Journal of Universal Computer Science, vol. 27, no. 10 (2021), 1096-1127
submitted: 1/2/2021, accepted: 20/7/2021, appeared: 28/10/2021 CC BY-ND 4.0

Understanding the Impact of Development Efforts in Code Quality

Ricardo Pérez-Castillo
(1 Facultad de Ciencias Sociales de Talavera de la Reina,
University of Castilla-La Mancha
Avenida Real Fábrica de Seda s/n 45600, Talavera de la Reina, Spain,
https://orcid.org/0000-0002-9271-3184, ricardo.pdelcastillo@uclm.es)

Mario Piattini
(2 Information Technology & Systems Institute (ITSI),
University of Castilla-La Mancha
Paseo de la Universidad 4, 13071, Ciudad Real, Spain,
https://orcid.org/0000-0002-7212-8279, mario.piattini@uclm.es)

Abstract: Today, there is no company that does not attempt to control or assure software quality in a greater or lesser extent. Software quality has been mainly studied from the perspectives of the software product and the software process. However, there is no thorough research about how code quality is affected by the software development projects' contexts. This study analyses how the evolution of the development effort (i.e., the number of developers and their contributions) influences the code quality (i.e., the number of bugs, code smells, cloning, etc). This paper presents a multiple case study that analyses 13 open-source projects from GitHub and SonarCloud, and retrieves more than 95,000 commits and more than 25,000 quality measures. The insights are that more developers or higher number of commits does not necessary influence worse quality levels. After applying a clustering algorithm, it is detected an inverse correlation in some cases where specific efforts were made to improve code quality. The size of commits and the relative weight of developers in their teams might also affect measures like complexity or cloning. Project managers can therefore understand the mentioned relationships and consequently make better decisions based on the information retrieved from code repositories.

Keywords: Software quality; Software project management; repository mining; Data Science; GitHub; SonarCloud
Categories: D.2.7, D.2.8, D.2.9, K.6.1, K.6.3
DOI: 10.3897/jucs.72475

1 Introduction

Code quality has been widely investigated in the literature and has been recognized as one of the most significant factors with a direct impact in competitiveness in the software development industry [Abrahao, Baldassarre et al., 2016, Baggen, Correia et al., 2012, Janicijevic, Krsmanovic et al., 2016]. Some consequences generated by inadequate levels of quality are, for example, poor designs that lead to systems that are difficult to be maintained and extended [Schranz, Schindler et al., 2019], systems delivered with many defects resulting in a high dissatisfaction of end-users [Maxim and Kessentini, 2016], many dead or duplicated code that dramatically increase the

maintenance and evolution cost [Perez-Castillo, Piattini et al., 2018], among many other harmful effects.

Such problems, and their consequences, are usually addressed by researchers and practitioners considering a product and/or process quality approach. This means, software engineers control and assure the software quality regarding internal software features [Baggen, Correia et al., 2012, Papamichail and Symeonidis, 2020], as well as the processes to produce software in a proper way [Shrestha, 2018].

Both approaches have been supported by international standards for several decades. First, ISO/IEC 9126 emerged in 1991 (then updated in 2001) [ISO/IEC, 2001] providing a quality model with a set of software quality characteristics to be evaluated during software development. That standard was then superseded by ISO/IEC 25010 (SQuaRE) [ISO/IEC, 2011] by including more characteristics and specifying in detail others. In a similar way, the process quality has been addressed by international standards like the ISO/IEC 33000 series [ISO/IEC, 2015] that allow the assessment and improvement of the software development process based on the capability evaluation. That standard superseded the previous one, ISO/IEC 15504 (SPICE) [ISO/IEC, 2004].

Apart from the product and process software quality approaches, there is a third approach: the software quality treated from the project management perspective. While the process software quality approach considers the repetitive processes at organizational levels, the project perspective takes into consideration the factors of individual projects [Jia, Mo et al., 2018] (limited in time and conducted by specific development teams) and how those impact on code quality. For example, today development software projects deals with time-to-market pressure [Kuutila, Mäntylä et al., 2020], and continuous changes or new features requested by customers and end-users [Papamichail and Symeonidis, 2020]. With agile software development approaches, software evolution methodologies should also consider new requested features while stable functionality has to be supported [Gao, Li et al., 2020]. Also, software development projects include large and highly distributed teams around the world [Borrego, Morán et al., 2019]. According to this contemporary context in software development project, there is an evident need to produce software with enough quality [ISO/IEC, 2011].

Although product and process quality has been extensively studied, there is a limited research about how the project context impacts on code quality. In particular, there is not a clear understanding of some relationship on project metrics and code quality metrics. This study tries to figure out how the development effort (i.e., numbers of contributors and the way in which they contribute during a software development project) influence certain code quality measures, such as bugs, code smells, cloning, among other.

The main objective of this study is to analyse how some aspects in the context of software development projects influence software quality. The study follows a repository mining approach. During the last years, a lot of studies considered "mining of software repository data for software development analytics" [Czerwonka, Nagappan et al., 2013]. Some of these studies follow descriptive approaches (i.e., those that apply statistical tests and models to explain relationships between mined data) [Papamichail and Symeonidis, 2020, Saini and Chahal, 2018, Singh, Chaturvedi et al., 2017, Verma and Kumar, 2017, Wang, Meng et al., 2019]. Whilst other studies are more predictive since those are based on artificial intelligence (AI) methods like machine (or deep) learning to predict software quality as a classification problem

[Dam, Pham et al., 2019, Hoang, Dam et al., 2019, Kiehn, Pan et al., 2019]. This means the quality of the new pieces of code are classified regarding the previous analysis of code that have been characterized. Despite the promising results, these AI, classification-based proposals have some weaknesses. For example, metrics vary between projects [Lewis, Lin et al., 2013] which prevent the reuse of these classification/predicting models since system has to be trained again [Ghotra, McIntosh et al., 2017]. Also, some of these models generate many false positive [Nayrolles and Hamou-Lhadj, 2018]. For this reason, this study follows a descriptive repository analysis approach based on statistical models rather than a predictive one based on AI techniques.

In particular, we analyse in combination GitHub and Sonarcloud (two open cloud services automatically accessed through their APIs) to retrieve code and quality measures information of the 13 open-source systems that are in both repositories at the same time. The proposal defines a common data model to collect and relate information collected from the two repositories. First, GitHub supports the code repositories of these projects, so information about commits and committers can be retrieved. Second, Sonarcloud is queried to get information about builds and associated software metrics (bugs, code smells, violations, duplications, etc.). In total, more than 95,000 commits and 782 builds (with more than 90 different metrics measured by each build) were analysed. After data were collected and pre-processed, some correlation and clustering algorithms were applied to figure out relationships between software development effort and software quality measures.

The main insight achieved from this multiple case study is that the quantity of committers and commits may influence the code quality measures. Although the development effort does not necessarily affect code quality, there are some releases where the more committers and commits (relative to the system size), the worse code quality. The main implication of this insight is that project managers and, in general, software engineers can better understand the mentioned relationships and, therefore, they can make better decisions during project execution, so that, development effort can be adjusted, and code quality levels might be improved.

The remaining of the paper is structured as follows. Section 2 first introduces some concepts necessary to understand the proposal. Section 3 presents work related to this research. Section 4 explains the research method followed in the study. Section 5 presents the design and planning of the case study according to the research method. Section 6 shows the experimental results obtained after conducting the case study on 13 open-source projects. Section 7 sets out our conclusions and directions for future work.

2 Preliminaries

This section introduces some concepts that are used throughout the whole paper.

2.1 Concepts and measures

First, the concept and associated measures used in the case study are introduced:

Releasing. This concept refers to the version evolution of the development project. The release version is also an independent variable representing the unit of analysis

according to the case study design. It expresses the open-source project status for a specific version.

Development effort. This concept attempts to measure the volume of changes and contributors in a specific released version. These measures are associated with values retrieved from GitHub (the public control version system employed in this study).

- LOC is the total number of lines of code in the system for that release and represents the size of the system.
- Commits is the total number of source code changes registered in the version control system for a certain release branch, which is normalized by the system size (i.e., LOC).
- Committers is the total number of different developers who committed something on a certain release branch. It is normalized using LOC.
- Changes by Commits is the average of changes by commit, i.e., total of additions and deletions in source code divided by the total number of commits during a specific release.
- Committers weight represents the relative contribution of each individual committer during a specific release. This is used under the assumption that in some releases could be involved many different committers, but only few of them accumulate the majority of commits. Because of this, it measures compute the percentage of contribution of each committer. Then the measure aggregates all these percentages through a harmonic mean. In this way, we limited the weight of sporadic contributors.

Code quality. This concept includes measures and indicators about the quality of the software that is released. Provide a definition for software quality is a hard task due to the numerous definitions in the literature. Probably, one of the most accepted definition is provided in ISO/IEC 25010, that defines software quality as "the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value" [ISO/IEC, 2011]. Measures for this concept come from SonarCloud and are normalized considering the system size:

- Bugs represents the number of bugs. According to SonarCloud, a bug is an issue that symbolizes something wrong in the source code. Thus, bugs should be fixed as soon as possible before actual defects appears.
- Code Smells is the number of maintainability-related issues in the source code. Assuming this kind of issue is left as-is, maintainers will spend time on making code changes. Even worst, code smells can mislead maintainers and lead them to introduce additional errors when they make changes.
- Complexity refers to the cyclomatic complexity and it is related to the number of different paths in the source code sequence. Intuitively, bifurcations in the control flow increments complexity. The calculation can slightly vary by programming language.
- Violations is the number of rules checked by SonarCloud that are violated. Rules are the mechanism of SonarCloud to detect bugs and code smells but also security issues, vulnerabilities, and so on. So, this measure is an indicator of how many rules, as defined for the project, are not met. It should be noticed that rules can be managed and customized for every project despite there are some built-in sets of rules that are predefined in SonarCloud.

- Duplicated lines, it is the number of duplicated lines of source code. For a piece of code to be tagged as repeated, there should be at least a certain number of successive and duplicated tokens. Depending on the programming language, the threshold number of tokens could vary. For example, in COBOL, it is 30 lines of code, while in Java, it corresponds to 10 successive and duplicated statements.
- Open issues represent the total of violations that have been marked by developers as 'open'. Possible status values in SonarCloud are: Open, Confirmed, Reopened, Resolved. This measure is considered to pay attention to the effort on fixing issues.

2.2 Data model

The raw data of the case study is collected from two public data sources. On one hand, GitHub (an online version control system) is accessed to get the code base of projects together with all the information about development effort (e.g., numbers of committers, commits, etc.). On the other hand, the raw data concerning code quality metrics that were retrieved from SonarCloud (a platform to inspect continuously the quality of source code and detect bugs, vulnerabilities, and code smells). These two data sources are then integrated into a common database to be analysed.

The data collected from both data sources are integrated into a single NoSQL database. Figure 1 shows the summarized data model for managing all the retrieved information. Top part represents GitHub entities while SonarCloud is represented in the bottom part. Entities from the two data sources are related through the concept of project and their releases (see Figure 1). This data model is then used during the pre-processing phase to get the final data set to be analysed.

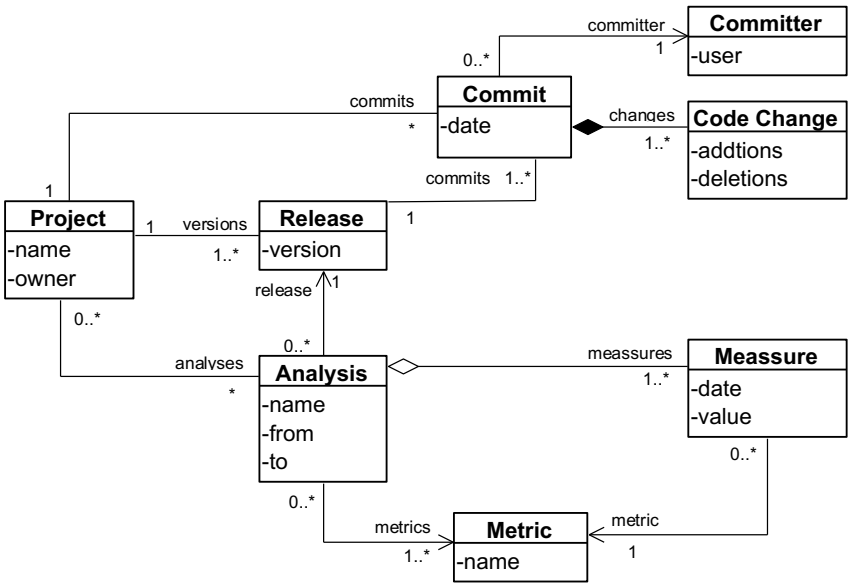

Figure 1: Simplified data model for GitHub and SonarCloud information

2.3 Descriptive analysis

This study focuses on a descriptive analysis based on statistical analysis. The study employs qualitative and quantitative data analyses. On one hand, the qualitative analysis consists of *explanation building,* as explained in [Yin, 2014]. This technique tries to identify patterns derived from the possible cause–effect relationships and develop possible explanations. Thus, it possible to discover and describe relationships concerning the tendency of the factors under analysis (i.e., commits and committers by release) and the code quality measures fluctuation alongside the project evolution. This qualitative analysis technique mainly relies on descriptive statistics as well as explanatory plots with visualization relating different variables.

On the other hand, quantitative analysis is used in combination to confirm preliminary insights obtained through the preliminary qualitative inspection. In this sense, the technique used is the statistical correlation tests and, in particular, the Spearman correlation test. In statistics, the Spearman correlation coefficient is a measure of the linear correlation between two variables. We have chosen Spearman's test instead of Pearson correlation test since some of the managed measures are not normally distributed variables. Also, Spearman's test produces better results for outlier values, which is the case of some measures with various atypical cases. The Spearman's correlation factor has a value ranging between -1 and 1. Values close to 1 are positive linear correlation, 0 is a non-linear correlation, and −1 is a negative (inverse) linear correlation. The correlation values are used to determine (with a certain statistical significance) if a specific development effort factor affected code quality measures.

Together with the correlation test, this study uses a clustering algorithm. This kind of algorithms classifies data points int different groups. Data points in the same group share common features, while data points in different groups should have clear dissimilarities features. The rationale for using a clustering algorithm is that we can groups different releases with similar properties. As we mentioned, the relationships between development effort and quality measures might vary for different releases. For example, a higher and hectic development effort could lead to poor quality levels in some cases, while thorough development efforts specifically focus on improving quality may lead to the different result.

The clustering algorithm used in this study is the K-means. This machine learning algorithm classifies data according to their attributes into K number of clusters. This algorithm is a method of unsupervised learning. The user first defines K, the algorithm then defines K centroids for K clusters (initially very distant). Data points are then classified into one of the clusters according to the minimal centroid distance. In every step, centroid is then recalculated according to the points in each cluster, and data points changes with the new centroids. The K-means algorithm ends when clusters elements are stable.

3 Related Work

First, section 3.1 presents some works that have analysed the impact of development project context on software quality. Second, some works that have analysed code repositories to extract both software project and quality metrics are presented in section 3.2.

3.1 Project influence on Software Quality

The research hypothesis of this research is in line with previous works that have analysed the impact of project context and environment on software quality metrics. For example, [Chunli and Rongbin, 2016] investigated the main problems of software project management (as well as software project risk management) and defined a strategy of improving the quality management of software project based on CMMI. Among other things, the defined strategy of that work focuses on the comprehensive ability of software developers. In the same line, [Hayat, Rehman et al., 2019] have focused on the impact of agile methodologies on the software project management, including quality management. Moreover, [Wong, Yu et al., 2018] studied the relationships of some project activities alongside project lifecycle. This study analysed those relationships based on the monitoring of software changes. Authors stated that "it is important for every project team member to comply and adhere to change control standards following best practices maximizing business advantages, and enhancing product quality". Our study based on the analysis of commits and committers is in line with the investigation of change control standards of that study.

[Janicijevic, Krsmanovic et al., 2016] proposed a markovian decision system that models the "stochastic processes of a quality management system and selection of the optimum set of factors impacting software quality". Among other factors, this research evaluates how programmer skills and software development methods impact the customer requirements fulfilment. In comparison with this work, this is too much focussed on customer satisfaction regarding requirement fulfilment. [Papamichail and Symeonidis, 2020] analysed different software by employing the trends of static analysis metrics for evaluating software maintainability. Although this approach analyses different releases, this research does not consider information about commits or committers, neither do other works.

[Jia, Mo et al., 2018] investigated through a cluster analysis how environmental factors influence software quality. This research analyses more than 200 factors grouped into 11 categories such as challenging work, enterprise assistance, suitable physical conditions, the nature of the activity, team distribution, technical competence, among other. In contrast with our proposal, this study focuses on decision-making behaviour in software development project instead of the relationship of such factor and software quality. [Schranz, Schindler et al., 2019] addressed the challenge of dissatisfaction and lack of engagement of developers in free open-source software and how it affects code metrics. However, this study presents only a case study in the context of a refactoring process of a single open-source system.

Moreover, there have already been studies analysing the relationship between quality and number of developers in open-source projects. [Norick, Krohn et al., 2010] did not find significant evidence about the effect of number of developers in some code metrics like cyclomatic complexity, lines of code per function, comment density, and maximum nesting in various open-source systems. [Voulgaropoulou, Spanos et al., 2012] draw similar conclusions after analysing various R statistical open-source systems. A more recent study by [Roehm, Veihelmann et al., 2019] rejected the hypothesis of code developed by a team has better quality than code developed by an individual, since the analysis thousands of GitHub repositories did not provide such evidence.

In a more specific manner, [Greiler, Herzig et al., 2015] investigated the dependency between code ownership (through the usage of several ownership metrics proposed in [Bird, Nagappan et al., 2011]) and the probability of having defects in source code at file and directory level. The code quality is analysed by only measuring the number of bugs. Also, although our study uses some metrics similar to the ownership metrics, these are analysed regarding various quality metrics at release level. In addition to code ownership, [Rodriguez, Tanaka et al., 2018] analysed how are the working behaviours of developers (basically the temporal dimension) and the effects these habits have on coding efficiency. [Joonbakhsh and Sami, 2018] also study the interactions logs gathered by the integrated development environments to compute Personal Software Process (PSP) [Humphrey, 2005] quality metrics. PSP includes metrics like number of defects introduced by a developer but are oriented to the improvement of the developer performance. Whilst our study focuses on the analysis of a project and their quality metrics during its whole lifecycle. Also, [Wang, Meng et al., 2019] provided an automatic way for calculating some developer scores in GitLab that is based on the amount of code and their quality values, their contribution, personalized commit time, and projects in which they are involved. Similar to the previous one, this study focuses on quality of software developers instead of software quality.

[Perez-Castillo, Piattini et al., 2018] provided a similar study considering information of project context collected from code repositories, although the scope of the study was code cloning while other software quality metrics were not included, and this work also performs a clustering to figure out different project trends. Additionally, the new contribution of this paper is the method for automatically collecting and analysing data from GitHub and SonarCloud. Other similar study that is focuses on project information is provided by [Gautam, Vishwasrao et al., 2017]. This study analyses through a clustering algorithm the continuous integration practices followed by teams in open-source development projects and how these practices impact in the projects' success. The practices analysed are activity, popularity, size, testing, and stability (different to those used in our study) and are used to help with developer hiring.

3.2 Repository Mining for Quality Analytics

There is an increasing interest on analysing source code repositories since it is perceived that there is a vast amount of knowledge that can be mined with plenty of application [Güemes-Peña, López-Nozal et al., 2018, Kalliamvakou, Gousios et al., 2016]. Some works like [Papamichail and Symeonidis, 2020, Perez-Castillo, Piattini et al., 2018, Wang, Meng et al., 2019] presented in the previous section also consider the analysis of source code repositories.

[Namiot and Romanov, 2020] discuss recurrent neural networks used to analyse software repositories. This study provides a survey about predictive analysis methods used for analysing code repositories, such as methods of classification, clustering and deep learning. This is secondary study and, therefore, does not provide a particular proposal to analyse code repositories. However, this study illustrates the problems that can be addressed through predictive analysis like classifying and predicting errors, changing the properties of code in the process of its evolution, detecting design flaws and debts, or assist for code refactoring.

[Saini and Chahal, 2018] analysed changes and their associated, intentional messages in code repositories. This research analysed through a classification algorithm hundreds of projects with the aim of deriving change evolution patterns. Similarly, [Coelho, Valente et al., 2020] present a data-driven proposal to evaluate the maintenance degree of projects in GitHub. The scope of these two proposals is the software evolution and maintainability analysis, without considering a wide software quality perspective.

[Singh, Chaturvedi et al., 2017] provided a method to estimate the release time of a software product by analysing the complexity of code the code change complexity (defined as entropy), code improvements, implementation of new functionality and bugs fixing. Some of the metrics are used in our study, however we analyse how these metrics are impacted by others (i.e., as dependent variables), instead of using it as independent variables. This goal of the study was to analyse the estimation of releasing time rather than software quality.

[Verma and Kumar, 2017] proposed a method for predicting defect density in software through the usage of code repository metrics and applying linear regression models. In a similar way, [Querel and Rigby, 2018] analyses source code statically and commits in code repositories to provide precise bug warnings.

In contrast to these statistical methods, [Hoang, Dam et al., 2019] use AI to predict defects for every new commit. This study uses a deep learning algorithm to classify the new change regarding the history of last changes and assumes that defects that happened under similar changes can be reproduced. Similarly, [Dam, Pham et al., 2019] predict defects through a deep learning algorithm, however it uses the internal code structure (the abstract syntax tree) to predict code defects. In the same line, [Kiehn, Pan et al., 2019] define a machine learning model for classifying change risks. These studies are aimed at predicting defects by considering the characterization of code changes and code structure, while our study focuses on the project development effort data and follows a statistical analysis approach instead. There are some common problems for all these AI, classification-based proposals. First, metrics usually vary from one project to another, preventing the reuse of these classification/prediction models [Lewis, Lin et al., 2013]. Moreover, these models something generate high false positive rates by classifying elements [Nayrolles and Hamou-Lhadj, 2018]. Also, the accuracy defect classifiers is impacted by the features used in the training phase [Ghotra, McIntosh et al., 2017].

Finally, [Manzano, Ayala et al., 2019] provided a combination of descriptive and predictive method. They proposed a generic tool based on R scripts that is able to generate a REST API to predict the evolution of metrics based on various forecasting models. This generic tool can consume data from various repositories such as source control, defect tracking systems and project management tools.

4 Research Method

The main goal of this research is to analyse how some factors in the context of software development projects (such as commits and committers among others) affect code quality metrics (such as bugs, code smells, cloning, complexity, among other). The study follows a repository mining approach, i.e., it is based on the data stored in code and quality repositories regarding real-life software development projects. Specifically,

the study is based on statistical methods to carry out a descriptive analysis of the correlation relationships between factors analysed.

To achieve the main goal, the specific research method used in this paper is a multiple case study. The multiple case study is designed, conducted and reported according to the method proposed by [Runeson, Host et al., 2012]. Multiple case study is an empirical research method that allows to extend the study to various cases (see Figure 2). Thus, examining more software development projects leads to further information about the phenomenon under study. "This is not only due the increased amount of data collected from the informants but also based on the characteristics of the selected cases themselves" [Runeson, Host et al., 2012]. Nevertheless, multiple case studies must not be associated with the concept of statistical sampling or statistical replication. While statistical approaches are based on sampling and representativeness, case studies rely on the cases and their features [Runeson, Host et al., 2012] (independently cases are typical or special in some way).

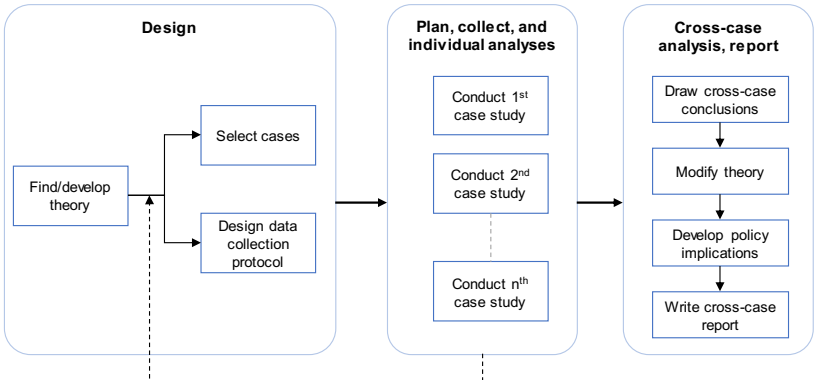

Figure 2: Overview of multiple case study (Adapted from [Yin, 2014])

Our study includes the analysis of 13 real-life software development projects. Section 5 shows the design of the multiple case study and how it has been planned, while Section shows the analysis of data and its interpretation for answering research questions. The evaluation of the validity of this work is also presented at the end of that section.

4.1 Execution protocol and data collection

Considering previous design aspects, the multiple case study was carried out by following the steps depicted in Figure 3. First, both data sources are queried according to the python scripts depicted in section 5.7. The raw data collected from GitHub and SonarCloud are then stored into the Mongo database. This database stores 95,000 commits and 782 analyses with more than 90 measures by each analysis. The raw data is then pre-processed to aggregate data and compute other derived measures by each release of each project. As a result, a dataset with 156 rows is produced as a CSV file. This file is what we use to import in R and perform treatments and analyses depicted in section 5.7.3. The outgoing R markdown file is online available at [Pérez-Castillo, 2020]. Until this point, all the tasks are performed (semi)automatically. However, the

last task is manually done, which consists of the analysis and interpretation of the obtained results in R.

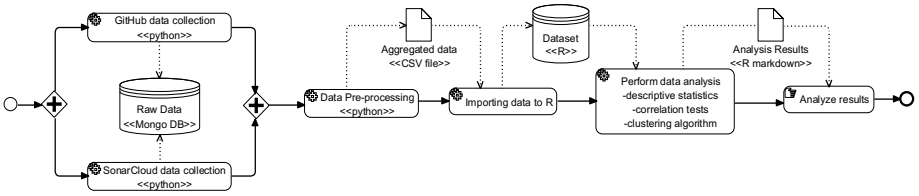

Figure 3: Case study procedure

4.2 Involved software development projects

In order to select software development projects, we stablished the following selection criteria.

- **C1. It should be an open-source system**. This criterium ensures that the project consisted of the development of an open-source system which could be freely accessed and analysed and in which different developers made their source code contributions.
- **C2. It must be available as a project in GitHub and SonarCloud**. First, this guarantees first that the source code was tracked through the use of Git, a control version system. Therefore, the project attains information on committers, commits, and so on. Second, it ensures that the selected development project is analysed in SonarCloud to get quality measures during the project evolution. In other words, we ensure that there exist data from the two data sources we established for data collection (cf. section 5.7).
- **C3. It must have at least 3 releases with quality software measures**. This criterium helps to select projects that follows a configuration control, and version history is managed properly through the evolution of the source code and their associated quality measures. Therefore, we stablished a minimum number of three versions to evaluate the aforementioned measures during the evolution of the development project.
- **C4. It must not be smaller than 5 KLoC**. This allows to discard small systems and thus guarantee the generalisation of the results. The limit was established as 5000 lines of source code. It should be noticed that this value could vary for the different versions, so in or to compute and apply this criterium we consider the value for the last release.

After applying the mentioned selection criteria, 13 open-source systems were selected to be analysed in this study. Table 1 provides the owner organization and project keys in GitHub, as well as the project name in SonarCloud. Thus, the GitHub URL can be composed as: *https://github.com/<organizaton_key>/<project_key>*; while the SonarCloud URL can be composed as: *https://sonarcloud.io/dashboard?id=<project_key>*. Table 1 also provides a brief description of the systems, the programming language in which the systems was coded (the most common one), the number of lines of source code, and the number of releases (versions) to be analysed.

Table 1: Open-source projects included in the case study

Id	GitHub Org.	GitHub Project	SonarCloud Project	Description	Lang	KLoC	Releases	From	To
P1	ant-media	Ant-Media-Server	io.antmedia:ant-media-server	Ant Media Server supports RTMP, RTSP, MP4, HLS, WebRTC, Adaptive Streaming, etc	Java	23	28	2018-04-02	2020-03-01
P2	esig	dss	eu.europa.ec.joinup.sd-dss:sd-dss	Support electronic signature services	Java	80	11	2018-12-17	2020-03-10
P3	jacoco	jacoco	org.jacoco:org.jacoco.build	Java code coverage library	Java	16	13	2014-09-07	2020-02-28
P4	apache	jmeter	JMeter	Designed to load test functional behaviour and measure performance	Java	115	3	2019-10-05	2020-03-01
P5	demasherbrezon	jradio	ru.r2cloud:jradio	Software radio decoding written in Java	Assem-bly	55	13	2019-03-03	2020-03-03
P6	monicahq	monica	monica	Personal CRM (Customer Relationship Management)	PHP	35	38	2018-01-08	2020-03-02
P7	payara	Payara	fish.payara.server:payara-aggregator	Payara Server is a middleware platform for reliable and secure deployments of Java EE	Java	799	8	2019-11-28	2020-03-0
P8	simgrid	simgrid	simgrid_simgrid	Framework for the simulation of distributed applications (Clouds, HPC, Grids, IoT, etc.)	C++	91	6	2019-06-06	2020-03-03
P9	apache	sling-org-apache-sling-app-cms	apache_sling-org-apache-sling-app-cms	Apache Sling is a Web framework designed to create content-centric, java-based applications.	Java	10	11	2019-05-29	2020-03-08
P10	apache	sling-org-apache-sling-scripting-jsp	apache_sling-org-apache-sling-scripting-jsp	Support for JSP scripting in Apache Sling.	Java	27	7	2019-09-26	2020-03-01
P11	apache	sling-org-apache-sling-scripting-sightly-compiler	apache_sling-org-apache-sling-scripting-sightly-compiler	The Apache Sling scripting HTLM compiler	Java	7	17	2019-06-03	2020-03-09
P12	SonarSource	sonar-dotnet	sonaranalyzer-dotnet	Code analyser for C# and VB.NET projects	C#	65	57	2018-05-03	2020-03-10
P13	SonarSource	sonarqube	org.sonarsource.sonarqube:sonarqube	A tool for continuous inspection of code quality	Java	296	12	2018-10-19	2020-02-28

5 Case Study Design & Planning

This section presents the design and plan of the multiple case study according to the aforementioned method [Runeson, Host et al., 2012].

5.1 Rationale and objective of the study

The motivation of the study is the need to get a better understanding about some specific relationships between development effort aspects (such as committing or releasing) and the evolution of code quality metrics throughout different releases. The main rationale for this, from the practitioners' point of view, is to depict and build a theory to provide a deeper comprehension of such relationships.

Considering such motivation, the goal of the study is to determine how code quality metrics in different releases in open-source development projects are influenced by some specific properties in the development effort. We expect to enhance the prediction and, consequently, the prevention of inadequate quality levels and, therefore, it contributes to make better decisions. This goal is lead to the research questions that are introduced afterword.

5.2 Cases and units of analysis

The design of the study consists of a holistic multiple case study [Yin, 2014] since it focuses on 13 development projects that are, in turn, analysed for each project release. Actually, the unit of analysis (acting as independent variable) is each different project release. A project release is typically associated with a version in the version control system that is queried. Although GitHub, the version control system, can be queried with data for individual commits, SonarCloud, the quality measuring system, is typically used to take measures for every version after the respective release. The study consequently first considers all the raw data from both systems, and then it pre-processes data to aggregate some metrics by version. This is then explained in section 5.7.

5.3 Theoretical framework

In this study, Section 3, that presents related work, is considered as the theoretical framework. Those works shows other research analysing the effects of projects' development efforts, along with the evolution of code quality metrics throughout the project lifecycle to predict and prevent inadequate quality levels. The limited theoretical development in the area of predicting and decision-making process for managing code quality in development projects signifies that it is hard to generalize the theoretical base. Nevertheless, the aforementioned research has influenced the design of this study.

5.4 Research questions

- **RQ1**. How does the development teams' effort during project releasing affect the evolution of code quality measures?
- **RQ2**. Are there specific trends or patterns in the evolution of code quality measures regarding different development effort configurations?

The study defines two research questions: RQ1 and RQ2. RQ1 is based on the main assumption that the kind of commits and number of contributors can affect some quality measures and indicators. With RQ1 we attempt to figure out what measures are affected and how are affected, i.e., positively or negatively. RQ2 is then proposed to investigate if there is certain trends or patterns during the evolution of software development projects concerning code quality measures and development effort ones. The analysis of these possible patterns might be useful to predict and make decision during software project lifecycles.

5.5 Propositions and hypotheses

The defined research questions are related to the evolution of certain development effort measures retrieved for each project release, and how certain aspects may influence on the fluctuation of code quality measures. Regarding RQ1, the formulated hypotheses consist of a null and alternative hypothesis as follows.

- $\boldsymbol{H_{0_{RQ1}}}$: There is no significant difference in the code quality measures for different numbers of commits and committers.
- $\boldsymbol{H_{1_{RQ1}}}$: $\neg H_{0_{RQ1}}$

With regard to these hypotheses, the proposition is that a higher number of committers and commits in a release leads to variations in code quality measures. This assumption is based on the idea that the more developers there are contributing to the same time, the less communication there might be in the development team, thus making reuse difficult and leading to more bugs, code smells or code violations [Harder, 2013, Perez-Castillo, Piattini et al., 2018]. It should be noticed that the examination to the number of committers and commits implies two factors that do not necessarily have a linear relationship. Also, it should be pointed out that this hypothesis assumes values normalized by the system size in number of lines of code. It is probably expected to have a higher number of committers and commits for bigger systems.

For the second question (RQ2), the proposition is that the relationships between development effort (e.g., number of commits, committers, etc.) and quality measures (as investigated in RQ1) could vary alongside the project lifecycle.

- $\boldsymbol{H_{0_{RQ2}}}$: There are no different correlation values among development effort and code quality measures in different releases.
- $\boldsymbol{H_{1_{RQ2}}}$: $\neg H_{0_{RQ2}}$

The main hypothesis in this case is based on the idea that we could find patterns during project lifecycle in which the relationships are different. The assumption here is that different development efforts with different goals and motivations can be carried out during the project lifecycle. Let us imagine that a development team discovers that they have an upward trend concerning the cloning ratio. In this hypothetical case, they might decide to accomplish a reduction of cloning through code refactoring. During releases in which this refactoring goal is in the developers' minds, some quality measures could revert the upward trend, and even if the total of commits and committers is higher, cloning could experiment an opposite correlation during these releases. The detection of these different configurations during project lifecycle is the main rationale of RQ2 and these hypotheses.

5.6 Variables

Various measures are considered (which are organized in 3 concepts) to answer the research questions. Table 1 summarizes all the variables. For each variable is defined: (i) the concept to which the variable belongs (releasing, development effort, code quality) as defined in background (cf. Section 2.1); (ii) the variable name; (iv) if the variable is independent or dependent; (iv) the scale (i.e., interval, ratio, nominal or ordinal); (v) the range definition for the possible values; and finally (vi) if the variable comes from GitHub, SonarCloud or is a computed measure derived from others.

Table 2: Concept and measure definitions (Type: I – independent, D – dependent; Origin: G – GitHub, S – SonarCloud, D – derived value)

Concept	Variable	Type	Scale	Definition or Range	Origin
Releasing	Project	I	Nominal	Project name	G/S
	Release version	I	Nominal	Release branch number X.Y.Z different for every project	G
Development Effort	Commits	D	Interval	$x = \frac{\#commits}{LOC}$ $x \in \mathbb{R}$	G/D
	Committers	D	Interval	$x = \frac{\#committers}{LOC}$ $x \in \mathbb{R}$	G/D
	Changes by commits	D	Interval	$x = \frac{\#changes}{\#commits}$ $x \in \mathbb{R}$	G/D
	Committers weight	D	Ratio	$x = \frac{\#committers}{\sum_{c \in \{committers\}php} \left(\frac{\#commits}{\#commits_c}\right)}$ $x \in \mathbb{R}$, $x \in [0, 1]$	G/D
Code Quality	LOC	D	Interval	Number of lines of code, $x \in \mathbb{N}$	S
	Bugs	D	Interval	$x = \frac{\#bugs}{LOC}$ $x \in \mathbb{R}$	S/D
	Code Smells	D	Interval	$x = \frac{\#code\ smells}{LOC}$ $x \in \mathbb{R}$	S/D
	Complexity	D	Interval	cyclomatic complexity, $x \in \mathbb{N}$	S
	Violations	D	Interval	$x = \frac{\#violations}{LOC}$ $x \in \mathbb{R}$	S/D
	Duplicated lines	D	Interval	$x = \frac{\#duplicated\ lines}{LOC}$ $x \in \mathbb{R}$	S/D
	Open issues	D	Interval	$x = \frac{\#open\ issues}{LOC}$ $x \in \mathbb{R}$	S/D

5.7 Data collection methods

The raw data of the case study is collected from GitHub and SonarCloud and are then integrated into a common case study database according to the data model presented in Section 2.2.

Both data sources, GitHub and SonarCloud, are widely adopted open-source platforms that can be systematically queried through public API based on RESTful web services. In order to consume the API endpoints, we coded two client programs in Python. In order to ensure the replicability of the case study, these programs are online available in this repository [Pérez-Castillo, 2020]. Both APIs provides textual results based on JSON format. Because of that and the expected data volume, we decided to manage imported data with a Mongo database. Mongo is a NoSQL database

management system and, for our purpose, it makes the integration and management of collected data simpler.

5.7.1 GitHub data collection

GitHub[1] site provides the API information[2]. Algorithm 1 illustrates the client written in Python. First, a list of commits is retrieved with general information (step 1). Then, detailed information is individually queried (step 3). The information regarding a single commit coming from the queries in steps 1 and 3 is integrated and made persistent in Mongo (step 4). It should be noticed that the length of the list of commits is limited by GitHub, so it has to be queried by chunks, named also as page, since in somehow results are accesses with some kind of pagination. Also, although it has been omitted in Algorithm 1 by simplicity, the rate limit has to be queried in the loop to ensure the algorithm can continue do requests to GitHub API. These rate limits are common in public API to guarantee certain service levels for every user. Table 3 shows the endpoint URLs used in step 1 and step 3 respectively, as well as the parametrization we used.

Algorithm 1: Pseudocode for GitHub data retrieval

```
step 1. Query next commits page with commits' general information
step 2. For every commit in commit page
step 3.         Query detailed info for the commit
step 4.         Store commit info
step 5. Are there more commits pages? If yes, go to Step 1. Else, go to
        Step 6
step 6. End
```

Table 3: GitHub API endpoints used to retrieve development effort information

Commits' General info Endpoint	https://api.github.com/repos/<owner>/<project>/commits ?page=<page>&per_page=<page_size>
owner	Name of the user or organization in GitHub
project	Name of the project for a certain organization in GitHub
page	Number of result page to be accessed
page_size	Number of total commits to be retrieved by page. Page size up to 100.
Detailed Commit info Endpoint	https://api.github.com/repos/<owner>/<project>/commits/<coimmit_sha>
owner	Name of the user or organization in GitHub
project	Name of the project for a certain organization in GitHub
commit_sha	It is the commit hash used in git to identify commits in a unique way

[1] https://github.com/
[2] https://developer.github.com/v3/

5.7.2 SonarCloud data collection

SonarCloud[3] also provides information for its API[4]. Algorithm 2 shows how the coded client for SonarCloud operates. First all project analyses and attached events are queried in step 1 to 3 (see the first endpoint in Table 4). SonarCloud offers information about analyses, i.e., explicit actions for analysing source code as is in the code repository in that moment. This retrieves general information, for example, who and when the analysis was performed, but nothing about measuring is retrieved yet. Step 5 then queries project's measures (see the second endpoint and its parameters in Table 4). Before doing this, all metric keys have to be queried first in step 4. Metric keys represent the available metrics in SonarCloud that can be retrieved (e.g., bugs, code smells, duplicated lines, etc.). These metric keys are used as mandatory parameter in step 5. The result set obtained in step 5 is a list with all the metric names, which contains in turn a list with the history of all the measure values according to the analyses performed. All these values are persisted in Mongo (see step 6). As for GitHub, it is necessary to manage the pagination, which is represented in step 7.

Algorithm 2: Pseudocode for SonarCloud data retrieval

```
step 1. Query project analyses
step 2. For every analysis in project analyses
step 3.        Store analysis info
step 4. Query all metric keys
step 5. Query next project measures page for available metric keys
step 6. Store all project measures
step 7. Are there more project measures pages? If yes, go to Step 5. Else,
        go to Step 8
step 8. End
```

Table 4: SonarCloud API endpoints used to retrieve development effort information

Endpoint for searching project analyses	https://sonarcloud.io/api/project_analyses/search?project=<project>
project	Name of the SonarCloud project to be queried.
Endpoint for searching measures history of a project.	https://sonarcloud.io/api/measures/search_history?component=<project>&metrics=<metrics>&p=<page>&ps=<page_size>&from=<from>&to=<to>
project	Name of the SonarCloud project to be queried.
metrics	Comma-separated list of metric keys
page	1-based page number
page_size	Number of measures per page. It must be between 1 and 1000
from	Filter measures created after the given date (inclusive).
to	Filter measures created before the given date (inclusive).
Endpoint to get information on automatic metrics	https://sonarcloud.io/api/metrics/search?ps=<page_size>
page_size	Number of metrics per page. It must be between 1 and 500.

[3] https://sonarcloud.io/
[4] https://sonarcloud.io/web_api

5.7.3 Data Pre-processing

After raw data is collected and stored into the database, it has to be pre-processed before it can be analysed in a way that can help to answer our research questions. This pre-processing is automatically done through another python script we coded. It consists of aggregating data from commits and committers for every release (system version) as well contrast it with code quality measures that are produced in the same period for the SonarCloud analyses retrieved. As a result, a CSV (Coma-Separated Values) file is generated with values, in a tabular layout, for all the variables depicted in section 5.6. It should be noticed that some variables are computed (i.e., derived from values of other variables), which is also done through this pre-processing script.

5.8 Data analysis methods

The study relays on a set of statistical methods, both qualitative and quantitative, as these have previously been depicted in section 2.3. The proposed analysis methods are performed in R. R is a programming language and a statistical suite all together. R is highly flexible and can be customized with many statistical tools (also third-party extension) [Cano, Moguerza et al., 2015]. R has reproducible research and literate programming capabilities, i.e., data analysis can be integrated within reports that can be rebuilt, so these are reproducible by ourselves or by third parties. Therefore, all the analysis performed are integrated into a R markdown file that is online available at [Pérez-Castillo, 2020].

5.9 Quality control and assurance

We have considered four mechanisms to ensure certain levels of quality int the multiple case study:

- Some external peers have been requested to review a draft of the case study design.
- We conducted a pilot case study to assess a preliminary case study design. The pilot study analysed two single cases, projects P6 and P8. All data collection and statistical and clustering analyses were applied to ensure all the expected data can be gathered and analysed as it was preliminary designed.
- In parallel with the case study protocol execution, the actual progress of the case study is reviewed against the planned progress to figure out possible deviations. Additionally, a thorough review was accomplished after data collection and storage steps for each of the 13 open-source development projects.
- In order to assure the reproducibility of the multiple case study, we decided to generate the analysis results as an R markdown that is available for the research community.

6 Result Analysis

After the execution of the case study, the collected data is analysed to draw chains of evidence that help to answer research questions RQ1 and RQ2. All the experimental

data are available online at [Pérez-Castillo, 2020], so that research community can replicate the study or the whole dataset is used for future research.

6.1 RQ1. Development effort against code quality

According to RQ1, it is hypothesized that the more commits and committers, the worse quality levels in software. Figure 4 shows the box plots with the commits and committers distribution by releases for each project. This reflects that there is a huge variability. There are some projects that counts with few contributors and a reduced number of commits, while there are other with higher volume of commits and committers (up to 600 and 24 respectively in absolute terms). Also, it can be realized that there are projects like *dss* (P2) with few committers and many commits, while there are others like *monica* (P6) with more committers and relatively few commits. It demonstrates that the defined variable committers weight is important to consider relative contribution of individuals. Despite these differences, Figure 5 shows that the more committers, the more commits. Actually, according to the density area, it shows that most of the projects vary between 0 and 0.2 committers for every thousands of lines of source code, and up to 2 commits per release and KLOC.

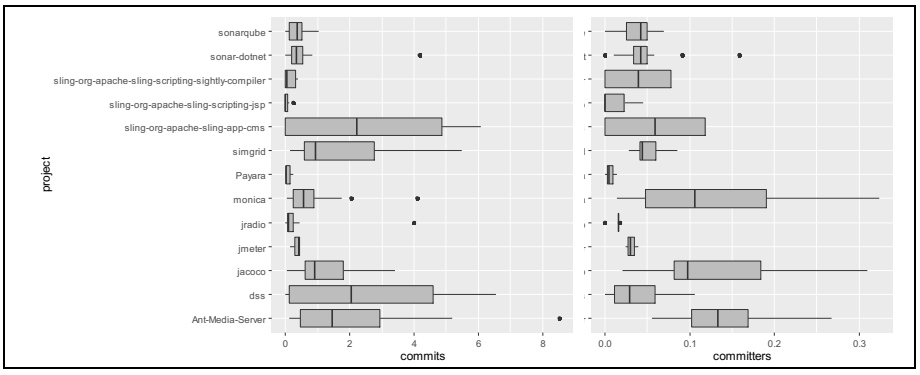

Figure 4: Distribution of commits and committers by project releases (normalized by KLOC)

After checking how the development effort is distributed, Figure 6 summarizes the correlation tests performed between all the variables (see background scale at right side). Coloured cells represent correlations that are statistically significant. Green colour in correlation cells means a negative correlation and it is used (in contrast with red) because most of the used measures represents better software quality levels for lower values (e.g., number of bugs, code smells, complexity, among other). The value in every cell is the Spearman's correlation value. First, we can show that some of the code quality variables are extremely correlated. It might be guessed before this study, but this is an important insight since it demonstrates that bugs, code smells, violations and open issues are related, i.e., these improves or degrade together. However, complexity and cloning do not show the same relation with other code quality. Regarding commits and committers, we can observe there is a certain positive correlation with quality measures (see Figure 6). In particular, complexity is reduced for higher number of commits and committers, and duplicated lines is also reduced with

more committers. We can observe that the size of commits (changes by commit) has a negative impact in code smells, violations and open issues.

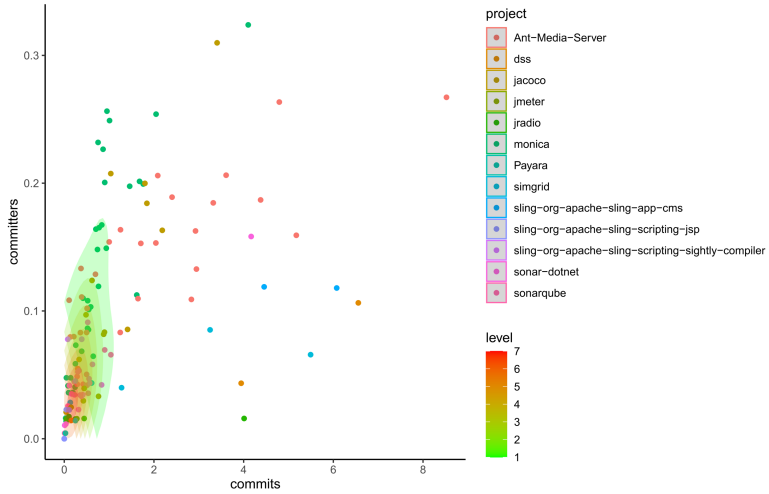

Figure 5: Density of commits-committers scatter plot

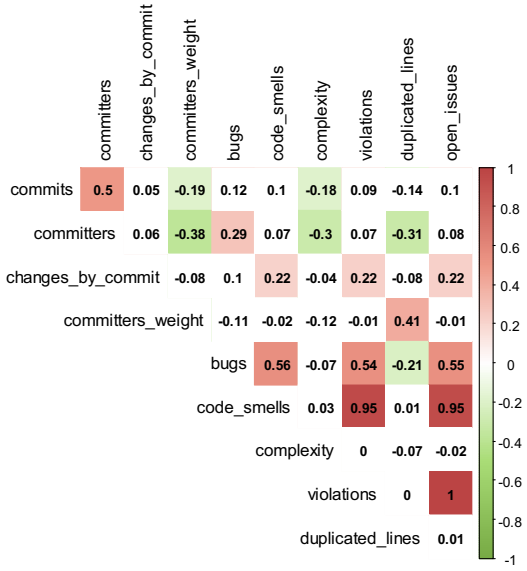

Figure 6: Density of commits-committers scatter plot

Other interesting insight regarding the committers weight is its relationship with duplicated lines. This signifies that releases in which few committers accumulating most of the commits could lead to higher cloning rates.

Before providing an assertive answer for RQ1, despite the correlation found between development effort and code quality measures, these correlations are weak in most of the cases and others do not present statistically significant correlations. As result, differences between projects should be analysed. Figure 7 shows some individual scatter plots and correlations values for four specific projects and certain variables.

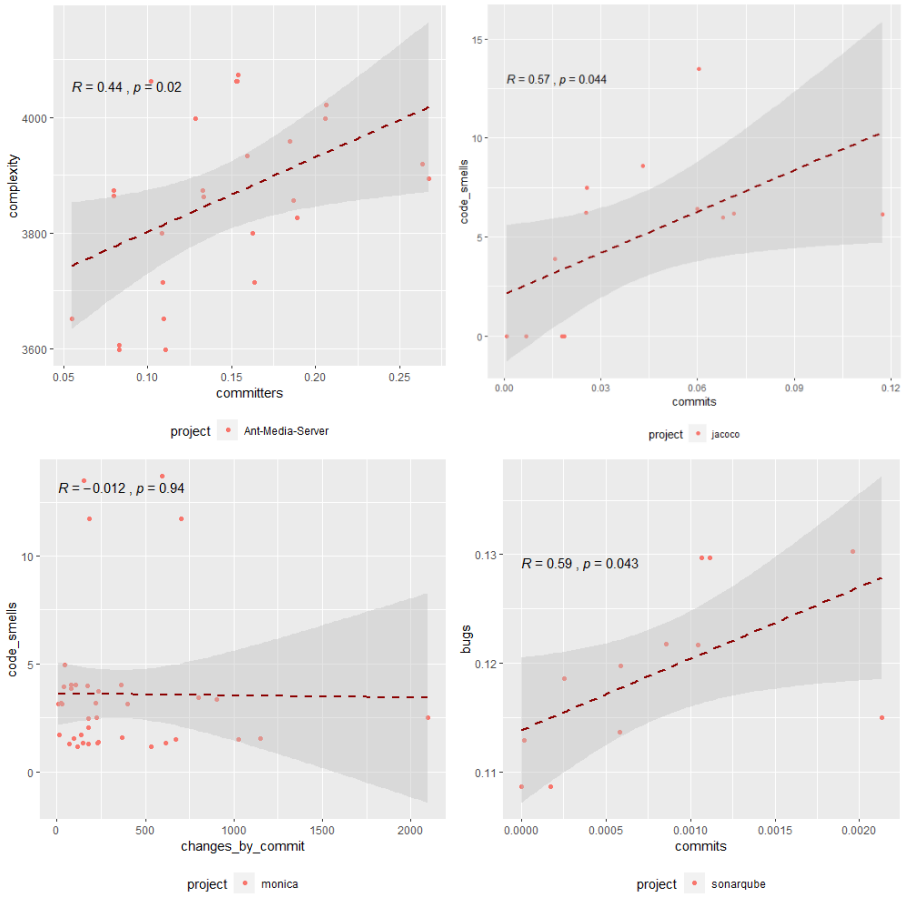

Figure 7: Individual correlations for some variables in specific projects

First, the top-left plot in Figure 7, evaluates the correlation between committers and complexity for the project *ant* (P1). It shows a certain correlation, with R=0.44. This correlation is statistically significant and is against the overall results (see Figure 6) which reported a negative correlation with R=-0.03. Second, the top-right plot in Figure 7 shows another positive correlation for the project *jacoco* (P3). In this case, between commits and code smells. This shows a stronger correlation (R=0.57) than is also against the aggregated results that do not show a correlation between this pair of variables. Thus, it demonstrates that the number of commits can influence the density

of code smells in some project releases. Another counter example is shown in the bottom-left plot in Figure 7 for the project *monica* (P6). It certainly presents the absence of correlation among changes by commits and code smells, while the overall results in Figure 6 show at least a weak correlation. Finally, the bottom-right plot in Figure 7 shows the scatter plot and correlation value for commits and bugs for the project *sonarqube* (P13). It shows a strong positive correlation (i.e., the more commits, the more bugs found) while the general result does not show the same.

To conclude the analysis of RQ1, we can state that $\boldsymbol{H}_{\mathbf{0}_{RQ1}}$ can be rejected, since there is difference in the quality software measures for different numbers of commits and committers. However, this affirmation has to be carefully considered. While some projects exhibit strong, positive correlation between the factors analysed and the code quality measures, other projects do not report conclusive results. This is in line with results presented by [Norick, Krohn et al., 2010] and [Voulgaropoulou, Spanos et al., 2012] that did not find significant evidence in the correlation study of the number of committers and some code metrics, although the quality metrics in those studies were different. For this reason, the analysis of RQ2 through the clustering algorithm is useful to complement the answer provided for RQ1.

6.2 RQ2. Patterns in code quality management

The goal of this question is to figure out the trends or patterns of code quality measures regarding different development effort configurations. As we explained before, the clustering algorithm used is K-means. Since the number of clusters must be defined before for this algorithm, we computed first the optimal number of clusters. We looked for a bend or elbow in the sum of squared error plot. The location of the first elbow in the resulting plot suggests a suitable number of 4 clusters for the K-means algorithm.

The K-means algorithm is then executed and every row in the dataset under analysis (identified by project and release) is annotated with the cluster id (1 to 4). Clusters 2 and 4 agglomerate most of the releases with 36 and 78 respectively, while clusters 1 and 3 group only 27 and 15 releases. After this, the correlation plots are executed again for every cluster dataset (see Figure 8). These plots show clear differences for the four clusters. These differences are explained in the next paragraphs.

Cluster 1 groups 27 releases with most of them belonging to projects P5 and P6. The correlation plot in Figure 8 shows that the more commits and the more committers, the more code smells are detected. However, higher numbers of committers are related with lower values of complexity. Regarding duplications, larger commits seem to lead to higher cloning density. It might be due to larger commits being locally developed by developers have higher probability to be overlapped after all those changes are committed. Results also show that releases with few committers, that accumulate most of the commits, lead to lower cloning density. These insights about cloning are aligned with those reported by [Perez-Castillo, Piattini et al., 2018].

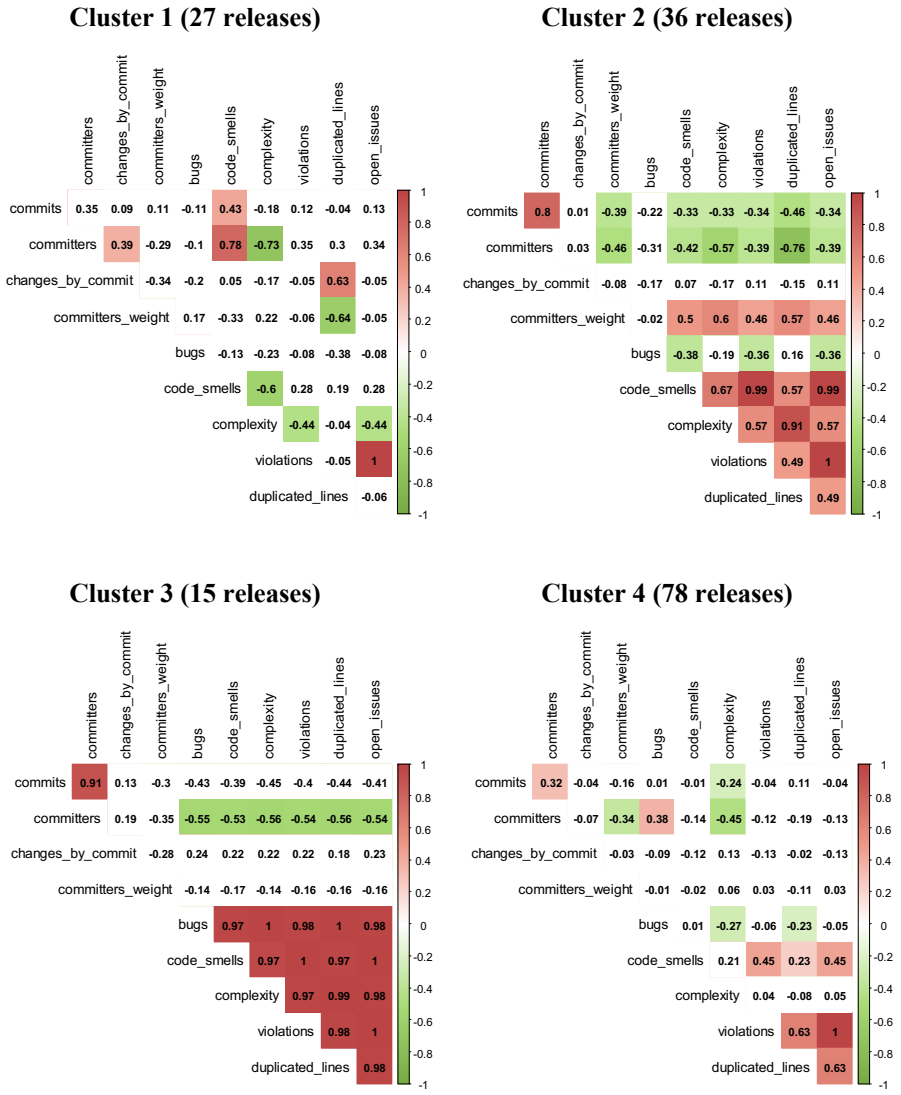

Figure 8: Correlation plots for the four K-means clusters

Cluster 2 (see Figure 8) groups 36 releases. Releases of this clusters correspond with most of the projects' releases of P1 and P10. This cluster is probably the most interesting one. It shows, how the number of commits and committers is inversely related to some of the quality measures such as code smells, complexity, violations, duplications, or open issues. Additionally, cluster 2 shows that most of the code quality measures are correlated except for bugs. It reports that the more bugs, the fewer bad smells, violations, and open issues. A possible explanation for these results lies in the fact that the development team might have decided to accomplish direct effort to

improve software quality. In this case, most of the commits would be related to refactoring and bug fixing tasks instead of adding new functionality. To support this idea, we present the evolution of code smells, duplicated lines, and committers during the releasing history of project P1 (see Figure 9). The releasing history exhibits how upward trends in the number of committers corresponds with downward trends in code smells and duplications.

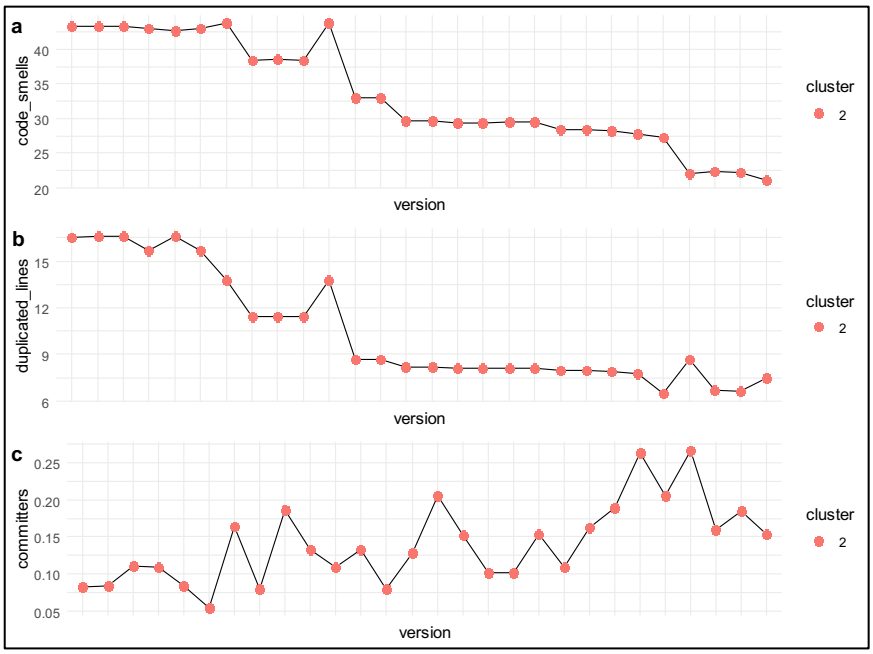

Figure 9: Code smells, duplicated lines, and committers evolution through releasing history of project P1 (ant)

Cluster 3 (see Figure 8) groups 15 releases (the smaller one), and it aggregates releases mainly from project P13, and some from project P7. Correlations presented by this cluster are slightly similar at those of cluster 2. First, the cluster shows that all the code quality measures are correlated. It was already mentioned for general (non-clustered) results in RQ1. However, correlation values of this cluster differ from general ones in the stronger correlation concerning commits and committers. The number of committers regarding quality measures presents a strong positive correlation. However, the number of commits and committer weight do not present correlations as it happened in cluster 2.

In particular, Figure 10 shows the scatter plot for code smells and committers for all project releases aggregated by clusters. While the density of code smells is positively correlated with committers for cluster 1, an inverse correlation happens for cluster 2 and 3. This means code smells are reduced with higher number of developers in project releases of those clusters. In contrast, cluster 1 does not presents a significant correlation.

Finally, Cluster 4 (see Figure 8) groups 78 releases, which means it is the biggest one. This clusters consists of some releases for a wide variety of the projects analysed. It shows that code complexity is directly related with commits and committers. In particular, higher number of commits and committers can reduce the complexity of source code. However, the more committers, the more bugs reported.

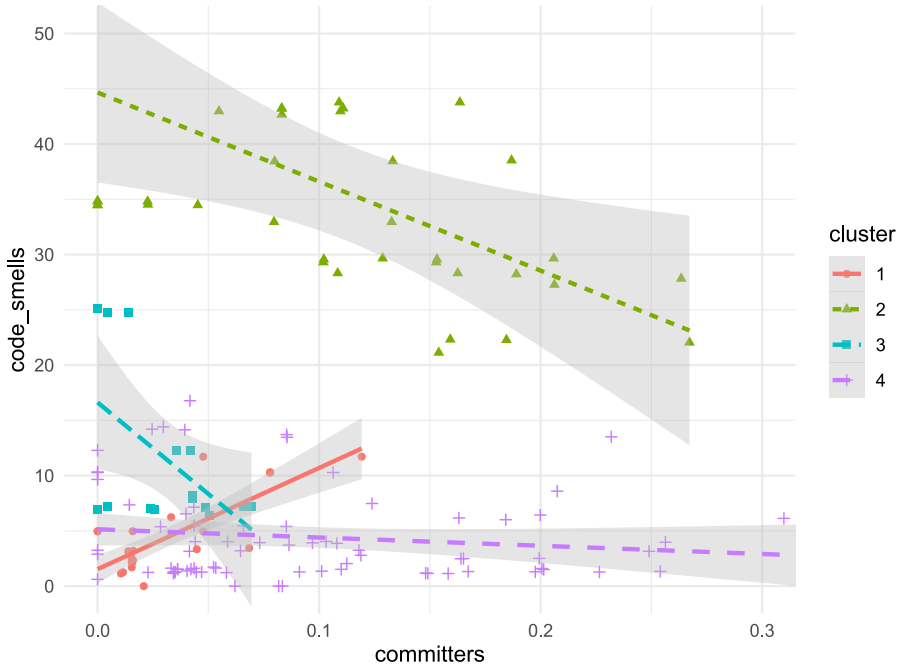

Figure 10: Scatter plot with clustering distinction for code smells and committers

In order to provide an answer to RQ2, we can conclude that the null hypothesis is rejected, so $\boldsymbol{H}_{1_{RQ2}}$ has to be accepted. This signifies that there are specific trends or patterns in the evolution of code quality measures. We have extracted various patterns from the identified clusters.

- **Commits**. The density of commits generally does not affect code quality measures. In some project releases (23%), higher number of commits might influence a better-quality level. We guess that development team is worried about code quality measures and trigger specific efforts to improve quality measures.
- **Committers**. This behaves in a similar way to the commit density. It generally does not influence quality, although in this case up to 33% of project releases are affected by committers in the same way, i.e., the more developers, the better quality. The negative counterpart is that around 17% of project releases

experimented further code smells with more contributors, and up to 50% of releases reported more bugs with more developers.

- **Changes by commit**, i.e., the commit size affects the cloning ratio in at least 17% or project releases.
- **Committer weight**. Up to 23% of project releases the quality measures become worse when few developers were in charge of most of the commits. It makes sense if we imagine that in this context there is a tinny development team with not enough developers to undertake compensatory actions against other. Thus, bad practices are easy to be quickly adopted by tiny teams. In other releases (up to 17%), higher committer weights led to reduced cloning ratio.

On the one hand, the obtained results show that some patterns are aligned with results of some previous work where a significant difference in code quality regarding number of committers cannot be demonstrated [Norick, Krohn et al., 2010, Roehm, Veihelmann et al., 2019, Voulgaropoulou, Spanos et al., 2012]. On the other hand, the correlation for other clusters was stronger as other work previously suggested [Perez-Castillo, Piattini et al., 2018]. Thus, the main insight is that the strategy for software development or maintainability that is followed in every moment of the project life cycle may affect the relationship between the project effort and software quality.

6.3 Evaluation of validity

The multi case study has some threats to the validity that must be discussed to ensure results are reliable. We follow the classification of threats to the validity presented in [Runeson, Host et al., 2012], i.e., construct validity, internal validity, external validity and reliability.

6.3.1 Construct validity

The concepts and measures we used in the study were appropriate for finding answers for the defined research questions. Nonetheless, other factors might affect those measures and need to be discussed. Among these factors we highlight team diversity measured by country and its associated time zone, developer skills, among other [Vasilescu, Filkov et al., 2015]. Other important factor is the programming languages used in each project, since it can affect the way how the developers complete their tasks and how they commit their changes. These factors together with other possible ones have been purposely placed outside of the scope of this research, in order to focus on some specific ones.

This paper also investigates the configuration of developer efforts, which is based on two direct factors (number of commits and committers) plus other derived measures. These measures might be not considered as the configuration of developer effort in some open-source projects since anyone can contribute to the project. In most of the open-source projects there is not a clear role of manager who configure or control the number of commits and contributors. This threat can be mitigated in two ways: adding further projects coming from the industry, and adding further measures associated with the project development effort configuration.

6.3.2 Internal validity

We consider a multiple case study with 13 systems and 156 subunits of analysis, that corresponds with the aggregation of 95,000 commits and 782 analyses. Although there are in literature studies with broader populations [Jarczyk, Gruszka et al., 2014], the case selection was not trivial because of the difficulty of finding projects that are in both repositories with enough releases and quality analyses performed (SonarCloud became popular later than GitHub). Even though, we believe the selected cases represents a certain population with enough statistical representativeness and some specific trend for the proposed measures have been identified in this preliminary case study. As a result, we suggest extending this preliminary study with further open-source projects and combine the results of these studies through meta-analysis.

Furthermore, another threat must be mentioned. Since code quality measures have been collected from SonarCloud, the correctness of these measures must be supposed. Also, the way in which these quality measures are computed depends on that tool, which defines different set of rules that are checked to count number of bugs and violations. Actually, these rule sets can be parametrized by project managers in SonarCloud. For example, some rules could be activated by some projects while remain deactivated for others. Also, the number of duplicated lines varies between projects, since it has certain sensibility to the programming language.

Moreover, in order to analyse the evolution together with the variation of the measures, they were aggregated for each project release version. Alternatively, these variables might be aggregated through, for example, every fixed period of time or other kind of temporal series. This may allow to have further insights.

Other important issue is the fact that the information of most of the projects under study is not fully retrieved since it is not available for all releases. Various projects were migrated to GitHub and SonarCloud at the mid of its lifecycle. To mitigate this, future replications could include (as a selection criterium) the necessity of having all the releases available in GitHub and SonarCloud. In this study, it was certainly difficult to find projects that fulfil all the defined selection criteria and also met the mentioned constraint.

6.3.3 External validity

Concerning the result generalisation, the multiple case study could be generalized to open-source projects, specifically those available in GitHub and SonarCloud at the same time. In order to expand this generalization, we attempt to select projects with different characteristics, e.g., different programming languages, sizes, and domains (see Table 1)

Anyway, in order to attain a broader generalisation, it is necessary to analyse projects stored in other platforms apart from GitHub and SonarCloud and even business projects that are not open source.

In turn, alongside with the generalisation, it would be important to distinguish additional factors during the result analysis. For example, different sectors or domain in which the software is used, or the type of software such as end-user software, utility back-end components, etc.

6.3.4 Reliability

Reliability attempts to determine if the collected data and the performed analysis are dependent, or not, on the researchers. We firmly believe that this study can be conducted by other researchers and obtain the same results. To ensure this, we provide a web page showing the entire experimental material [Pérez-Castillo, 2020]. This web page includes the raw and derived data, the analysis results integrated into the R markdown script, as well as the Python scripts for collecting data. Even more, in order to replicate the study, the source code and code repository information could be directly accessed through GitHub, as well as the quality analysis measures are available in SonarCloud.

7 Conclusions

The assurance and control of software quality has been extensively investigated in the literature. The majority of that research focuses on the study of software product and software process quality, while the impact of software development context is quite often neglected. This study tries to figure out some correlation relationships between the software development effort evolution and code quality measures.

The presented case study analysed thousands of commits and quality measures from 13 open-source projects. The main conclusion is that both, the number of committers and commits affect most of the quality measures analysed. However, there are a specific trend in some of the project releases where this relationship is inverted for the number of commits. A possible explanation is that the development team decided to perform particular efforts to improve code quality instead of simply adding new functionality, although this has to be demonstrated.

The main lessons learned of the study is that during project lifecycle the relationship between the team efforts (and the way the commit changes) and software quality can vary. As a consequence, software developers and project managers should be eager to know in which configuration the project is in every moment according to the patterns analysed in this study. Having this acknowledgement, software developers could commit changes in a different manner, and project managers may make better decisions. For example, when software quality is dramatically degraded in a development project, the common decision made by many managers is still to add more contributors to the development team in order to improve quality. However, the insights of this study suggest that might be better to stop (or reduce) adding new functionality and focus on improving code quality with no additional contributors could be better.

Sometimes, the problem of that common practice in the industry not only lie in the fact of adding more developers, but also in the matter of most added developers have a junior profile in order to keep project costs. Unfortunately, these low-skilled developers will probably add more quality bugs and code smells. So, teams with fewer high-skilled developers that accumulate most of the commits may lead to software with better quality levels. These thoughts have not been covered by this study anyway.

As a future research, we will analyse in-depth these and other important factors in the context of software development projects as we suggested in the validity evaluation section. Also, it is necessary to extend this research to proprietary software projects developed in private companies. Although this will entail many difficulties due to the

opacity of private companies, such projects should be integrated to get broader generalisation and avoid this limitation that is common to many similar studies.

Acknowledgements

This study has been partially funded by the projects GEMA (SBPLY/17/ 180501/ 000293) and SOS (SBPLY/17/ 180501/ 000364) funded by the 'Dirección General de Universidades, Investigación e Innovación – Consejería de Educación, Cultura y Deportes; Gobierno de Castilla-La Mancha'. This work is also part of the projects BIZDEVOPS-Global (RTI2018-098309-B-C31) and ECLIPSE (RTI2018-094283-B-C31) funded by Ministerio de Economía, Industria y Competitividad (MINECO) & Fondo Europeo de Desarrollo Regional (FEDER).

References

[Abrahao, Baldassarre et al., 2016] Abrahao, S., Baldassarre, M. T., Caivano, D., Dittrich, Y., Lanzilotti, R. y Piccinno, A. (2016). Human Factors in Software Development Processes: Measuring System Quality (PROFES 2016). 16th International Conference on Product-Focused Software Process Improvement. Bolzano, Italy, Springer International Publishing. Lecture Notes in Computer Science 9459: 691-696.

[Baggen, Correia et al., 2012] Baggen, R., Correia, J. P., Schill, K. y Visser, J. (2012). "Standardized code quality benchmarking for improving software maintainability." Software Quality Journal 20(2): 287-307.

[Bird, Nagappan et al., 2011] Bird, C., Nagappan, N., Murphy, B., Gall, H. y Devanbu, P. (2011). Don't touch my code! examining the effects of ownership on software quality. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. Szeged, Hungary, Association for Computing Machinery: 4–14.

[Borrego, Morán et al., 2019] Borrego, G., Morán, A. L., Palacio, R. R., Vizcaíno, A. y García, F. O. (2019). "Towards a reduction in architectural knowledge vaporization during agile global software development." Information and Software Technology 112: 68-82.

[Cano, Moguerza et al., 2015] Cano, E. L., Moguerza, J. M. y Corcoba, M. P. (2015). Quality Control with R. An ISO Standards Approach, Springer.

[Chunli and Rongbin, 2016] Chunli, S. y Rongbin, W. (2016). Research on Software Project Quality Management Based on CMMI. 2016 International Conference on Robots & Intelligent System (ICRIS): 381-383.

[Coelho, Valente et al., 2020] Coelho, J., Valente, M. T., Milen, L. y Silva, L. L. (2020). "Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects." Information and Software Technology 122.

[Czerwonka, Nagappan et al., 2013] Czerwonka, J., Nagappan, N., Schulte, W. y Murphy, B. (2013). "CODEMINE: Building a Software Development Data Analytics Platform at Microsoft." IEEE Software 30(4): 64-71.

[Dam, Pham et al., 2019] Dam, H. K., Pham, T., Ng, S. W., Tran, T., Grundy, J., Ghose, A., Kim, T. y Kim, C. (2019). Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR): 46-57.

[Gao, Li et al., 2020] Gao, T., Li, T., Zhu, R., Jiang, R. y Yang, M. (2020). A Research About a Conflict-Capture Method in Software Evolution. The 8th International Conference on Computer Engineering and Networks (CENet2018). Q. Liu, M. Mısır, X. Wang and W. Liu. Cham, Springer International Publishing: 530-537.

[Gautam, Vishwasrao et al., 2017] Gautam, A., Vishwasrao, S. y Servant, F. (2017). An Empirical Study of Activity, Popularity, Size, Testing, and Stability in Continuous Integration. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR): 495-498.

[Ghotra, McIntosh et al., 2017] Ghotra, B., McIntosh, S. y Hassan, A. E. (2017). A Large-Scale Study of the Impact of Feature Selection Techniques on Defect Classification Models. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR): 146-157.

[Greiler, Herzig et al., 2015] Greiler, M., Herzig, K. y Czerwonka, J. (2015). Code ownership and software quality: a replication study. Proceedings of the 12th Working Conference on Mining Software Repositories. Florence, Italy, IEEE Press: 2–12.

[Güemes-Peña, López-Nozal et al., 2018] Güemes-Peña, D., López-Nozal, C., Marticorena-Sánchez, R. y Maudes-Raedo, J. (2018). "Emerging topics in mining software repositories." Progress in Artificial Intelligence 7(3): 237-247.

[Harder, 2013] Harder, J. (2013). How multiple developers affect the evolution of code clones. Software Maintenance (ICSM), 2013 29th IEEE International Conference on, IEEE: 30-39.

[Hayat, Rehman et al., 2019] Hayat, F., Rehman, A. U., Arif, K. S., Wahab, K. y Abbas, M. (2019). The Influence of Agile Methodology (Scrum) on Software Project Management. 2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD): 145-149.

[Hoang, Dam et al., 2019] Hoang, T., Dam, H. K., Kamei, Y., Lo, D. y Ubayashi, N. (2019). DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR): 34-45.

[Humphrey, 2005] Humphrey, W. S. (2005). PSP: A Self-Improvement Process for Software Engineers, Addison-Wesley Professional.

[ISO/IEC, 2001] ISO/IEC (2001). ISO/IEC 9126-1:2001. Software engineering — Product quality — Part 1: Quality model. https://www.iso.org/standard/22749.html, ISO/IEC.

[ISO/IEC, 2004] ISO/IEC (2004). ISO/IEC 15504-1:2004. Information technology — Process assessment — Part 1: Concepts and vocabulary, International Organization for Standardization.

[ISO/IEC, 2011] ISO/IEC (2011). ISO/IEC 25010:2011.Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models. https://www.iso.org/standard/35733.html, ISO/IEC.

[ISO/IEC, 2015] ISO/IEC (2015). ISO/IEC 33002:2015 Information technology -- Process assessment -- Requirements for performing process assessment.

[Janicijevic, Krsmanovic et al., 2016] Janicijevic, I., Krsmanovic, M., Zivkovic, N. y Lazarevic, S. (2016). "Software quality improvement: a model based on managing factors impacting software quality." Software Quality Journal 24(2): 247-270.

[Jarczyk, Gruszka et al., 2014] Jarczyk, O., Gruszka, B., Jaroszewicz, S., Bukowski, L. y Wierzbicki, A. (2014). GitHub Projects. Quality Analysis of Open-Source Software. Social Informatics: 6th International Conference, SocInfo 2014, Barcelona, Spain, November 11-13, 2014. Proceedings. L. M. Aiello and D. McFarland. Cham, Springer International Publishing: 80-94.

[Jia, Mo et al., 2018] Jia, J., Mo, H., Capretz, L. F. y Chen, Z. (2018). "Grouping environmental factors influencing individual decisionmaking behavior in software projects: A cluster analysis." Journal of Software: Evolution and Process 30(1).

[Joonbakhsh and Sami, 2018] Joonbakhsh, A. y Sami, A. (2018). Mining and Extraction of Personal Software Process Measures through IDE Interaction Logs. 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR): 78-81.

[Kalliamvakou, Gousios et al., 2016] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M. y Damian, D. (2016). "An in-depth study of the promises and perils of mining GitHub." 21(5 %J Empirical Softw. Engg.): 2035–2071.

[Kiehn, Pan et al., 2019] Kiehn, M., Pan, X. y Camci, F. (2019). Empirical Study in using Version Histories for Change Risk Classification. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR): 58-62.

[Kuutila, Mäntylä et al., 2020] Kuutila, M., Mäntylä, M., Farooq, U. y Claes, M. (2020). "Time pressure in software engineering: A systematic review." Information and Software Technology 121.

[Lewis, Lin et al., 2013] Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R. y Whitehead, E. J. (2013). Does bug prediction support human developers? Findings from a Google case study. 2013 35th International Conference on Software Engineering (ICSE): 372-381.

[Manzano, Ayala et al., 2019] Manzano, M., Ayala, C., Gómez, C. y Cuesta, L. L. (2019). A Software Service Supporting Software Quality Forecasting. 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C): 130-132.

[Maxim and Kessentini, 2016] Maxim, B. R. y Kessentini, M. (2016). An introduction to modern software quality assurance. Software Quality Assurance: In Large Scale and Complex Software-intensive Systems: 19-46.

[Namiot and Romanov, 2020] Namiot, D. y Romanov, V. (2020). On Data Analysis of Software Repositories. Convergent 2018. Convergent Cognitive Information Technologies. 1140 CCIS: 263-272.

[Nayrolles and Hamou-Lhadj, 2018] Nayrolles, M. y Hamou-Lhadj, A. (2018). CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects. 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR): 153-164.

[Norick, Krohn et al., 2010] Norick, B., Krohn, J., Howard, E., Welna, B. y Izurieta, C. (2010). Effects of the number of developers on code quality in open source software: a case study. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Bolzano-Bozen, Italy, Association for Computing Machinery: Article 62.

[Papamichail and Symeonidis, 2020] Papamichail, M. D. y Symeonidis, A. L. (2020). "A generic methodology for early identification of non-maintainable source code components through analysis of software releases." Information and Software Technology 118.

[Pérez-Castillo, 2020] Pérez-Castillo, R. (2020). "Experimental Data for Understanding the impact of Development efforts in Software Quality." Retrieved 27/03/2020, 2020, from https://github.com/ricpdc/sonar-git/wiki.

[Perez-Castillo, Piattini et al., 2018] Perez-Castillo, R., Piattini, M. J. J. o. S. E. y Process (2018). "An empirical study on how project context impacts on code cloning." 30(12): e2115.

[Querel and Rigby, 2018] Querel, L. P. y Rigby, P. C. (2018). WarningsGuru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings: 892-895.

[Rodriguez, Tanaka et al., 2018] Rodriguez, A., Tanaka, F. y Kamei, Y. (2018). Empirical Study on the Relationship Between Developer's Working Habits and Efficiency. 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR): 74-77.

[Roehm, Veihelmann et al., 2019] Roehm, T., Veihelmann, D., Wagner, S. y Juergens, E. (2019). Evaluating Maintainability Prejudices with a Large-Scale Study of Open-Source Projects. Cham, Springer International Publishing: 151-171.

[Runeson, Host et al., 2012] Runeson, P., Host, M., Rainer, A. y Regnell, B. (2012). Case study research in software engineering: Guidelines and examples, John Wiley & Sons.

[Saini and Chahal, 2018] Saini, M. y Chahal, K. K. (2018). "Change profile analysis of open-source software systems to understand their evolutionary behavior." Frontiers of Computer Science 12(6): 1105-1124.

[Schranz, Schindler et al., 2019] Schranz, T., Schindler, C., Müller, M. y Slany, W. (2019). Contributors' impact on a FOSS project's quality. SQUADE 2019: Proceedings of the 2nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies. Tallinn Estonia, Association for Computing Machinery: 35-38.

[Shrestha, 2018] Shrestha, A. (2018). Towards a taxonomy of process quality characteristics for assessment. 918: 47-59.

[Singh, Chaturvedi et al., 2017] Singh, V. B., Chaturvedi, K. K., Khatri, S. y Sharma, M. (2017). Complexity of the code changes and issues dependent approach to determine the release time of software product. 10408 LNCS: 519-529.

[Vasilescu, Filkov et al., 2015] Vasilescu, B., Filkov, V. y Serebrenik, A. (2015). Perceptions of Diversity on Git Hub: A User Survey. 2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering: 50-56.

[Verma and Kumar, 2017] Verma, D. y Kumar, S. (2017). "Prediction of defect density for open source software using repository metrics." Journal of Web Engineering 16(3-4): 294-311.

[Voulgaropoulou, Spanos et al., 2012] Voulgaropoulou, S., Spanos, G. y Angelis, L. (2012). Analyzing Measurements of the R Statistical Open Source Software. 2012 35th Annual IEEE Software Engineering Workshop: 1-10.

[Wang, Meng et al., 2019] Wang, J., Meng, X., Wang, H. y Sun, H. (2019). An Online Developer Profiling Tool Based on Analysis of GitLab Repositories. 1042 CCIS: 408-417.

[Wong, Yu et al., 2018] Wong, W. Y., Yu, S. W. y Too, C. W. (2018). A Systematic Approach to Software Quality Assurance: The Relationship of Project Activities within Project Life Cycle and System Development Life Cycle. 2018 IEEE Conference on Systems, Process and Control (ICSPC): 123-128.

[Yin, 2014] Yin, R. K. (2014). Case study research: Design and methods, Sage publications.