

**Автономная некоммерческая организация высшего образования
«Университет Иннополис»**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»**

**GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)**

**Field of Study
09.03.01 – «Computer Science»**

**Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»**

**Тема /
Topic**

**Качество программного обеспечения: удобство
использования и оценка показателей программного
обеспечения / Software Quality: Usability and Evaluation of
Software Metrics**

**Работу выполнил /
Thesis is executed by**

**Чавез Родригез Роберто
Энрике / Chavez Rodriguez
Roberto Enrique**

подпись / signature

**Руководитель
выпускной
квалификационной
работы /
Supervisor of
Graduation Thesis**

**Маццара Мануэль /
Mazzara Manuel**

подпись / signature

**Консультанты /
Consultants**

**Комолов Сирожиддин
Зайнитдин Угли /
Komolov Sirojiddin**

подпись / signature

Contents

1	Introduction	7
1.1	Research Question	8
1.2	Software Metrics in Software Engineering	8
1.2.1	Product Metrics	9
1.2.2	Process Metrics	10
1.2.3	People Metrics	11
1.3	The ISO Standards for Software Quality	12
2	Literature Review	14
2.1	Works on Software Metrics	14
2.2	Works on Product Metrics	15
2.3	Programming Languages	16
2.4	Code Evaluation	17
2.4.1	Use of Metrics for code evaluation	17
3	Methodology	19
3.1	Selection of Project for the Analysis	21
3.1.1	Backend Description	22
3.1.2	Frontend description	25
3.2	Selection of Metrics	27

3.2.1	Lines of Code	28
3.2.2	Number of attributes and methods	29
3.2.3	Cyclomatic complexity	30
3.2.4	Message passing coupling	32
4	Implementation	34
4.1	Calculation of LOC	34
4.1.1	LOC measurement in the front-end	35
4.1.2	LOC measurement in the back-end	36
4.2	Calculation of <i>SIZE2</i>	38
4.2.1	Obtainment of <i>SIZE2</i> in the front-end	38
4.2.2	Obtainment of <i>SIZE2</i> in the back-end	43
4.3	Calculation of CC	46
4.4	Calculation of MPC	49
4.4.1	MPC measurement in the front-end	49
4.4.2	MPC measurement in the back-end	50
5	Evaluation and Discussion	53
5.1	LOC results evaluation	53
5.2	<i>SIZE2</i> results evaluation	54
5.3	CC results evaluation	54
5.4	MPC results evaluation	55
6	Conclusion	56
6.1	Final thoughts and future work	57
	Bibliography cited	58

List of Figures

- 3.1 Illustration of a general backend structure. Source: www.upwork.com. 23
- 3.2 Illustration of a general frontend structure. Source: www.upwork.com. 25
- 3.3 User interface. Data fields for a new user registration 26
- 3.4 System log for loan status 27
- 3.5 Definition of the LOC metric according to the ISO 9126. 28
- 3.6 Definition of the *SIZE2* metric according to the ISO 9126. 30
- 3.7 Definition of the CC metric according to the ISO 9126. 31
- 3.8 Definition of the MPC metric according to the ISO 9126. 32

- 4.1 Graph of LOC for the *Components* Module 35
- 4.2 LOC measurement for the *Dashboard.jsx* component of the Components module 36
- 4.3 LOC for the frontend of the project. Ignored Files are the files that have no lines of code in them - src: zero sized file, src/components: zero sized file, src/util: zero sized file 36
- 4.4 LOC for the back-end of the project. 37
- 4.5 Output of the file *file1.txt*, explaining the reasons why some files were ignored when doing the back-end LOC calculation. 37

4.6 CC calculation for the whole project with COCOMO cost calculations. 48

4.7 CC calculation for the front-end including COCOMO cost calculations. 48

4.8 CC calculation for the back-end including COCOMO cost calculations. 49

Abstract

Having multiple software metrics to choose from can represent an issue when it comes to selecting an specific set of metrics to apply in a program. Currently, there exist different standards that state how these metrics are used and what measures they perform in written code; however, many of them address aspects of quality that are not primarily relevant to a program, and on the other hand, when there is a feature of a program that needs to be evaluated, several metrics can be used but some with less or more precision than others. To provide a guide to understand under what circumstances a certain metric of software can be used, an analysis of the most commonly used metrics is performed in a software which was written with a set of programming languages. These metrics are applied in the code of the program, and after the analysis, a few set of conclusions is performed, to lead the reader towards a series of thoughts that will help them make a better choice when it comes to selecting the most appropriate software metrics to assess the quality of a software program. Although the present work evaluates software with specific characteristics at a fairly general level, it will certainly be useful for professionals in the field of Information Technologies and Computer Science.

Chapter 1

Introduction

Since people started to code computer programs, there existed several different ways to measure the quality of a given software. Formerly, programming languages were rusty and very tedious to work with. The first languages implied deep connection with the way in which the hardware of a computer worked, and knowledge about this field was strictly necessary. These languages are called low-level languages, and they were used to interact directly with the processor of a computer. Then there exist middle-level languages, languages that contain a certain level of abstraction; and finally, there are high-level languages, where the level of abstraction is superior and they do not have a straightforward connection with the processor instructions of a computer.

This is why, when it comes to measuring the quality of software, there exist several ways to do it. Precisely because of being so many programming languages, there are different standards and different software quality metrics that can be used, and this sometimes creates a difficulty for the person in charge to evaluate quality of code. This is the reason why this research is executed, in order to alleviate the load of having to choose among so many features

and giving the possibility to the reader to implement only the most relevant ones according to their goals. For this project, the Software Quality ISO 9126 Standards are used as a reference to choose the quality software metrics and apply them in the program that this research analyzes. After the analysis, some conclusions will be made and hopefully, the reader will be able to get a helpful guide that will collaborate with his or her decisions.

1.1 Research Question

The goal of this project is to bring the reader an analysis of different metrics applied in a software project for the reader to have a general guide on how to select metrics according to the project the reader works on. Therefore the research question arises:

- *What metrics could be more suitable to apply when measuring the quality of code of a software product?*

With this question, the attempted objective of the present work is to provide the reader with a general, yet comprehensible guideline on how to analyze what metrics could best be used when taking quality measurements in a software product.

1.2 Software Metrics in Software Engineering

As mentioned already, there exist several metrics that evaluate different aspects of quality and production of a software. Likewise, there are different angles of measure of a software product that is released. Mainly, there are two

main sections of a software product that metrics evaluate in Software Engineering: *Product Metrics* and *Process Metrics* [1].

1.2.1 Product Metrics

Product metrics of a software program are related to the analysis of software development at any of its phases. The measures can be done either at the requirement assessment phase or even to already built systems. Product metrics are related to the features of software, and they are subdivided in two parts, *Dynamic metrics* and *Static metrics* [2].

Dynamic Product Metrics

These metrics assist with the evaluation of how efficient and reliable a program can be. They are in general highly closely connected to the quality properties of software. It is fairly feasible to quantify the time used for a task to run and to start a system. Dynamic metrics work to check for system efficacy and failure types, and the failure area can be directly connected to how trustworthy a software product is.

Static Product Metrics

Static metrics are gathered by analysis of a system in terms of its programs, design, or also the measures can be performed in the system's documentation. Static metrics in general give support with the comprehensibility and complexity maintenance of a software product. These metrics indirectly related to the quality properties of a program. Several of these metrics were designed to understand the and validate the connection among maintainability,

complexity and level of understanding of software. Metrics such as the length of a program, and the complexity that it has among others.

1.2.2 Process Metrics

These metrics evaluate the process in which the development of software and its parts is executed. For instance, a common metric of this type is commonly related to process time measurement of software parts.

Starting from the point that product quality is intrinsically linked to its creation process, process metrics have the goal of monitoring, estimating and enhancing the quality and level of reliability of software products. In fact, there exist standards created to establish benchmarks and references to results of measures taken when evaluating software products. The main standard is the ISO-9000 referencing the "Standards of Management Quality", which is a series of guidelines delivered by the International Standard Organization - ISO [3].

Quite frequently process metrics are used as operational tools that can help improve and control the management of how software is created. In its nature, software is not a tangible object, but an abstract product that consists of several other intangible parts. Therefore, it is at a certain level difficult to keep track of the progress of software development with an objective set of measurements. Generally management of a software product focuses on administrating the productivity level of software, its progress and time it takes to be developed. The objective of management is to improve these areas and foresee results about them.

Process metrics pertain to models of elaboration of software. A highly known model in this field is the Boehm's COCOMO, which stands for Construc-

tive Cost Model. This model consists of performing cost evaluations concerning software. Another common model is the Thebaut's COPMO, which basically predicts how much extra work will be necessary to finalize big projects.

It is worth to mention that even though metrics of this type are usually considered important for software management, they do not aim at comprehending in depth how a program is built. These metrics address more the areas of prediction and management of resources, time of development and software utilization.

1.2.3 People Metrics

Metrics in reference to the performance of people in the development of IT products are also important in terms of software quality management and analysis. Another name for such metrics is *personnel metrics*. Several writers reference the analysis of human performance using this term.

Personnel metrics make quantifiable different attributes that have to do with software generation in professionals. They help analyze properties like productivity, presence in work, and other rates such as time required to build code [4].

In fact, the primary objective of applying such kind of metrics is to help the human resources stay at ease and furthermore, concentrated on the assignments they are given. Although these metrics were not applied in the present work, it is still worthwhile to mention them briefly. Below their categorization:

1. Metrics related to the working experience:

- Experience in programming languages.
- Experience in designing and writing methods.

- Experience in software management.

2. Metrics concerning the level of productivity:

- Level of productivity.
- Statistics of productivity.
- Quality in productivity.

3. Metrics to asses communication level:

- Experience in team working.
- Ability of communicate between software and hardware.
- Personal time that can be invested in the project.

4. Metrics to check for team structure:

- Metrics of hierarchy.
- Metrics to perform stability in conformed teams.

All these metrics are essential when evaluating the performance of development of a software product. They help determine the resources distribution amongst workers developing the software. Also these metrics very much help understand in what stages of the project more time and effort should be spent.

1.3 The ISO Standards for Software Quality

The standard applicable to the measurement of software quality is the ISO/IEC 9126, which depicts a model of quality that sets the quality of software into six different areas that are also split into its corresponding features.

These features are shown upon usage of software made by its internal software properties [5].

These internal properties are assessed with the help of inner metrics. For example, an inner metric could be the control of software development before it is released. There are different internal metrics that can be found in the subsection of the norm ISO 9126-3. As to quality features, these are evaluated with the implementation of outer metrics; for example, the assessment of software products that will be taken into the market. Again, different examples of these external metrics are in the subsection ISO 9126-3 of the norm.

For this work, internal quality, which can be measured in the source code, was taken for the evaluation. Now, the quality model contained in the section ISO 9126-1 is briefly explained in general terms.

1. **Functionality:** It checks how well functions work when software is written.
2. **Reliability:** The ability of a program to work without failure under a certain period of time.
3. **Usability:** An aspect to analyze how much of use can a program have once it is written and released.
4. **Efficiency:** It evaluates the ability of software to provide resources whenever they are needed.
5. **Maintainability:** This area focuses and how much work must be put on in order to perform editions and modifications to an existing software.
6. **Portability:** It checks for the level of adaptation of a software when it is intended to be transferred from one environment to another.

Chapter 2

Literature Review

As in every existing piece of work, there usually is an initial stage of background on which a research can be built, and this research is by no means an exception. The field of Software Metrics includes an extensive set of tools that measure the quality of software and several sources of information have developed knowledge in them. For this current research, a number of relevant sources will be cited that will help with the development of this work.

2.1 Works on Software Metrics

Different software metrics have been established for the measurement of code consistency and quality. They have to do a lot with how and why the code is implemented. In fact, these metrics present different nuances when they evaluate different scenarios; moreover, when different software metrics tools are used, they evaluate software differently with different approaches [6].

But this does not seem to be negative, as a matter of fact, the world of software engineering is so big that a few set of tools would not be enough to cover the evaluation of different types of software. That is why software metrics,

as already mentioned, is an extensive field in software engineering with lots of implementations that help evaluate software and predict its attributes, factor that is very important when assessing software quality [7].

There exist a considerable number of software metrics that evaluates different aspects of software development, for example, lines of code evaluates how big a program is and how long it can take for this program to compile and run. On the other hand, cyclomatic complexity determines how structured and encapsulated the architecture of a program is [8]. Again, different metrics evaluate different aspects of software products, and understanding their functioning is essential for the research process of this work.

2.2 Works on Product Metrics

Product metrics is another area of interest for this research because it is a superset of software metrics that evaluate different aspects of software products, including software requirements and level of product interaction with its end user [9].

Whereas software metrics mostly focus their attention on the source code of a program, product metrics bring a higher understanding of what the source code of a program is intended for, and therefore, understanding how it is possible to evaluate a product in a higher level will be a great complementary tool to perform the task of this research, which is to evaluate software metrics on a program and determine which of those metrics can be the most appropriate to apply.

However, it is at times difficult to pick what product metrics will be used for program assessment, and that is why it is primarily important to under-

stand the purpose of a software product and its functional and non-functional requirements [10].

2.3 Programming Languages

In computer science, there exist several types of programming languages available for the programmer for the execution of a program. These languages basically follow different structures and architectures that differentiate them one from the other. Two of the main programming types are functional programming (FP) and object oriented programming (OOP) [11]. In functional programming, the written code is subdivided in subsystems of functions that perform different types of operations, and these operations can be used inside other functions to carry out other operations. This type of programming is highly useful when designing user interfaces, for example, in web applications [12]. On the other hand, OOP focuses more on the formation of objects and classes that encloses a bigger abstract representation of a model [13]. For example, a model that needs to be set in code could be a house, where chair would be an object of type furniture, being furniture a class and along with other classes, they would form the model of a house. This kind of programming is much more seen in backend coding, where languages like Java are utilized.

There are many programming languages that work under these paradigms; therefore, it must be of key importance to understand the type of language that is being analyzed in order to grasp what software metrics might be the best to apply. In general terms, cyclomatic complexity would work for an OOP solution, whereas lines of code could be a more suitable metric to apply in a functional programming language, such as ReactJS.

2.4 Code Evaluation

As to the evaluation of code, standards to indicate software quality and to evaluate software metrics have also been established. A very well known standard for measuring quality of software is the ISO/IEC 9126, a standard that measures software quality by utilizing a set of software metrics and values that indicate how well a program is written and how structured it is [14].

And it is also primordial to get clear knowledge in the standards that currently exist for software quality, because this research aims at providing a comprehensible guide towards selecting appropriate metrics when analyzing the source code of a program. Hence, this software quality standard will be used to perform the software metrics analysis of the source code.

2.4.1 Use of Metrics for code evaluation

Several works and papers have been written to classify a certain software metric into a more precise set of characteristics that they describe. One example of such works can be related to the counting of lines of code in the source code. One interesting sub classification can be found in the paper “A SLOC Counting Standard” written by Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm [15].

In this paper, the authors talk about lines of code (abbreviated as LOC) mentioning that they are one of the most widely utilized metrics in the industry as well as in pieces of literature. LOC is the main measurement when calculating models like the COCOMO, the SLIM and the SEER-SEM¹. Even so other standardization organizations like the IEEE have elaborated other definitions

¹These are estimation models that check for the cost of software

for the calculation of LOC, there are still exist lots of questions as to how properly LOC should be counted to have a proper result and valid information. The problem of particularly measuring LOC in different ways triggers in generating issues when finding the cost of software; therefore, the estimations of cost may vary and even worse, they might differ galore. To alleviate this unfortunate predicament, some clear and widely used standard should be used and more importantly, the calculation of LOC and other metrics in general could be more efficiently done by the implementation of other software tools that can actually facilitate finding results and generating similar measurements rather than calculating metrics by hand.

When checking the quality of a program, its size (how big the source code is) should be taken into account as one of the primary factors to understand and find software estimations in terms of software cost. In fact, LOC is not the only measurement that can be evaluated to find the cost of software, but it definitely affect how other metrics will be calculated, and what other metrics will also be implemented to analyze software quality and determine its cost.

Chapter 3

Methodology

The present work aims at giving an understanding of what metrics should be applied among the vast set of them to a specific software project that has been successfully built and has practical use. The idea of the methodology is to first identify a good project that contains different programming languages with different programming paradigms in order to cover a good range of code writing in the field of computer science. Then after selecting the project, the next step is in fact the metrics selection and implementation process. Below, the following list of steps is given in order to clearly identify the series of steps that will be taken for the execution of the research.

1. **Project selection:** The first step, the choice of the project in which the software metrics will be applied. As expressed already, this should be a project that is written in a couple of programming languages in order to see how different metrics are applied to the code written in these languages. Other considerations could be that it would be good for the project to be written in such a way that has in it the common general structure of a program, i.e., a program that has a backend development

and a frontend development. This way a more complete program will be evaluated. In contrast, however, the project should not be excessively large so that more time in understanding the project and trying to run it (either locally or in a server) is used instead of using the time to apply the metrics and draw conclusions from its results. Also, the project should have practical use; namely, it should be a project that could be used by users and gives them some utility rather than just being a demo project that has no major purpose in its creation. Thereby, the measures will be more realistically done and the results and conclusions will be more veracious. The chosen project and explanation to its features and characteristics can be found in the Section 3.1.

2. **Metrics selection:** In this step the idea is to cover some aspects of the quality factors that were briefly described in the Section 1.3. Different metrics were taken into consideration when doing the research. In fact, for the selection it was considered the type of language that was used for the coding, the design paradigms, and the most commonly used metrics that are usually implemented when evaluating the quality of a software product¹. Based on this information, some metrics were selected to be applied in the already selected project to later on elaborate conclusions of the results.
3. **Metrics implementation:** In this part, the implementation of the metrics was executed. For this step, mostly only computer tools were utilized in order to carry out the measurements and calculations of the code in the project. As it was already mentioned in the Section 2.4.1, it was

¹These metrics are considered to be the most commonly found in literature.

just better to use existing tools to obtain the measurements rather than utilizing sheer hand to make the calculations. This action helped with the speed of calculations and with the human error when implementing calculations and measurements by hand. The tools used to perform the measurements and their description are stated in the Section 4.1.

4. **Results and conclusions:** This is the final step, and in this section the user guide for software metrics selection was written. However, it must be clarified that the guide was not written as a series of steps for choosing the metrics that were going to be used in a project. Rather, the guide is a series of conclusions that lead the reader of this work towards a series of thoughts that might help him or her to decide among the vast range of metrics and choose the most appropriate ones for further implementation. The conclusions are obtained from the results found in the previous step, and the development of this section can be found in the Chapter 6 of this work.

3.1 Selection of Project for the Analysis

First of all, it must be taken into consideration on what kind of project the research will be done. The selected project has to do with a machine learning² system that can decide whether or not a given client can take a monetary loan from a bank based on his current status and loans history. The system uses various biometric data that the client fills out in a given form to decide about the loan. The source code and further explanations on how this system works

²A comprehensible guide on machine learning: https://www.sas.com/en_us/insights/analytics/machine-learning.html

are given in the next subsections of this section. It is worth to mention that the source code of the chosen system consists of a frontend and a backend sides like most web applications do; hence, the project is quite complete in terms of software design.

The source code of the project can be found under the following link: <https://github.com/gfredtech/loan-project>

The instructions for running the project locally are given in the repository containing the project. It should be mentioned that the project currently runs a server online; thus, it is not necessary to run the server (backend) locally for the project to run, only the frontend is necessary.

3.1.1 Backend Description

First of all, it should be clear what backend development of a web site or application means. Backend or back-end is the inside development of a given software system. It works mainly with the storing of data in databases, and the architecture of the software product. In this area there are hidden task that are activated when there exist some action in a website. In this area, the developer writes code in order to help a browser to better establish communication with the information which is contained in a database [16]. Figure 3.1 shows how the backend of a software system works.

Functioning of the project backend

The backend is primarily written in Python, it uses the Flask web framework developed precisely in Python. Flask is basically a framework to write backend web applications that does not require any extra libraries or external tools for its functioning. It does not contain any abstractions for database nor

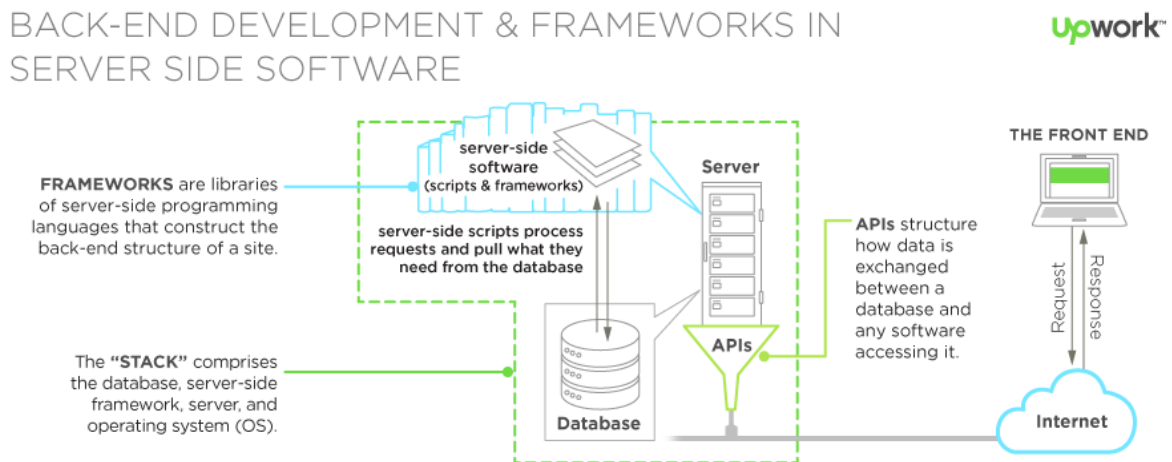


Fig. 3.1. Illustration of a general backend structure. Source: www.upwork.com.

for any other different features which would commonly require the use of other tools. Nevertheless, Flask has different extensions that can contribute with other features just as if these features were written in Flask. Such extensions are useful for various tasks such as form validation and authentication features [17].

The flask backend has routes for:

1. Login
2. Signing up new admins
3. Retrieve admin details by id or email
4. Update the admin details

For the entries:

1. Get all loan application entries
2. Add a new entry

Finally, the backend also works with error handling routes, which means that when a new loan application entry is added, the entered data is run through a machine learning algorithm to decide whether the loan application should be granted to the user or not.

As to the machine learning functioning of the backend, it works with a dataset. The dataset used comes from kaggle, which is a popular data science competition platform with different tools and resources for data scientists. The dataset contains the loan id, the gender of the applicant, also if the applicant is married or not.

Steps of the machine learning algorithm:

1. Fill missing values: some of the rows might be missing data, so we fill some of them with default values if no values are found there. For example, if there is no gender at a given field for a given profile, the gap will be filled out with the gender as male, which is the default value. This feature can be seen in the file *ml/train.py* of the source code.
2. After that, the data is split into a 1/3 split, which means that for every 1 test data, there exist 3 training data.
3. The data is then encoded into better values using the scikit-learn Label Encoder class.
4. Then feature scaling on the data is performed, using the Standard Scaler class also from scikit-learn.
5. Afterwards, the training of the model happens by using Logistic Regression. Logistic regression is a type of predictive analysis performed on data

where the dependent variable is binary. This type of regression is utilized to depict data and clarify the relation between a binary variable and one or many other nominal or interval variables that are independent.

- Eventually, the model is saved to a file so that it can be used again later on. When someone enters their data, these data will be run through the saved model.

3.1.2 Frontend description

Front-end development in general is the process that is done in order to connect the internal layers of a software system with the interface that interacts with the client that uses such system. It is the method of using technologies like HTML, CSS and mostly JavaScript for the creation of a website. The objective of this development is for the clients to see and use relevant information in a friendly and interactive way. Figure 3.2 shows generally how the structure of a frontend area of a software product works.

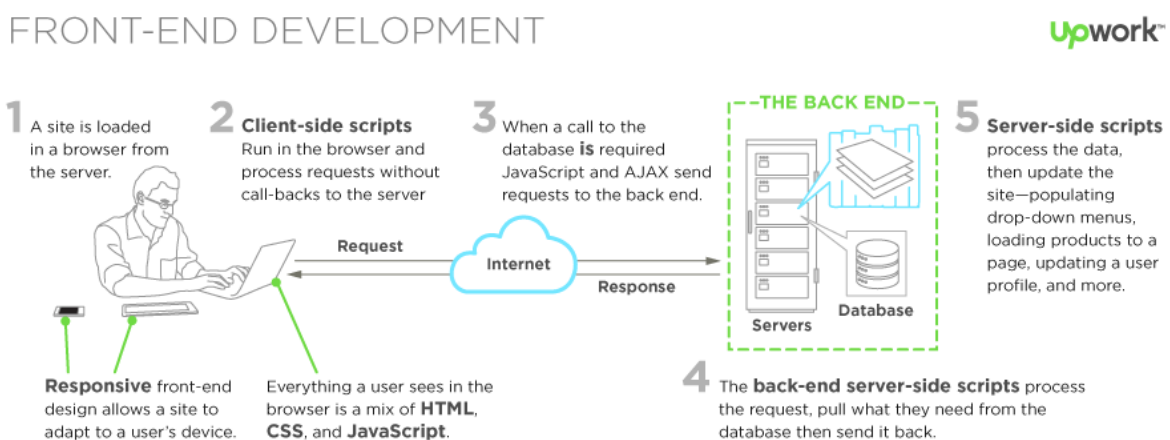


Fig. 3.2. Illustration of a general frontend structure. Source: www.upwork.com.

In the project, the frontend is written with ReactJS, a frontend development framework with the purpose of designing and creating user interfaces. This is done by means of user interface (UI) components, each component representing an integral part of the web application. A component is code that can be reused and has a similar purpose of that of the functions in JavaScript, with the difference that components work independently and mostly return HTML[18]. Figure 3.3 shows a portion of the UI of the application when a new user is being registered.

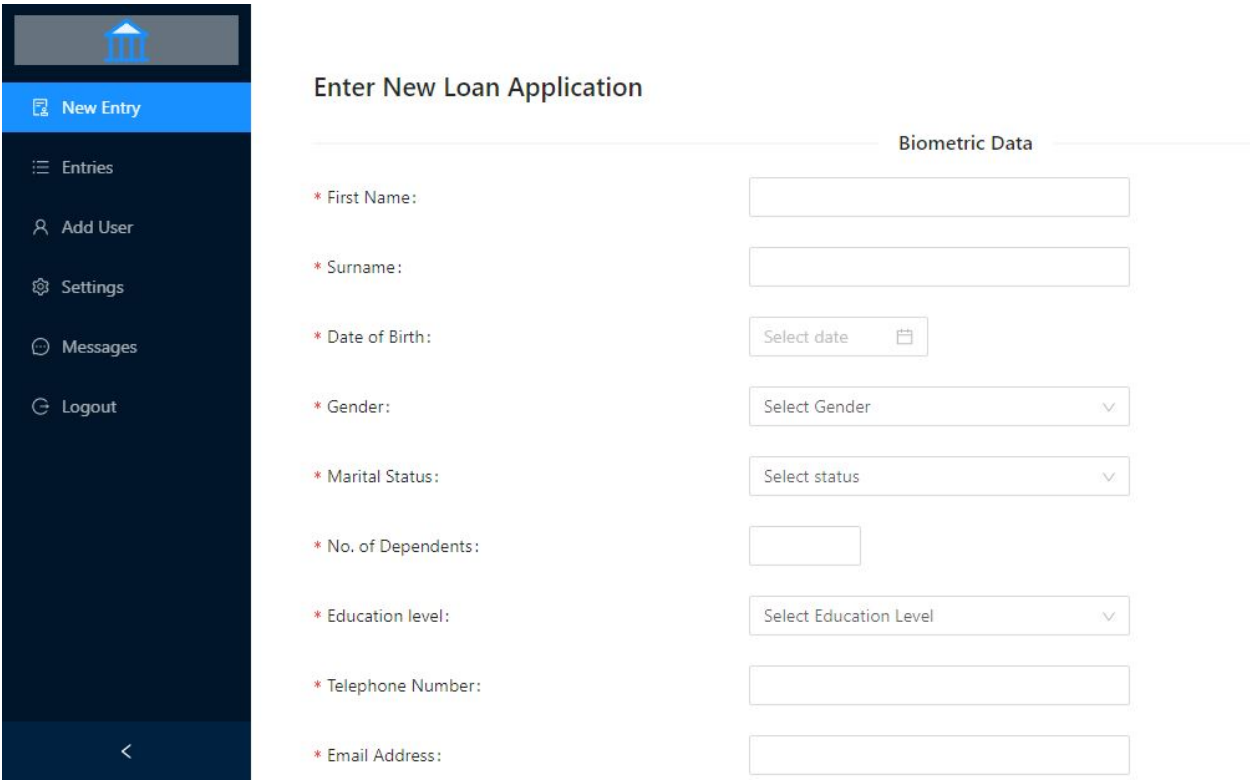
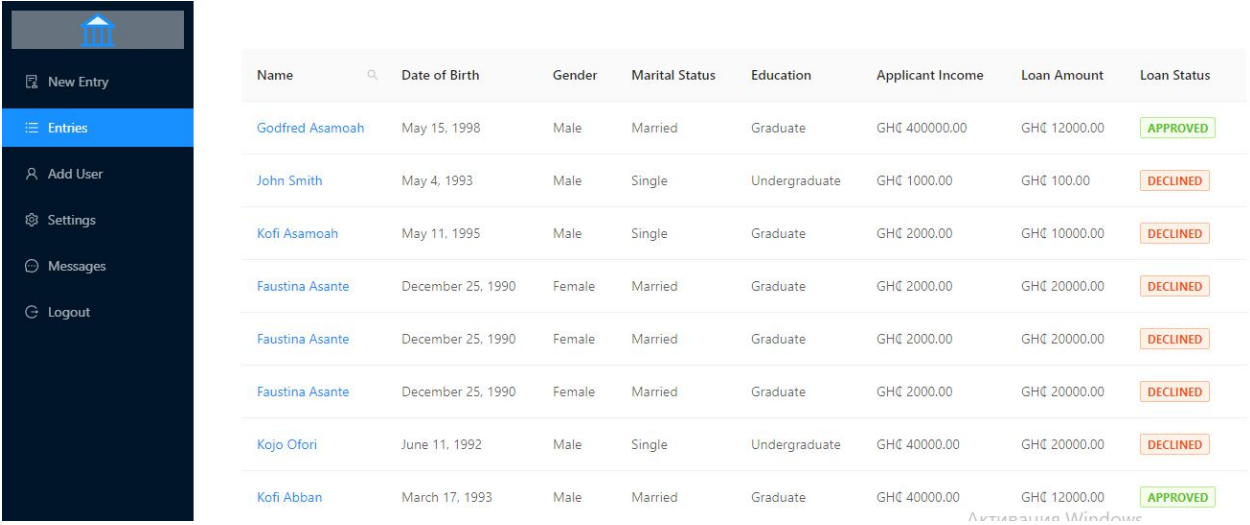


Fig. 3.3. User interface. Data fields for a new user registration

The frontend connects with the server and the design of it is just basically a form for submitting loan applications, alongside with a table that displays the data of the user and his or her loan status. The frontend contains components which represent and design the different parts of the interface the user interacts with. Figure 3.4 shows part of the current loan status of each registered client,

where it is possible to see whether a loan has been granted to a particular client or not.



Name	Date of Birth	Gender	Marital Status	Education	Applicant Income	Loan Amount	Loan Status
Godfred Asamoah	May 15, 1998	Male	Married	Graduate	GHC 400000.00	GHC 12000.00	APPROVED
John Smith	May 4, 1993	Male	Single	Undergraduate	GHC 1000.00	GHC 100.00	DECLINED
Kofi Asamoah	May 11, 1995	Male	Single	Graduate	GHC 2000.00	GHC 10000.00	DECLINED
Faustina Asante	December 25, 1990	Female	Married	Graduate	GHC 2000.00	GHC 20000.00	DECLINED
Faustina Asante	December 25, 1990	Female	Married	Graduate	GHC 2000.00	GHC 20000.00	DECLINED
Faustina Asante	December 25, 1990	Female	Married	Graduate	GHC 2000.00	GHC 20000.00	DECLINED
Kojo Ofori	June 11, 1992	Male	Single	Undergraduate	GHC 40000.00	GHC 20000.00	DECLINED
Kofi Abban	March 17, 1993	Male	Married	Graduate	GHC 40000.00	GHC 12000.00	APPROVED

Fig. 3.4. System log for loan status

3.2 Selection of Metrics

In this section, the metrics that were used to evaluate the project are described. The selection of these metrics was based on the type of project that was selected, considering the programming languages that were used in the project plus the programming paradigms that were implemented in the project design and creation.

Another consideration for the metric selection was the type of metrics that were going to be evaluated. In this case, to perform a quality check strictly on the code, Static Product Metrics were selected. Finally, the metrics were selected among the range of metrics that are supported by the ISO standards described in Section 1.3.

3.2.1 Lines of Code

Source lines of code (SLOC) or commonly called only lines of code (LOC) merely calculates the number of lines in source code of a given software unit. It is one of the simplest metrics to be implemented, but at the same time is one of the most powerful ones to evaluate especially how complex software entities are. Provided that this metric depends on some code standards and their format, it is relevant to utilize it in code when it is generated since there might be a lack of breaks in lines. Also, it can identify how much effort was applied when developing a program. Figure 3.5 shows the definition of the metric according to the ISO-9126 standard.

View

$$V^{LOC} = (G^{LOC}, R^{LOC})$$

- Grammar $G^{LOC} = (\{\text{scope}^{LOC}\}, \emptyset, \text{scope}^{LOC})$
- Relations $R^{LOC} : \emptyset$
- Mapping $\alpha^{LOC} :$

$$\alpha^{LOC}(\text{Class}) \mapsto \text{scope}^{LOC}$$

$$\alpha^{LOC}(\text{Method}) \mapsto \text{scope}^{LOC}$$

$$\alpha^{LOC}(\text{CompilationUnit}) \mapsto \text{scope}^{LOC}$$

Definition

The *LOC* value of a element $s \in \text{scope}^{LOC}$ is defined as:

$$LOC(s) = s.loc$$

Fig. 3.5. Definition of the LOC metric according to the ISO 9126.

With this metric, the idea is to evaluate the re-usability, belonging to the **usability** attribute of the quality model described in the Section 1.3. Inside this attribute, the metric will check the understandability of the software, which is the ability of a software engineer to understand the system, its components and

libraries, to later on integrate the software product into new systems. According to the ISO standard, it is expected that understanding if software can be appropriate for reuse, it very much depends on how big it is. Understandability should decrease with the increase of LOC.

3.2.2 Number of attributes and methods

Abbreviated as *SIZE2*, the number of attributes and methods metric in general does the counting of attributes and methods that the developer wrote in a class. In fact, it is mostly an object oriented paradigm metric, but it can also be applied to other modular languages by means of taking into account the amount of global variables in a module and the number of functions and methods. Figure 3.6 shows how the metric is defined. The definition is done according to the ISO standard for software quality metrics.

The purpose of this metric is to assess the **portability** attribute of the quality model described in the Section 1.3 of this work. Within this attribute, the sub-characteristic that *SIZE2* measured in the project is the *Replaceability* which works with the objective of finding conclusions on at what level new software can replace existing old parts of another software. It acts in correlation with other metrics that do measurements of other different software properties to check for opportunity of change and required effort utilizing it instead of other provided software inside the environment of such a software. Consequently, according to the ISO standard, replaceability evaluates that the incorporation of a new component instead of another component must be able to equally simulate its interface. Big interfaces are hard to assess in terms of checking how replaceable their units can be by other units. The size of an interface is particularly evaluated by *SIZE2*. Replaceability should decrease with the

View

$$V^{SIZE2} = (G^{SIZE2}, R^{SIZE2})$$

- Grammar $G^{SIZE2} = (\{\text{class}^{SIZE2}, \text{method}^{SIZE2}, \text{attribute}^{SIZE2}\}, \emptyset, \text{class}^{SIZE2})$
- Relations $R^{SIZE2} : \{\text{attributeof}^{SIZE2} : \text{attribute}^{SIZE2} \times \text{class}^{SIZE2}, \text{methodof}^{SIZE2} : \text{method}^{SIZE2} \times \text{class}^{SIZE2}\}$
- Mapping α^{SIZE2} :

$$\alpha^{SIZE2}(\text{Class}) \mapsto \text{class}^{SIZE2}$$

$$\alpha^{SIZE2}(\text{IsMethodOf}) \mapsto \text{methodof}^{SIZE2}$$

$$\alpha^{SIZE2}(\text{IsFieldOf}) \mapsto \text{fieldof}^{SIZE2}$$

$$\alpha^{SIZE2}(\text{Method}) \mapsto \text{method}^{SIZE2}$$

$$\alpha^{SIZE2}(\text{Field}) \mapsto \text{attribute}^{SIZE2}$$

Definition

The $SIZE2$ value of a class $c \in \text{class}^{SIZE2}$ is defined as:

$$A(c) = \text{pred}(c, \text{attributeof}^{SIZE2})$$

-- set of attributes directly contained in c

$$M(c) = \text{pred}(c, \text{methodof}^{SIZE2})$$

-- set of methods directly contained in c

$$SIZE2(c) = |A(c) \cup M(c)|$$

Актив
Чтобы

Fig. 3.6. Definition of the $SIZE2$ metric according to the ISO 9126.

increasing of $SIZE2$.

3.2.3 Cyclomatic complexity

The Cyclomatic complexity, abbreviated as CC, is the measurement of the level of complexity of a software control structure. In other words, CC is the amount of paths that are linearly independent and hence, it is the minimum amount of independent paths when running software. Figure 3.7 shows how the CC metric is defined according to the ISO-9126 standard of software quality.

This metric was implemented in the research to analyze the **efficiency** attribute of the quality model described by the ISO 9126 standard in Section 1.3. As mentioned earlier, efficiency of software analyzes how well software

View

$$V^{CC} = (G^{CC}, \emptyset).$$

- Grammar $G^{CC} = (T^{CC}, P^{CC}, \text{method}^{CC})$
- Entities $T^{CC} = \{\text{method}^{CC}, \text{ctrl_stmt}^{CC}\}$
- Productions P^{CC} :

$$\text{method}^{CC} = \text{ctrl_stmt}^{CC} *$$

$$\text{ctrl_stmt}^{CC} = \text{ctrl_stmt}^{CC} *$$

- Mapping α^{CC} :

$$\alpha^{CC}(\text{Method}) \mapsto \text{method}^{CC}$$

$$\alpha^{CC}(\text{Switch}) \mapsto \text{ctrl_stmt}^{CC}$$

$$\alpha^{CC}(\text{Loop}) \mapsto \text{ctrl_stmt}^{CC}$$

Definition

The CC value of a method $m \in \text{method}^{CC}$ is defined as:

$$CC(m) := |\text{succ}^+(m, \text{contains}^{CC})| + 1$$

Fig. 3.7. Definition of the CC metric according to the ISO 9126.

can give a given level of performance in relation to the amount of utilized resources. This quality attribute can be useful for the control and prediction of how much software satisfies requirements of efficiency. That is why, inside this attribute, the sub-characteristic of *time behavior* was evaluated in this work. Time behavior checks for the time behavior of software that operates or is tested in combination with other computer systems. As to the ISO standards, it is defined that time behavior could actually become worse with a high degree of Cyclomatic Complexity.

3.2.4 Message passing coupling

Message passing coupling (MPC) does the measurement of the number of method calls belonging to a class that are used in other classes. Thus, it checks the dependency of methods that are local with methods that were implemented in other classes. MPC helps draw conclusions about the method calls among objects of the classes that are being evaluated. Such analysis helps assessing the level of effort for software maintenance, also its reusability and how much effort must be made to test the software. Figure 3.8 shows the definition of the metric according to the ISO standard.

View

$$V^{MPC} = (G^{MPC}, R^{MPC})$$

- Grammar $G^{MPC} = (\{\text{class}^{MPC}\}, \emptyset, \text{class}^{MPC})$
- Relations $R^{MPC} : \{\text{call}^{MPC} : \text{class}^{MPC} \times \text{class}^{MPC}\}$
- Mapping $\alpha^{MPC} :$

$$\alpha^{MPC}(\text{Class}) \mapsto \text{class}^{MPC}$$

$$\alpha^{MPC}(\text{Invokes}) \mapsto \text{call}^{MPC}$$

Definition

The *MPC* value of a class $c \in \text{class}^{MPC}$ is defined as:

$$MPC(c) = \text{outdegree}^*(c, \text{call}^{MPC})$$

Fig. 3.8. Definition of the MPC metric according to the ISO 9126.

With this metric, the evaluation of the quality attribute of **maintainability** was performed. Maintainability helps with the understanding of how much effort is needed for software to be maintained. It can also help to analyze the level of effort that must be applied to modify a software product or any of its parts. Inside the attribute of maintainability, the sub-characteristic of *stability*

was assessed. Stability helps understand how stable a software product is. It also analyzes the risk of any unexpected effects that may occur when modifying the software.

According to the ISO standards, software parts with a high coupling could show less stability, since parts of the system interact and use resources of other parts, and this could create a negative effect. Stability is reduced with a growing MPC.

Chapter 4

Implementation

The calculation of the metrics was performed in both the backend and frontend of the project. It should be noticed that the backend follows an object oriented design paradigm, whereas the frontend follows a functional programming paradigm. Some metrics correspond more to an object oriented design (like the metric of message passing coupling); however, they were equally applied in the frontend and backend as there were equivalent functions in both sides of the project.

4.1 Calculation of LOC

The lines of code were implemented initially for the “modules” section of the front-end part of the project. The calculations were made with the help of **CLOC**¹, an NPM library to count lines of code. CLOC counts lines of code, comment lines, blank lines of many different programming languages. The tool is written in Perl² and it is possible to use this tool in various sorts of

¹To know more about CLOC, visit <http://cloc.sourceforge.net/>

²An introduction to Perl: <https://www.perl.org/>

operating systems which makes CLOC a versatile, yet efficient tool to perform the counting of LOC in a project.

4.1.1 LOC measurement in the front-end

Figure 4.1 shows the total number of lines in the whole *Components* module, where the language that was used was JavaScript in React (Jsx).

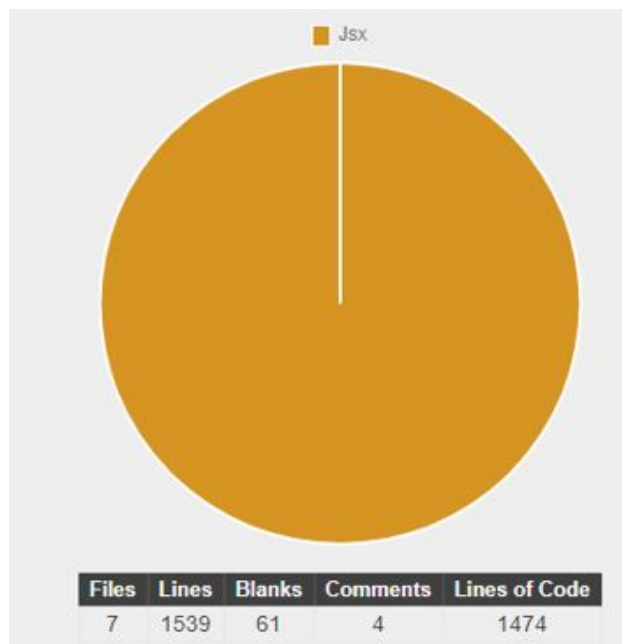


Fig. 4.1. Graph of LOC for the *Components* Module

Similarly, the calculation of the LOC of inside the members of the *Components* module can be calculate with the help o CLOC. Figure 4.2 for example, shows the calculation for the component *Dashboard.jsx* where the total amount of actual lines of code (meaning effective lines of code) adds up to 98.

Although the separate results for the rest of the components are not showed in this work, the calculation can be done equally for all of the other components and files of the frontend part of the project. Figure 4.3 shows the total calculation of LOC for the frontend, plus the report of languages that

```

cloc src/components/Dashboard.jsx
  1 text file.
  1 unique file.
  0 files ignored.

github.com/AlDanial/cloc v 1.88 T=0.41 s (2.4 files/s, 251.2 lines/s)
-----
Language          files      blank      comment      code
-----
JSX                1          6           0           98
-----

```

Fig. 4.2. LOC measurement for the *Dashboard.jsx* component of the Components module

were used to write it.

```

PS D:\Roberto\Innopolis\EightSemester\Thesis\project\loan-project\loan-frontend> cloc src --ignored=file1.txt
  16 text files.
  16 unique files.
Wrote file1.txt
  3 files ignored.

github.com/AlDanial/cloc v 1.88 T=0.34 s (46.8 files/s, 5399.1 lines/s)
-----
Language          files      blank      comment      code
-----
JSX                7          61           4          1474
JavaScript         6          21          50           158
CSS                2          10           0            61
SVG                1           0           0             7
-----
SUM:              16          92          54          1700
-----

```

Fig. 4.3. LOC for the frontend of the project. Ignored Files are the files that have no lines of code in them - src: zero sized file, src/components: zero sized file, src/util: zero sized file

4.1.2 LOC measurement in the back-end

Likewise, the LOC calculations for the backend are determined with the help of the CLOC tool. Figure 4.4 shows the calculation of LOC for the whole backend, which mainly consists of the machine learning implementation and the server written with Flask. It can be seen that the backend was written

with a total of 429 effective lines of code in the Python programming language. Figure 4.5 shows the output of the file *file1.txt*, a file that gives information on why some files were ignored when performing the calculation of the LOC of the backend.

```
PS D:\Roberto\Innopolis\EightSemester\Thesis\project\loan-project> cloc loan-backend
  9 text files.
  9 unique files.
Wrote file1.txt
  9 files ignored.

github.com/AlDanial/cloc v 1.88 T=0.17 s (23.1 files/s, 3086.5 lines/s)
-----
Language           files      blank      comment      code
-----
Python              4          98           8          429
-----
SUM:                4          98           8          429
-----
```

Fig. 4.4. LOC for the back-end of the project.

```
file1.txt - Блокнот
Файл  Правка  Формат  Вид  Справка
loan-backend: zero sized file
loan-backend/.gitignore: listed in $Not_Code_Extension{gitignore}
loan-backend/dataset: zero sized file
loan-backend/dataset/test.csv: listed in $Not_Code_Extension{csv}
loan-backend/dataset/train.csv: listed in $Not_Code_Extension{csv}
loan-backend/ml: zero sized file
loan-backend/ml/__init__.py: zero sized file
loan-backend/procfile: language unknown (#3)
loan-backend/requirements.txt: listed in $Not_Code_Extension{txt}
```

Fig. 4.5. Output of the file *file1.txt*, explaining the reasons why some files were ignored when doing the back-end LOC calculation.

4.2 Calculation of *SIZE2*

The calculation considered methods that were implemented by the programmer and not built-in methods and functions from the programming languages that the programmer imports. Likewise, attributes were calculated only under the basis of declared attributes and not imported attributes from libraries or built-in objects. Also, the measurements were done according to the definitions and theory in Computer Science for attributes and methods (or functions in functional programming) where an attribute was considered to be a variable that can be modified and viewed, and a method or function is does something or performs some action in the code.

For example, methods such as *useState()* and *useEffect()* (react built-in methods) are seen constantly throughout the code; however, they were not considered for the calculations. Setters are also not considered for the method calculation since they are just extensions of the declared attributes.

4.2.1 Obtainment of *SIZE2* in the front-end

The number of attributes and methods are implemented initially for the “modules” section of the front-end part of the project.

Components Module:

1. **AddUser.jsx:**

- Attributes:
 - (a) const alert
 - (b) const layout

(c) const tailLayout

- Methods:

(a) const showAlert()

(b) const onFinish()

2. Dashboard.jsx:

- Attributes:

(a) const jwt

(b) const pageIndex

(c) const collapse

(d) const push

- Methods:

(a) const currentPage()

3. Entries.jsx:

- Attributes:

(a) const dataSource

(b) const searchText

(c) const searchedColumn

(d) const visible

(e) const selectedEntry

(f) const columns

- Methods:

(a) const showDetail()

- (b) `const getColumnSearchProps()`
- (c) `const handleSearch()`
- (d) `const handleReset()`
- (e) `const handleClose()`

4. **Login.jsx:**

- Attributes:
 - (a) `const login`
 - (b) `const jwt`
- Methods:
 - (a) `const handleChange()`
 - (b) `const handleLogin()`

5. **Messages.jsx:**

- Attributes:
 - (a) `const userData`

6. **NewEntry.jsx:**

- Attributes:
 - (a) `const hasCollateral`
 - (b) `const hasGuarantor`
 - (c) `const hasAccount`
 - (d) `const positive`
 - (e) `const loading`

- (f) const visible
- (g) const collateralImage
- (h) const name
- (i) const layout
- (j) const tailLayout

- Methods:

- (a) const handleSubmit()
- (b) const fakeFetch()
- (c) const normFile()
- (d) const handleClose()
- (e) const handleChange()

7. Settings.jsx:

- Attributes:

- (a) const userData
- (b) const emailButtonDisabled
- (c) const passwordButtonDisabled
- (d) const form
- (e) const layout
- (f) const emailField

- Methods:

- (a) const openNotificationWithIcon()
- (b) const handleEmailUpdate()
- (c) const handlePasswordUpdate()

- (d) `const handleEmailChange()`
- (e) `const handlePasswordChange()`

For the components module there is a total of 32 attributes and 20 methods.

Util Module:

1. `hooks.js`:

- Attributes:
 - (a) `const storedValue`
- Methods:
 - (a) `function useLocalStorage(key, initialValue)`

For the Util module there is a total of 1 attribute and 1 method.

To finish the evaluation for the frontend, the analysis for the file ***service-Worker.js*** where there are 1 attribute and 4 methods:

1. Attributes:

- (a) `const isLocalhost`

2. Methods:

- (a) `function register(config)`
- (b) `function registerValidSW(swUrl, config)`
- (c) `function checkValidServiceWorker(swUrl, config)`
- (d) `function unregister()`

4.2.2 Obtainment of *SIZE2* in the back-end

Machine learning module (ML):

1. **predict.py:**

- Methods:
 - (a) def load-model()
 - (b) def predict(*entry: Entries*)

2. **train.py:**

- Attributes:
 - (a) data
 - (b) X-train, X-test
 - (c) y-train, y-test
 - (d) sc
 - (e) classifier
 - (f) y-pred
- Methods:
 - (a) def fill-missing-values()
 - (b) def split-data()
 - (c) def encode-data(*X-train, y-train*)

In total, there are 8 attributes and 5 methods for the Machine Learning module.

Now the calculation for the server and models implementation which consist of the following files:

1. **app.py:**

- Attributes:
 - (a) DEFAULT-TAG
 - (b) app
- Methods:
 - (a) def create-app(*test-config=None*)

2. **models.py:**

- Attributes:
 - db
 - database-path
 - id
 - email (*class Admins*)
 - password (*class Admins*)
 - primary-key (*class Admins*)
 - id (*class Entries*)
 - firstName (*class Entries*)
 - surname (*class Entries*)
 - dob (*class Entries*)
 - gender (*class Entries*)
 - maritalStatus (*class Entries*)
 - numberOfDependents (*class Entries*)
 - isSelfEmployed (*class Entries*)

-
- education (*class Entries*)
 - telephoneNumber (*class Entries*)
 - email (*class Entries*)
 - city (*class Entries*)
 - address (*class Entries*)
 - presentEmployer (*class Entries*)
 - occupation (*class Entries*)
 - yearsOfExperience (*class Entries*)
 - monthlyNetSalary (*class Entries*)
 - socialSecurityNumber (*class Entries*)
 - loanAmount (*class Entries*)
 - loanAmountTerm (*class Entries*)
 - loanPurpose (*class Entries*)
 - loanCategory (*class Entries*)
 - propertyType (*class Entries*)
 - creditScore (*class Entries*)
 - isAccountHolder (*class Entries*)
 - accountNumber (*class Entries*)
 - hasPendingLoan (*class Entries*)
 - hasCollateral (*class Entries*)
 - collateralImage (*class Entries*)
 - hasGuarantor (*class Entries*)
 - guarantorName (*class Entries*)
 - guarantorIncome (*class Entries*)

- hasBankingRelationship (*class Entries*)
- hasIncomeSentViaBank (*class Entries*)
- loanStatus (*class Entries*)
- Methods:
 - (a) def setup-db(*app*)
 - (b) def create-tables()
 - (c) def create-superuser(*email, password*)
 - (d) def insert(*self*) (*class Admins*)
 - (e) def update(*self*) (*class Admins*)
 - (f) def delete(*self*) (*class Admins*)
 - (g) def format(*self*) (*class Admins*)
 - (h) def insert(*self*) (*class Entries*)
 - (i) def update(*self*) (*class Entries*)
 - (j) def delete(*self*) (*class Entries*)
 - (k) def format(*self*) (*class Entries*)

There are 44 attributes and 12 methods for this section.

4.3 Calculation of CC

Cyclomatic Complexity can be calculated with the help of the following formula [19]:

$$CC = E - N + 2P \quad (4.1)$$

In this equation:

P = number of disconnected parts of the flow graph (e.g. a calling program and a subroutine).

E = number of edges (transfers of control).

N = number of nodes (sequential group of statements containing only one transfer of control).

This translates to the number of decisions + one. Binary decisions — such as “if” and “while” statements — add one to complexity. Boolean operators can add either one or nothing to complexity. For example, one may be added if a Boolean operator is found within a conditional statement.

In fact, instead of using the formula manually, there exist already software programs that have been written to automatically calculate the cyclomatic complexity of a software project. For this project, the calculation was implemented with the **boyter**³ tool, which is a very fast accurate code counter with cyclomatic complexity and COCOMO calculations written in pure **Go**⁴. Using the software tool, the following results were obtained initially for the whole project, where the results are showed in Figure 4.6.

As it is observed in Figure 4.6 the complexity amounts up to 91 for the entire project. The last 3 rows of the figure show the COCOMO calculations to estimate different costs of the project. Likewise, Figure 4.7 shows the results for the frontend, where the CC is equal to 55, and Figure 4.8 shows the results for the backend part of the project, where the CC is equal to 36.

³To know more about boyter visit: <https://github.com/boyter/scc>

⁴To learn more about Go visit: <https://golang.org/>

```
tremi@DESKTOP-H54R1TJ MINGW64 /d/Roberto/Innopolis/EightSemester/Thesis/project
$ scc --exclude-dir --no-gen loan-project
```

Language	Files	Lines	Blanks	Comments	Code Complexity
JSX	7	1539	61	4	1474
JavaScript	6	229	21	50	158
Python	5	535	98	8	429
JSON	3	16018	0	0	16018
CSS	2	71	10	0	61
CSV	2	983	0	0	983
Markdown	2	87	34	0	53
Plain Text	2	25	0	0	25
gitignore	2	163	31	42	90
HTML	1	43	0	23	20
SVG	1	7	0	0	7
Total	33	19700	255	127	19318
Estimated Cost to Develop (organic) \$605,139					
Estimated Schedule Effort (organic) 11.363685 months					
Estimated People Required (organic) 4.730993					
Processed 780545 bytes, 0.781 megabytes (SI)					

Fig. 4.6. CC calculation for the whole project with COCOMO cost calculations.

```
tremi@DESKTOP-H54R1TJ MINGW64 /d/Roberto/Innopolis/EightSemester/Thesis/project
$ scc --exclude-dir --no-gen loan-project/loan-frontend
```

Language	Files	Lines	Blanks	Comments	Code Complexity
JSX	7	1539	61	4	1474
JavaScript	6	229	21	50	158
JSON	3	16018	0	0	16018
CSS	2	71	10	0	61
HTML	1	43	0	23	20
Markdown	1	68	31	0	37
Plain Text	1	3	0	0	3
SVG	1	7	0	0	7
gitignore	1	24	5	5	14
Total	23	18002	128	82	17792
Estimated Cost to Develop (organic) \$555,049					
Estimated Schedule Effort (organic) 10.996639 months					
Estimated People Required (organic) 4.484224					
Processed 701169 bytes, 0.701 megabytes (SI)					

Fig. 4.7. CC calculation for the front-end including COCOMO cost calculations.


```
tremi@DESKTOP-H54R1TJ MINGW64 /d/Roberto/Innopolis/EightSemester/Thesis/project
$ scc --exclude-dir --no-gen loan-project/loan-backend/
```

Language	Files	Lines	Blanks	Comments	Code Complexity
Python	5	535	98	8	36
CSV	2	983	0	0	0
Plain Text	1	22	0	0	0
gitignore	1	139	26	37	0
Total	9	1679	124	45	36
Estimated Cost to Develop (organic) \$41,640					
Estimated Schedule Effort (organic) 4.109908 months					
Estimated People Required (organic) 0.900129					
Processed 78455 bytes, 0.078 megabytes (SI)					

Fig. 4.8. CC calculation for the back-end including COCOMO cost calculations.

4.4 Calculation of MPC

Like in the other sections of this chapter, the MPC measurement was divided in two parts, the measurement for the frontend and then for the backend.

4.4.1 MPC measurement in the front-end

In the frontend some components are nested, which means that they interact with each other. Props⁵ are the ones that are in charge of generating the coupling mechanisms among these components; thus, the counting for the MPC will include them as well as methods that are used in different files of the frontend.

MPC mechanisms for the Components Module:

1. **Dashboard.jsx:**

⁵Read about what a Prop is in React at: <https://reactjs.org/docs/components-and-props.html>

- Lines 42 and 43: $id = jwt$. Here, the current component passes a message to the other ones in the form of id , which holds the json web token for authenticating the user.
- Line 23: prop $history$ is passed onto the component Dashboard.
- method $useLocalStorage()$, implemented in the file $hooks.js$ does message passing as well, it passes data from local storage to the component.

2. Login.jsx:

- Line 6: prop $history$ is passed onto the component Login
- method $useLocalStorage()$, implemented in the file $hooks.js$ also does message passing in the component.

3. Messages.jsx:

- Line 5: prop id is passed onto the component Messages

4. Settings.jsx:

- Line 5: prop id is passed onto the component Settings

For the components module there is a total of 8 props and 2 methods doing the message passing coupling among components.

4.4.2 MPC measurement in the back-end

The file $app.py$ file contains the following methods and objects implemented in the $models.py$ file:

- Method setup-db

- Class Entries
- Method create-tables
- Method create-superuser
- Class Admins

The import happens in line 8 of the *app.py* file: *from models import setup-db, Entries, create-tables, create-superuser, Admins*.

Then, the following coupling happens in the backend:

1. Method setup-db: used once
 - In line 17 of app.py file: *setup-db(app)*
2. Object of class Entries: used 2 times
 - In line 150 of app.py file: *entries = Entries.query.all()*
 - In line 201 of app.py file: *entry = Entries(firstName = firstName, surname = surname, dob = dob, gender = gender, ...)*
3. Method create-tables: used once
 - In line 18 of app.py file: *create – tables()*
4. Method create-superuser: used once
 - In line 22 of app.py file: *create – superuser(email, password)*
5. Objects of class Admins: used 12 times
 - Used 2 times in line 21 of app.py file: if not *Admins.query.filter(Admins.email == email).first()*

- Used 3 times in line 53 of app.py file:
 $admin = Admins.query.filter(Admins.email == email, Admins.password == password).first$
- In line 64 of app.py file: $admins = Admins.query.all()$
- In line 78 of app.py file: $admin = Admins.query.get(admin - id)$
- Used 2 times in line 94 of app.py file: $admin = Admins.query.filter(Admins.email == email).first()$
- In line 102 of app.py file: $admin = Admins(email = email, password = password)$
- In line 112 of app.py file: $admin = Admins.query.get(admin - id)$
- In line 131 of app.py file: $admin = Admins.query.get(admin - id)$

6. Method setup-db: used once

- In line 17 of app.py file: $setup - db(app)$

Chapter 5

Evaluation and Discussion

5.1 LOC results evaluation

As it can be seen in the measurements, there are 429 lines in the backend plus 1700 lines written in the frontend, having a total of **2129** effective SLOC written in the project. Previously, it was mentioned that the level of understandability of a project for a software engineer decreases as the number of SLOC goes up.

It has been shown in the different researches, that projects containing around 14000¹ physical SLOC are considered to be already big enough to have a level of understanding that requires a lot of time to reach when studying code of a software product. Therefore, the evaluated project is not considered to be large in comparison to larger projects. Consequently, the level of understandability is high enough since little effort is required to comprehend the code for a software developer or engineer who has some experience working with software.

¹Data obtained from source marked in reference [15] of this work

5.2 *SIZE2* results evaluation

The level of portability in its sub-characteristic of *Replaceability* was measured with the implementation of this metric. It was written that the ability of a software to be replaced by other parts of code when performing maintenance or updating of software goes down with a high number of *SIZE2*. After analysis of this metric in the entire project it has been found that the backend has 52 attributes and 17 methods written by the developer, whereas the frontend contains 34 attributes and 25 methods, making a total of **86 attributes** and **42 methods** for all the project.

Therefore, it can be understood that replacing parts of the backend might be more difficult than replacing parts in the frontend, given the fact that there are more attributes in the backend. In fact, methods are also important to consider, however, unless they are nested into other components, they usually work only inside the component where they are written, and replacing them with other methods may not complicate the functioning of all the project.

All in all, a total of 86 attributes and 42 methods are not big numbers for a project (previously, in Section 5.1 it has been determined that the project in analysis is rather small); thus, replaceability is feasible in this project.

5.3 CC results evaluation

With the McCabe's cyclomatic complexity, the purpose was to determine the time behavior efficiency of the project. The results for the project gave a complexity of 55 for the frontend and 36 for the backend, giving a total of **CC = 91**. Usually, it is considered that single methods with a complexity up to 10 are manageable to understand without major difficulties [20].

Hence, the complexity found for the project does not seem to be high, given the fact that the complexity of 91 is for the whole project where there exist 42 methods written solely by the developer, besides many other methods that were fetched from built-in libraries. And this is clearly showed in the time behavior when running the project, because the program functions well and does not suffer time delays when operating it.

5.4 MPC results evaluation

With this metric, in the project the sub-characteristic of Stability was measured inside the quality attribute of maintainability. With a growing value of MPC, the system tends to be less stable, taking into account that a single edition in the code could break other parts of the project that interact and are nested with this part of the code.

The frontend has 10 coupling mechanisms, and the backend 18, adding up to a total of **28 MPC mechanisms** for the whole project. Besides, there are other mechanisms that come from the use of libraries in the project; hence, for a program of 2129 SLOC, the number of MPC could be a little high, taking into consideration that components in the frontend and classes in the backend are dependant on one another in different props, attributes and methods. This factor could trigger in the fact that a change in a class or module could highly affect the functioning of other classes and modules inside the project. Therefore, the project might not have a high level of stability. However, still the maintainability of the project would not be costly, since the project itself is not large, and, although not high, the level of stability is not low either.

Chapter 6

Conclusion

The objective of the present research is to answer the Research Question formulated in the Section 1.1, where the idea is to determine what metrics could best describe and give information about the quality attributes of the analyzed project, and what criteria it could be used to choose these metrics.

From the analysis and results obtained in Chapter 5 it is possible to conclude that some metrics give more information than others. For example, SLOC give mostly the information of how large the project in analysis is. At the same time, other metrics like *SIZE2* and MPC give information about the difficulty that could exist when maintaining the project or updating it. However these last two metrics do not seem to be quite useful once that it was determined that the project was small, because in a small project, the levels of understandability and replaceability are by default not high, and evaluating metrics in reference to these attributes may not be completely worth it.

As to the calculation of the Cyclomatic Complexity, it gave a reference of how complex the project is. This metric is in fact important with the results obtained, because it gives information of the efficiency of the program and the

difficulty of understanding it has. Unlike the *SIZE2* and MPC metrics, this metric also marked relevance in the evaluation of the project.

In conclusion, to select metrics when evaluating a software project with similar characteristics of the project analyzed in this research, there should first have to be considered the size of the project (LOC), and the complexity of it (CC). Once these factors are determined, other metrics such as the *SIZE2*, MPC, and others could be also implemented to check for quality in a software product.

6.1 Final thoughts and future work

In fact, all metrics have a level of importance when the quality of a software product is evaluated. However, some metrics could give more information and be more useful to assess a project than others, and the idea of this research was to contribute to the decision making process when selecting a set of metrics for the analysis by analyzing some metrics in a project and determining the relevance of their results.

In the future, for similar works supporting or complementing the present research, it would be good to generate the analysis of larger projects, where more evaluators could be involved in the analysis of the metrics. Also, in this research 4 metrics were taking into consideration, but a further research could evaluate more metrics in order to bring a broader guideline for those who are looking for a guide when choosing metrics among the various software quality metrics that nowadays exist.

Bibliography cited

- [1] *Software engineering: Software metrics - javatpoint*. [Online]. Available: <https://www.javatpoint.com/software-engineering-software-metrics>.
- [2] R. Rupadhyay, *Product metrics in software engineering*, Apr. 2020. [Online]. Available: <https://www.geeksforgeeks.org/product-metrics-in-software-engineering/>.
- [3] Y. H. Yang, “Software quality management and iso 9000 implementation,” *Industrial Management & Data Systems*, 2001.
- [4] W. S. Humphrey, *Managing technical people: innovation, teamwork, and the software process*. Addison-Wesley Longman Publishing Co., Inc., 1996.
- [5] S. N. Bhatti, “Why quality? iso 9126 software quality metrics (functionality) support by uml suite,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–5, 2005.
- [6] R. Lincke, J. Lundberg, and W. Löwe, “Comparing software metrics tools,” in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 131–142.
- [7] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2014.

-
- [8] B. Kitchenham, “What’s up with software metrics?—a preliminary mapping study,” *Journal of systems and software*, vol. 83, no. 1, pp. 37–51, 2010.
- [9] K. El Emam and N. F. Schneidewind, “Methodology for validating software product metrics,” *Encyclopedia of Software Engineering*, 2002.
- [10] L. Briand, V. R. Basili, and S. Morasca, *Goal-driven definition of product metrics based on properties*. Citeseer, 1994.
- [11] B. C. Pierce and C. Benjamin, *Types and programming languages*. MIT press, 2002.
- [12] R. S. Bird and P. L. Wadler, *Functional programming*. Prentice Hall, 1988.
- [13] B. J. Cox, “Object-oriented programming: An evolutionary approach,” 1986.
- [14] H.-W. Jung, S.-G. Kim, and C.-S. Chung, “Measuring software product quality: A survey of iso/iec 9126,” *IEEE software*, vol. 21, no. 5, pp. 88–92, 2004.
- [15] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, “A sloc counting standard,” in *Cocomo ii forum*, Citeseer, vol. 2007, 2007, pp. 1–16.
- [16] O. Filipova and R. Vilão, “Backend development,” in *Software Development From A to Z*, Springer, 2018, pp. 101–131.
- [17] M. Grinberg, *Flask web development: developing web applications with python*. " O’Reilly Media, Inc.", 2018.
- [18] S. Aggarwal, “Modern web-development using reactjs,” *International Journal of Recent Research Aspects*, vol. 5, no. 1, pp. 2349–7688, 2018.

-
- [19] J. Britton, *What is cyclomatic complexity?* Oct. 2016. [Online]. Available: <https://www.perforce.com/blog/qac/what-cyclomatic-complexity>.
- [20] D. Schneller, *Why good metrics values do not equal good quality*, Apr. 2021. [Online]. Available: <https://blog.codecentric.de/en/2011/10/why-good-metrics-values-do-not-equal-good-quality/>.