gupta Neha Goyal. **Reusability Calculation of Object Oriented Software Model by Analyzing CK Metric**. *2014. 16*

[25] Dr. Pankaj Dashore2 Amit Gupta1. **An Approach to Analyse Software Reusability of Object Oriented Code**. *2017. 16*

[26] Carlos Carrillo, Rafael Capilla, Olaf Zimmermann, and Uwe Zdun. **Guidelines and Metrics for Configurable and Sustainable Architectural Knowledge Modelling**. *In* Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 63:1–63:5, New York, NY, USA, 2015. ACM. 18*

[27] Robert Seacord, J Elm, W Goethert, Grace Lewis, Daniel Plakosh, J Robert, L Wrage, and M Lindvall. **Measuring Software Sustainability**, *10 2003. 18*

[28] U. Zdun, R. Capilla, H. Tran, and O. Zimmermann. **Sustainable Architectural Design Decisions**. IEEE Software*,* **30***(6):46–53, Nov 2013. 18*

[29] **NASA Technology Readiness Level**. *https://www.nasa.gov/directorates/heo/scan/engineering/technology/txt_accordion1.html , 2012. [Online; last accessed 23-October-2018]. 20*

[30] ISO. **ISO 9241-11:1998 Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability**. *Technical report, International Organization for Standardization, 1998. 22*

[31] Sonarsource. **Sonarqube**. *https://www.sonarqube.org . [Online; last accessed 10-December-2018]. 22*

[32] Sonarsource. **Sonarqube metrics**. *https://docs.sonarqube.org/display/SONAR/Metric+Definitions . [Online; last accessed 9-November-2018]. 25*

[33] Codacy. **Codacy**. *https://www.codacy.com/ . [Online; last accessed 10-December-2018]. 26*

[34] **Codacy**. *https://support.codacy.com/hc/en-us/articles/207994765-What-are-the-different-Grades-and-how-are-they-calculated- . [Online; last accessed 9-November-2018]. 27*

[35] Codefactor. **Codefactor**. *https://www.codefactor.io/. [Online; last accessed 10-December-2018]. 27*

[36] Codefactor.io. **Codefactor**. *https://support.codefactor.io/i14-glossary. [Online; last accessed 9-November-2018]. 27*

[37] NASA. **Trl calculation worksheet**. *https://esto.nasa.gov/files/TRL_Worksheet_11-30-10.xls. [Online; last accessed 2-December-2018]. 33, 51*

[38] Rajib Mall. Fundamentals of software engineering. *PHI Learning Pvt. Ltd., 5 edition, 9 2018. 40*

[39] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. **How changes affect software entropy: an empirical study**. Empirical Software Engineering, **19***(1):1–38, Feb 2014. 40*

[40] Eugenio Capra, Chiara Francalanci, and Sandra A. Slaughter. **Is software âĂIJgreenâĂİ? Application development environments and energy efficiency in open source applications**. Information and Software Technology, **54***(1):60 – 71, 2012. 41*

[41] D. Haggander, P. Bengtsson, J. Bosch, and L. Lundberg. **Maintainability myth causes performance problems in SMP application**. *In* Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99) (Cat. No.PR00509), *pages 516–519, Dec 1999. 42*

[42] Gentzane Aldekoa, Salvador Trujillo, Goiuria Sagardui, Oscar DÃŋaz, Mondragon Unibertsitatea, G. Aldekoa, S. Trujillo, G. Sagardui, and O. DÃŋaz. **Experience measuring maintainability in software product lines**. *In* In JISBD, *pages 243–247, 2006. 42*

[43] Patrik Berander, Lars-Ola Damm, Jeanette Eriksson, Tony Gorschek, Kennet Henningsson, Per JÃűnsson, Simon KÃĕgstrÃűm, Drazen Milicic, Frans MÃĕrtensson, Kari RÃűnkkÃű, Piotr Tomaszewski, and Lars Lundberg. **Software quality attributes and trade-offs**. *01 2005. 42*

[44] **Software Engineering: An Idea Whose Time Has Come and Gone?**, *8 2009. 49*

[45] Wei Li and Sallie M. Henry. **Object-oriented metrics that predict maintainability.** Journal of Systems and Software, **23***(2):111–122, 1993.*

[46] Yuming Zhou and Baowen Xu. **Predicting the maintainability of open source software using design metrics**. Wuhan University Journal of Natural Sciences, **13***(1):14–20, Feb 2008.*

[47] Kulwant Kaur and Hardeep Singh. **Determination of Maintainability Index for Object Oriented Systems**. SIGSOFT Softw. Eng. Notes, **36***(2):1–6, May 2011.*

[48] Xiaogang Jian, Shuaibo Cai, and Qianfeng Chen. **A study on the evaluation of product maintainability based on the life cycle theory**. Journal of Cleaner Production, **141***(C):481–491, 2017.*

[49] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. **Standardized code quality benchmarking for improving software maintainability**. Software Quality Journal, **20***(2):287–307, Jun 2012.*

[50] Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomou. **Open Source Software Development Should Strive for Even Greater Code Maintainability**. Commun. ACM, **47***(10):83–87, October 2004.*

[51] Wei Zhang, Liguo Huang, Vincent Ng, and Jidong Ge. **SMPLearner: Learning to Predict Software Maintainability**. Automated Software Engg., **22***(1):111–141, March 2015.*

[52] Juliana de A.G. Saraiva, Micael S. de França, Sérgio C.B. Soares, Fernando J.C.L. Filho, and Renata M.C.R. de Souza. **Classifying Metrics for Assessing Object-Oriented Software Maintainability**. J. Syst. Softw., **103***(C):85–101, May 2015.*

[53] Mahmudul Huq Werner Janjic Danail Hristov, Oliver Hummel. **Structuring Software Reusability Metrics for Component-Based Software Development**. IARIA, *2012.*

[54] Young Lee and Kai H. Chang. **Reusability and Maintainability Metrics for Object-oriented Software**. *In* Proceedings of the 38th Annual on Southeast Regional Conference, *ACM-SE 38, pages 88–94, New York, NY, USA, 2000. ACM.*

[55] Technology Readiness Assessment Guide. *U.S. G.A.O.*

[56] Jr. Stephen Blanchette, Cecilia Albert, and Suzanne Garcia-Miller. **Beyond Technology Readiness Levels for Software: U.S. Army Workshop Report**. *Technical Report CMU/SEI-2010-TR-044, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.*

[57] P. Morrow, F. G. Wilkie, and I. R. Mcchesney. **Function Point Analysis Using NESMA: Simplifying the Sizing Without Simplifying the Size**. Software Quality Journal, **22***(4):611–660, December 2014.*

[58] **Characterizing the contribution of quality requirements to software sustainability**.

[59] Coral Calero, Maria Ángeles Moraga, and Manuel F. Bertoa. **Towards a Software Product Sustainability Model**. CoRR, **abs/1309.1640**, *2013.*

[60] **NASA Demonstratable Technology**. *https://www.nasa.gov/directorates/heo/scan/engineering/technology/Accordion2_text.html , 2014. [Online; last accessed 23-October-2018].*

[61] Shyam R. Chidamber and Chris F. Kemerer. **Towards a Metrics Suite for Object Oriented Design**. *In* Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications, *OOPSLA '91, pages 197–211, New York, NY, USA, 1991. ACM.*

[62] Brown J. Kaspar H. Lipow M. McLeod G. Boehm, B. and M. Merritt. **Characteristics of Software Quality**. *1978.*

[63] M. Howard and S. Lipner. **The Security Development Lifecycle SDL**. *2006.*

[64] **ISO 9126 on software quality.** *1993 and updated in 2001 and 2011.*

8

Appendix B

8.1 Analysis libraries

8.1.1 Codefactor

Language	*Name*	*Configuration*
C#	*StyleCop*	*Settings.StyleCop file*
C++	*CppLint*	*CPPLINT.CFG file*
CSS	*CSSLint*	*.csslintrc file SCSS*
Stylelint	*.stylelintrc* file*	
Less	*Stylelint*	*.stylelintrc* file*
SugarSS	*Stylelint*	*.stylelintrc* file*
JavaScript	*ESLint*	*.eslintrc.* file*
TypeScript	*TSLint*	*tslint.* file*
CoffeeScript	*CoffeeLint*	*coffeelint.json file*
Swift	*SwiftLint*	*.swiftlint.yml file*
Ruby	*RuboCop*	*.rubocop.yml file*
Go	*Govet*	*N/A*
Go	*Gocyclo*	*N/A*
Python	*Radon*	*N/A*
Python	*Pylint*	*.pylintrc file*
Python	*Bandit*	*.bandit file*
Java	*Checkstyle*	*checkstyle.xml file*
Scala	*Scalastyle*	*scalastyle.xml file*
Groovy	*CodeNarc*	*codenarc.xml file*

PHP	*PHP_CodeSniffer*	*phpcs.xml(.dist) file*
Bash	*ShellCheck*	*.shellcheck.yaml file*
Dockerfile	*Hadolint*	*.hadolint.yaml file*
YAML	*Yamllint*	*.yamllint file*

8.1.2 Codacy

Codacy works with code patterns, which is a long list and can be found and modified under this url https://app.codacy.com/account/patterns. Each account can specify the desired default patterns on account level, or the configuration file on project level.