SINGULAR®
ENGENHARIA, TECNOLOGIA E GESTÃO

Publicação: Outubro de 2019

Digital Object Identifier

Android Applications based on software repository analysis

TAYSE VIRGULINO RIBEIRO[1], MÁRCIO LOPES CORNÉLIO[2]

1-Master of Science in Computer Science - Center of Informatic (CIn), Universidade Federal de Pernambuco - UFPE, Recife/PE - Brazil (e-mail: tvr@cin.ufpe.br)
2- Assistant Professor Centro de Informática · Universidade Federal de Pernambuco PhD in Computer Science, Centro de Informática (e-mail: mlc2@cin.ufpe.br)
Autor Correspondente: Tayse Virgulino Ribeiro (e-mail: tvr@cin.ufpe.br).

Context: Software repositories have been a source for studies about software evolution and its relation to software defects. In addition, the context of repositories have also been used for the purpose of analyzing refactoring practiced by programmers throughout the development process. **Objective**: Our objective is based on android projects stored in software repositories, to determine what types of transformations, that is, which refactoring are used, seeking to relate them to quality and security factors. **Method**: This research uses as an approach an exploratory study of a qualitative character, based on a systematic review of the literature, which will be carried out between the period from 2015 to 2019, as well as application of research and quality criteria regarding the work context. In addition, develop a case study with projects for Android, relating refactoring quality criteria to non-aggregated projects in software repositories, glimpsing comparative and resulting factors. **Expected results**: It is expected with this review an analysis and a summary of existing literature on Code Quality in the process of Software Refactoring for Android projects. **Conclusions**: The research is guided by this approach in identifying the types of refactoring practiced and extracting the related quality factors in the development process. We believe that our results will benefit in the updating and summary of the literature in the context of refactoring, glimpsing comparative factors.

Software projects, Android projects, software refactoring, metrics quality repositories software.

I. INTRODUCTION

The research related to Systematic Review of Literature (SRL) is necessary as software repositories have been a source for studies that establish the relationship between evolution activities on software defects [1], which allow the measurement of contributions from developers [2] in software projects. In addition, repositories have also been used to analyze refactoring practiced by programmers throughout the development process [3]. In the case of mobile platforms, in particular for Android systems, applications have been analyzed with various static analysis tools in order to determine, for example, the excess permissions or potential bugs in different versions [4].

Understanding how software is created and preserved is essential to define how to make it faster, cheaper, and with higher quality. One way to get valuable information about the development process is to analyze existing projects: source code, how the application has evolved over time, and attributes such as security, failure, and size [4].

Different program understanding studies show that programmers rely on good software documentation [5]. In addition to improving, standardizing the quality of documentation and ensuring information security.

Out, the problem of this research is focused on identifying which refactoring are involved in the development process to achieve the software quality factor? Also, what refactoring are related to a specific quality model?

In addition, the work uses as an approach an exploratory study of qualitative character, based on

the accomplishment of a rapid systematic review of the literature. As well, a case study with projects for Android is carried out in the context of software repositories, looking for applicability of quality criteria.

This paper is organized as follows. In Section 2, we introduce fundamental concepts about of software repositories and software quality. In Section 3, we describe the research method we adopt. In section 4 and 5, we present results and considerations of the exploratory study presented. Finally, in Section 6, we present out conclusions.

II. CONCEPTUAL BACKGROUND AND RELATED WORKS

This section presents the main concepts, theories and research challenges directed at the systematic review and the case study of this qualitative research.

A. REFACTORING

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written [6].

"Improving the design after it has been written." That's an odd turn of phrase. In our current understanding of software development, we believe that we design and then we code. A good design comes first, and the coding comes second. Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The code slowly sinks from engineering to hacking [6].

Refactoring is the opposite of this practice. With refactoring you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. You move a field from one class to another, pull some code out of a method to make into its own method, and push some code up or down a hierarchy. Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decay [6].

With refactoring you find the balance of work changes. You find that design, rather than occurring all up front, occurs continuously during development. You learn from building the system how to improve the design. The resulting interaction leads to a program with a design that stays good as development continues [6].

In 1992, Willian Opdyke's thesis was the first work in the context of refactoring. The focus of the thesis was to automate the refactoring in a way that preserved the behavior of a program. This thesis establishes a set of refactoring operations that support the design, evolution and reuse of frameworks of object-oriented applications [17].

Contextualizes the design of reusable software in an especially difficult way [17]. In general, reusable Software is the result of many design iterations. These occur after the software has been reused and the resulting changes affect not only the design but also the design of other software that is using it. Therefore, making the software easier to modify facilitates subsequent design iterations and makes the software more reusable [17].

It is generally addressed in Opdyke's thesis that, for some refactoring, one or more of its preconditions are undecidable. I n this thesis he defines 23 primitive refactoring and shows three examples of compound refactoring. For primitives, a set of preconditions provides the notion of software behavior. The set of complex refactoring is defined in detail: generalization of the inheritance hierarchy, specialization of the inheritance hierarchy and use of aggregations to model relationships between classes [17].

Finally, this thesis explores several operations for applicability in the process of evolution and development of object-oriented applications. As well, it provides some conservative algorithms to determine if a program satisfies these constraints and describes how to use this design information to refactor a program [17].

In 1999, Don Roberts in his doctoral thesis continues the work of Opdyke [17], adding postconditions and developing the first refactoring tool. In his thesis, [18] provides:

> [...] a new definition of refactoring that focuses on preconditions and postconditions of refactorings rather than on program transformation itself. Preconditions are statements that a program must satisfy for refactoring to be applied, and postconditions specify how the assertions are transformed by refactoring. Post-conditions can be used for a number of purposes: to reduce the amount of analysis that subsequent refactorings must perform, derive preconditions for compound refactorings, and calculate dependencies between

refactorings.

In addition to examining techniques to aid refactoring, it presents the Refactoring Browser design, a Smalltalk refactoring tool that is used by commercial software developers and identifies the criteria necessary for any refactoring tool to be successful [18].

Thus, it redefines the refactorings proposed by [17], dividing them into three groups: class refactoring, method refactoring, and variable refactoring [19].

Refactoring, reuse e reality

Refactoring is an overhead activity [6]:

- Tools and technologies are available to allow refactoring to be done quickly and relatively painlessly
- Experiences reported by some object-oriented programmers suggest that the overhead of refactoring is more than compensated by reduced efforts and intervals in other phases of program development.
- Although refactoring may seem a bit awkward and an overhead item at first, as it becomes part of a software development regimen, it stops feeling like overhead and starts feeling like an essential.

How does one safely refactor? There are several options [6]:

- Trust your coding abilities.
- Trust that your compiler will catch errors that you miss.
- Trust that your test suite will catch errors that you and your compiler miss.
- Trust that code review will catch errors that you, your compiler, and your test suite miss.

The real-world concerns regarding a reuse program are similar to those related to refactoring [6].

- Technical staff may not understand what to reuse or how to reuse it.
- Technical staff may not be motivated to apply a reuse approach unless short-term benefits can be achieved.
- Overhead, learning curve, and discovery cost issues must be addressed for a reuse approach to be successfully adopted.
- Adopting a reuse approach should not be disruptive to a project; there may be Strong pressures to leverage existing assets or implementation albeit with legacy constraints. New implementations should interwork or be backward compatible with existing systems.

B. QUALITY OF SOFTWARE

1) Project of software: android

Android has grown to be the world's most popular mobile platform with apps that are capable of doing everything from checking sports scores to purchasing stocks. In order to assist researchers and developers in better understanding the development process as well as the current state of the apps themselves, we present a large dataset of analyzed open-source Android applications and provide a brief analysis of the data, demonstrating potential usefulness [4].

Android has become an extremely popular mobile platform, and Android apps are not immune to the problems which have hindered traditional software — especially security vulnerabilities, high maintenance costs, and bugs. Understanding how software is created and maintained is paramount in determining how to produce it faster, cheaper, and of higher quality. One way to gain valuable insight into the development process is to examine existing projects: source code, how the app has evolved over time, and attributes such as its security, defects, and size. App source code may be analyzed using static analysis tools, providing data about the software's security risk level, possible defects, or even lack of adherence to coding standards [4].

A dataset of Android applications with the results of the static analysis tools we have created is an important tool for understanding how Android applications are developed and maintained [4].

2) Software testing repositories: software bugs

Studying the evolution and understanding the structure of large legacy systems are two key issues in software industry that are being tackled by academic research. These problems are strongly coupled for various reasons: (i) examining the structure of subsystems allows us to gain a better understanding of the whole system evolution; (ii) the histories of software entities can reveal hidden relationships among them and (iii) analyzing several versions of a system improves our understanding of it [7].

3) Repositories of software: excess permissions

According [8], one of the most important principles of good computer security is the principle of least privilege: A user should have no more access to data and systems than is necessary for their task. Too often, security problems result from users having excessive privileges and excessive access to data.

In the area of software refactoring, some of the related works are worth referencing:

Firouzi [12] discusses automated refactoring tools in Visual Studio, analyzing their effectiveness with the success rate of the builds and version control system. This paper examines the actual impact of using ReSharper tools on the results of test runs, Builds and Version control commands to evaluate their impact using some features such as Quick-fixes, Context actions, and Refactorings. In this study, they investigate whether the above tools meet the expectations. In order to obtain the necessary information, Firouzi used the Data Set Enriched Event Flows provided by the MSR challenge in 2017. In the pre-process phase an application was developed that iterated all user zip files and all JSON files and converted the dataset into a relational database, and information stored in the SQL Server database.

It is a study focused on the Enhanced Events Stream Dataset and the presented results may not generalize well in other contexts, regardless of statistical justifications and significance calculations. In addition, other factors may influence the results, but due to the abstraction of the data set or not being collected could not be investigated. The results suggest that the automated refactoring tools of Visual Studio, ReSharper, may not consider an overview of the solutions [12].

In [13] software metrics are employed in the development and maintenance of software to evaluate different quality attributes, software design, test and reengineering processes. In this case, the software metrics used in relation to the project standards went through an analysis process. According to Derezińska [13], they focused on "classical" design patterns (DP, in short) applied to object-oriented software. Different metrics are used in evaluating the impact of the design pattern on selected quality attributes, for example, software maintenance, flexibility to change input, performance, susceptibility to failure, among others. One of the quality models of object-oriented software is QMOOD. Different object-oriented features are associated with quantitative measures such as SIZE, NOC, DIT, DAM, CBO, CAM, MOA, MFA, NOP, RFC, WMPC.

To automate the introduction of design patterns in the existing code, a refactoring process was proposed, this process is based on relevance metrics that support the recommendation of standards. The metrics were implemented in a prototype tool that supports refactoring to design standards. The tool extends the Eclipse environment and transforms Java programs. The main contribution of the paper is a new approach based on metrics for refactoring. The approach can be used as a recommendation presented to a user or can provide partial or fully automated refactoring. Preliminary experiments with the prototype on the refactoring of Java programs to design patterns have confirmed the profitability of the approach [16].

In article [14] the effect of clone refactoring (CR) on the size of unit test cases in object-oriented (OO) software is evaluated empirically. In general, the contributions of this paper are: (1) strong and positive correlation between the CR code and the reduction in the size of the unit test cases, (2) showed how the code quality attributes related to the testability of the classes are significantly improved when clones are refactored (3) the size of unit test cases can be significantly reduced when CR is applied, and (4) complexity/size measures are commonly associated with variations in the size of unit test cases when compared to coupling. The selected source code metrics address the size, complexity, and coupling attributes.

In the methodology was collected data from two open source Java software systems, ANT and ARCHIVA, which were refactored in clones. As reported by Badri [14], to quantify the size of unit test cases, we used two test code metrics already used in many previous empirical studies and the size of unit test cases. In addition, the results were based on three different techniques: single-exit cross-validation (LOOCV), ten-fold cross-validation (10FCV) and cross-project validation (IPCV).

In this article [15] explores better understanding of performance issues in mobile applications by focusing our study on iOS. We conducted an empirical study of 225 performance problem reports on four free software iOS applications written in the Swift programming language to study common types of performance problems. IOS applications are generally developed using the Model-View-Controller (MVC) design pattern. The layers in the MVC help to abstract the underlying device differences, such as screen sizes, and simplify application development. The study of problem reports found that inefficient user interface design, memory problems, and inefficient thread manipulation are the most common types of performance problems. These four anti-patterns were documented and a static analysis tool, called iPerfDetector, was developed to detect these patterns, evaluated in 11 applications in IOS.

In [16], it is discussed the importance of refactoring in software engineering and the difficulties that can be faced with the application of refactoring. In addition to providing an overview of the refactoring

process and discussing the critical components of this practice. Quality metrics are used to discover design flaws in software systems or measure the quality of the code in specific ways. In the literature [16], "Bad Smells" is a term used to describe common structural problems in the code that need to be eliminated to make the code more sustainable. To achieve this, an appropriate refactoring operation needs to be performed. Examples of "Bad smells" metrics: Duplicated code, Long method, Larg class, and Long paramter list.

For the most part, the refactoring deals with the static relationships between the properties of the units in the source code. For small software systems, structure or design can be considered as an aesthetic question. For a larger software design and company design, they become much more important because of their direct impact on cost. Therefore, a higher quality project should be the goal and this project needs continuous refinements to preserve the initial quality against software evolution. We briefly discuss the refactoring process and discuss several examples of "Bad smell", design problems that degrade software quality and proposed refactoring solutions [16].

Four different types of very important tools for successful refactoring have been listed: static analysis tools, visualization tools designed to support refactoring, refactoring tools for performing the refactoring process, and automated testing tools to verify such refactoring operations. Refactoring actually preserve the behavior of the system. The place of refactoring in industry and academia was addressed. In addition to highlighting that refactoring is one of the main practices in agile development processes, such as extreme programming [16].

III. METHOD

In a different direction, this research aims to explore activities based on android projects stored in software repositories, in search of which types of refactorings are used in relation to quality factors in the development and measurement of the code. From this perspective analyze and summary the existing literature, as well as conduct a case study with Android projects, relating to refactoring quality criteria in order to glimpse comparative factors. Our research question is formulated as follows:

Based on android projects, what types of refactoring are used, trying to relate them to quality factors?

From this, the research question of this work is derived from the definition of the following elements:

- *Context: projects for android*
- *Intervention: quality factors*
- *Result: types of refactorings*

This study is based on a systematic review, addressing a qualitative exploratory case study. Figure 1 shows the flow of the methodological proposal that this research will guide.

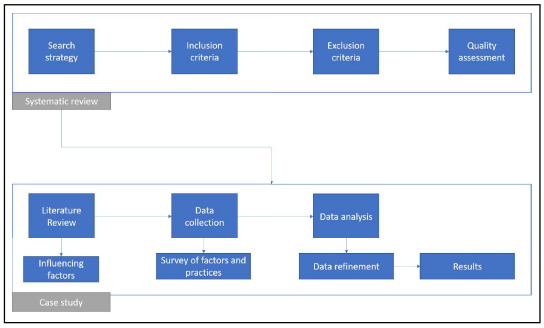

Fig. 1: Methodological flow of research

The research will comprise two phases, as shown in Figure 1, Systematic review and Case study.

In the first stage we understand Systematic review. First, in this process the Search strategy step must be carried out, which will follow the basic scope of definition of the search methods, construction of the search string and definition of the search source. The initial proposal of the protocol is based on a manual search in specific sources to obtain a set of known articles that allow us to measure the level of completeness of our research and help in the construction of our search string that will be used in the automatic search. This research aimed to analyze periodic and events related to Code Refactoring, as well as studies classified as Software Repositories of Mining (MSR), with the help of the use of Springer, IEEE and ACM databases.

Already in the automatic search we have as aid the use of the search engine Scopus. After this process of searching the engines, we be carried out the process of selection of studies. Some inclusion criteria can be performed to filter the most relevant studies for the research.

In the strategy of the research, we still designated some criteria to categorize the study. According to Hulley [9], the inclusion and exclusion criteria are based on a standard and necessary practice in the elaboration of high-quality protocols. In addition, characteristics of the target unit of analysis that researchers will use to answer the study question. While the exclusion criteria include eligible characteristics that make them have a high chance of loss of follow-up.

For this research, the following inclusion criteria will be assigned, taking into account that the researchers seek to identify the maximum of applied trends in code quality and software refactoring process. The protocol of this research chose to include studies written only in English, and not to differentiate in profiles of students and professionals. With regard to the publication time limit, publications from 2011 up to the current year will be included. The inclusion criteria of the studies are presented in Table 1:

1lightgray

Tab. 1: Inclusion Criterion.

Criterion	Inclusion Criterion Description
CI1	Language of the article: English
CI2	Publish date limit: 2011 to 2019
CI3	Primary studies that address the area of research knowledge
CI4	Target population: software engineers, software developers, programmers and related areas of computing.

On the other hand, the studies that do not have a focus on the area of computing and related or that do not have relevance to the research will be excluded. As well, studies without theoretical foundation and that are not related to the research question. As well as articles that were previously published the year 2011. The exclusion criteria for the retrieved studies will be presented in Table 2.

1lightgray

Tab. 2: Exclusion Criterion.

Criterion	Description of Exclusion Criterion
CE1	Articles with no theoretical basis and not related to the research question
CE2	Articles that are not focused on computing or that have no relevance
CE3	Articles before 2011
CE4	Language of the article: Portuguese.

In addition to these criteria, in this process a quality evaluation is performed, that is, a list of criteria necessary to meet the quality of the study. Initially, to carry out this quality analysis, 6 criteria were defined and evaluated according to a scale composed of the following values:

- (0) when the criterion is not met;
- (1) when the criterion is implicit;
- (2) when the criterion is fully met;

The quality criteria (QC) will be listed below 3:

1lightgray

Tab. 3: Quality Criterion.

Criterion	Description of Quality Criterion
CQ1	Is there a clear definition of the objectives of the study?
CQ2	Is there a clear definition of the study's justifications?
CQ3	Is there a clear definition of the study research question?
CQ4	Is there a clear description of the context in which the research was carried out?
CQ5	Does the study provide clear and reliable results?
CQ6	Is the study relevant to research or professional practice?

The criteria presented in Table 3 are defined to evaluate and prove, regarding reliability, quality of the selected studies and why they are required.

After the first step of systematic review, the case study is initiated. This will comprise of the steps of literature review, data collection and data analysis. In the literature review stage, the studies recovered from the systematic review stage will be analyzed according to the main influencing factors of the research, according to the research question. Afterwards, the data collection stage will be carried out in order to be based on a source of evidence, such as physical artifact analysis, in order to analyze tools, instruments or technological devices. In this case, with the objective of collecting information about projects developed for Android relating to quality criteria in code refactoring.

Finally, the data analysis step is performed, which in turn comprises refining the collected data. But also, it is worth mentioning that this step is executed in a parallel way with the collection of data and in a systematic way. In addition, it contributes to conclusions drawn from the data, maintaining a clear chain of evidence. According to [10], in this process the data are codified; the coded material can be combined with comments by the researcher; the researcher uses this material to identify the first set of hypotheses; an iterative approach is performed, so that data collection and analysis are performed in parallel; and a set of generalizations can be formulated from the analysis performed. Finally, a report is generated of the whole process, that is, the result of what was analyzed in the study. The study report is responsible for presenting the results obtained, besides being the main source of information to judge the quality of the study.

IV. THREATS TO VALIDATY

This section presents the set of possible threats, as listed by Wohlin [11] for the study, as well as the actions of controls in order to minimize them.

Listed as threats to internal validity we have: Including only journals articles and events classified by Mining Software Repositories (MSR) in one of the search processes as belonging to software engineering limits the possibility of generalizing the results to other forums in which refactoring technologies are published. Also limiting the selection of articles in other search engines may interfere with a more consistent result. This also introduces the risk of lack of technologies and evaluations published in conference proceedings, reports, workshops, etc. However, since automatic search is performed in a way that does not limit the type of publication classification, other main journals in the area of software engineering may be included in the review, so it is necessary that this threat is limited.

In the article selection process, first, the search

procedure used introduces a threat, since relevant work may be missing for inclusion in the review. Even if the research has lost its work, it should not introduce any systematic bias towards the results. Secondly, the inclusion criterion is applied, the abstracts of the selected studies are read. This introduces a threat because the summary may not necessarily reflect what is actually presented in the articles.

Finally, as threats to external validity, the following are punctuated: Already in the process of extracting the studies, a potential threat to validity is the judgment used to include/exclude a study and extract information from these included studies. With this in mind, in order to limit this threat, a pilot with the classification and inclusion, exclusion and quality criteria can be carried out, and these can be changed before use. In addition, the aspects used for data extraction are derived from the research question. These aspects are subject to variation. To limit this threat, the studies were classified giving the researchers the benefit of the doubt, ie, the studies were classified according to what is mentioned in the articles.

After the systematic review, the case study is started. Thus, during this stage, data collection and analysis phases identified in the previous phase will be performed. Inconsistent collection and analysis of this information may interfere with the extraction of the criteria from the case study phase. This is to generate a report of refinement of the whole process carried out based on the research question.

The threats to validity presented for the protocol of this research are an outline for the study, since it is a proposal, they may change during the application.

V. EXPECTED RESULTS

In this section the expected results related to the development of this research will be presented, in order to discuss the stages of the study carried out based on the steps of the methodology performed in this work.

Based on the studies captured during the review phase, these will be filtered based on the criteria presented above, as presented in Tables 1, 2 and 3. These criteria aim to extract studies with theoretical basis and related to the research question. Therefore, in this phase of systematic review, we hope to extract a summary of the studies related to refactorings in the development process. As well, explore specific quality factors for Android applications.

In addition, it is also expected with the case study stage to present a relationship between the collected studies, with the objective of collecting quality metrics on projects developed for Android. Finally, the data analysis step is performed, which in turn comprises refining the collected data. The accomplishment of this research comprises the analysis of studies in the context of software refactoring. In order to determine which types of transformations, that is, which refactorings are used, trying to relate them to quality and safety factors in the refactoring process.

VI. CONCLUSIONS

We believe that this work can provide an identification of the types of refactoring practiced when developing for the mobile applications area. As well, propose an extraction of the refactoring quality factors related to the development process. We believe that our results will be positive regarding the presentation of an update of the literature review in the context of refactoring, glimpsing comparative factors on the use of refactoring types in android development environments.

In addition, it is expected that the proposal of this work will serve as an input to support the research developed for the context of code refactoring. Therefore, we intend to contribute in a positive way in updating the systematic review, seeking to extract the quality metrics in the process of refactoring android application code.

REFERENCES

[1] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In Proceedings of the 2008 international working conference on Mining software repositories (MSR '08). ACM, New York, NY, USA, 35-38. 2008.

[2] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In Proceedings of the 2008 international working conference on Mining software repositories (MSR '08). ACM, New York, NY, USA, 129-132. 2008.

[3] Gustavo Soares et al. Analyzing Refactorings on Software Repositories. Software Engineering (SBES), 2011 25th Brazilian Symposium on Software Engineering. 2011.

[4] Daniel E. Krutz et al. A dataset of open-source Android applications. In Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15). IEEE Press, Piscataway, NJ, USA, 522-525. 2015.

[5] Paul W. McBurney and Collin McMillan. Automatic Documentation Generation via Source Code Summarization of Method Context. In Proceedings of the ICPC'14, June 2-3, 2014, Hyderabad, India.

[6] Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, don Roberts. Refactoring: Improving the Design of Existing Code. Another stupid release 2002.

[7] Marco D'Ambros and Michele Lanza. Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship. Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06), 2006.

[8] Larry Seltzer for Zero Day. Excess privilege makes companies and data insecure. ZDNet. October 22, 2013. Topic: Security. Link: https://www.zdnet.com/article/excess-privilege-makes-companies-and-data-insecure/
[9] Hulley SB, Cummings SR, Browner WS, Grady DG, Newman TB. Designing Clinical Research. 3rd ed, Philadelphia, PA: Lippincott Williams and Wilkins; 2007.
[10] Robson C. Case Studies for Software Engineers. (2002) Real World Research. Blackwell, (2nd edition).
[11] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A.. 2012. Experimentation in Software Engineering. Springer Publishing Company, Incorporated.
[12] Firouzi, E., Sami, A. Visual Studio Automated Refactoring Tool Should Improve Development Time, but ReSharper Led to More Solution-Build Failures. MAINT 2019, Hangzhou, China.
[13] Derezińska A. (2019) Metrics in Software Development and Evolution with Design Patterns. In: Silhavy R. (eds) Software Engineering and Algorithms in Intelligent Systems. CSOC2018 2018. Advances in Intelligent Systems and Computing, vol 763. Springer, Cham.
[14] Badri, M., Badri, L., Hachemane, O., Ouellet, A. Measuring the effect of clone refactoring on the size of unit test cases in object-oriented software: an empirical study. Innovations in Systems and Software Engineering. Springer-Verlag London Ltd., part of Springer Nature 2019.
[15] Afjehei, S.S., Chen, T-H. P., Tsantalis, N. iPerfDetector:Characterizinganddetecting performanceanti-patternsiniOSapplications. EmpiricalSoftwareEngineering. SpringerScience+BusinessMedia,LLC,partofSpringerNature 2019.
[16] Kaya, M., Conley, S., Othman, Z.S., Varol, A. Effective Software Refactoring Process. 2018 6th International Symposium on Digital Forensic and Security (ISDFS). Antalya, Turkey.
[17] OPDYKE, William F. Refactoring object-oriented frameworks. 1992.
[18] ROBERTS, Donald Bradley; JOHNSON, Ralph. Practical analysis for refactoring. University of Illinois at Urbana-Champaign, 1999.
[19] MINUZZI, Tiago da Silva. Ustory-Refactory: ferramenta de refatoração de requisitos aplicada em cartões user stories (CRC Cards). 2007.

TAYSE VIRGULINO RIBEIRO Possui graduação em Sistemas de Informação pelo Centro Universitário Luterano de Palmas (2018). Mestranda em Ciência da Computação no CIn-UFPE, na Área de Engenharia de Software. Atualmente é Gestora do setor de TIC e Professora na Unibra - Centro Universitário Brasileiro. Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: processos de desenvolvimento de software, engenharia de software, informática na educação e interação homem computador. Com pouco mais de 6 anos de experiência no mercado, centraliza suas atividades na área de Engenharia de Software e afins (Líder de projetos, Scrum Master e Analista de Sistemas).

MÁRCIO LOPES CORNÉLIO Possui graduação em Ciência da Computação (Bacharelado) pela Universidade Federal da Paraíba (1996), mestrado em Ciências da Computação pela Universidade Federal de Pernambuco (1998) e doutorado em Ciência da Computação pela Universidade Federal de Pernambuco (2004). Atualmente é professor adjunto da Universidade Federal de Pernambuco. Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: métodos formais, refatoração e transformação de programas.