Effects of Contextual Information on Maintenance Effort: A Controlled Experiment

Leandro Ferreira D'Avila, Kleinner Farias, Jorge Luis Victória Barbosa

Applied Computing Graduate Program (PPGCA)
University of Vale do Rio dos Sinos (UNISINOS)
São Leopoldo - 93.022-000 - Brazil

Abstract

There has been an increased focus on context-aware tools in software engineering. Within this area, an important challenge is to define and model the context for software-development projects and software development in general. This article reports a controlled experiment that compares the effort to implement changes, the correctness and the maintainability of an existing application between two projects; one that uses qualitative dashboards depicting contextual information, and one that does not. The results of this controlled experiment suggest that the usage of qualitative dashboards improves the correctness during the software maintenance activities and reduces the effort to implement these activities.

Keywords: Maintenance Effort, Empirical Study, Contextual Information, Qualitative Dashboard

1. Introduction

Developers need to deal often with maintenance activities on existing applications in order to adapt them to new scenarios and needs, for example, new features, bug fixing and conformance to legal requirements. Software maintenance is defined as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment (Rajlich, 2001). According to Lientz and Swanson (1980), maintenance activities are categorized as adaptive, perfective, corrective and preventive. To perform maintenance activities, developers must consider a lot of information regarding the code under maintenance to guarantee the software quality, maintainability and continuous integration.

Besides that, developers often deal with organization factors which may have a potential impact on the success or failure of software development projects. Some of these organization factors, observed by Lavallée and Robillard (2015), are: large amount of poorly documented

Email addresses: `leandro.davila@gmail.com` (Leandro Ferreira D'Avila), `kleinnerfarias@unisinos.br` (Kleinner Farias), `jbarbosa@unisinos.br` (Jorge Luis Victória Barbosa)

software, lots of interdependencies between software modules and conflicts between projects on the scheduling of deployment. According to the authors, these organizational factors cause the following impacts on the software-development projects:

- Some requirements are discovered late in the development process;
- Compatibility between modules is patched quickly and haphazardly;
- Frontiers between processes hinder information exchanges and developers must work with missing details.

According to Baysal et al. (2013), a way to mitigate the impact of these factors on software correctness and maintainability can be providing useful information regarding the context of code or application under development using the analytics approach. The availability of this information provides a better understanding of the developer in relation to issues surrounding the software and its environment. Zhang et al. (2013) showed that analytics techniques can be used to better understand software quality. According to the authors, the main objective of the analytic techniques is to provide actionable, real-time insights with quantitative and qualitative information.

The qualitative data provided through dashboards can improve the developers' situational awareness during software development process, more specifically on code maintenance, providing actionable information. This allows developers to anticipate possible gaps, be aware about diverse aspects regarding the source code, and finally implement a suitable solution in terms of correctness and maintainability.

The definition and use of context in software development have received special attention in recent years (Antunes et al., 2011; Leano et al., 2014; Latoza et al., 2014; Haron and Syed-Mohamad, 2015; Briand et al., 2017; Cazzola and Shaqiri, 2017; Murphy, 2018). In this scenario, this study aims to discuss what information should comprise the context of a software product and project, its history, and how this data can help software developers to implement change requests.

This paper reports empirical findings on the usage of contextual information through a dashboard, while software developers perform maintenance tasks. We have conducted a controlled experiment with 30 subjects to evaluate and compare software maintenance activities using contextual information with respect to correctness, effort and maintainability. The subjects were organized in two groups and performed two maintenance activities where only one group used the contextual information represented by a dashboard. The main results, supported by statistical analysis, suggest that: (1) the usage of dashboard produced a code with a higher correctness score; (2) significantly less effort spent on implementing changes in the group using the dashboard; and (3) the changes performed by the group using dashboard produced a better code, in terms of architecture. Even though we cannot generalize the results of the experiment to other software maintenance activities, this exploratory study represents a contribution to better understand the potential effects of contextual information on software maintenance.

The remainder of the paper is organized as follows. Section 2 outlines the main concepts used throughout the paper. Section 3 describes the study methodology. Section 4 discusses the study results. Section 5 describes how threats to validity were minimized. Section 6 compares this work with others, presenting the main differences and commonalities. Finally, Section 7 presents concluding remarks and future work.

2. Background

This section presents the main concepts related to relevant software data, their relation to context definition and its visualization. Section 2.1 describes the concepts of *context* and *context history*. Section 2.2 details the interplay between context and software information. Section 2.3 introduces the concepts regarding software analytics, because this topic was considered strategic for the next steps in this research.

2.1. Context and Context History

The term *context* has an intuitive meaning for humans, but its definition remains vague and an ill defined construct. Furthermore, the roles of context come from different fields, such as literature, philosophy, linguistics and computer science, with each field proposing its own view of context (Mostefaoui et al., 2004). It generally refers to what surrounds the center of interest. Additionally, contexts provide additional sources of information related to "who, where, when and what" (Morse et al., 2000) and increase understanding.

Dey et al. (2001a) provided a classical categorization for context data, which is used by many works. The authors argued that context-aware applications must understand the situation of users and their surroundings, and adapt the application behavior according to this information. Thus, they proposed four basic categories to model context, which are: (1) identity; (2) location; (3) time; and (4) activity. These context types do not only answer the questions of who, where, when, and what, but also act as pointers to other sources of contextual information. According to Satyanarayanan (2001), a user's context can be quite rich, consisting of attributes such as physical location, physiological state (such as body temperature and heart rate), emotional state (such as angry, distraught, or calm), personal history, daily behavioral patterns, and so on. In general, this means information such as the status, identity and spatiotemporal localization of persons, groups and physical and computational objects.

Dybå et al. (2012) suggested an omnibus context description approach, putting a phenomenon into context. It means, they proposed to ask what (e.g. a particular architectural style) works for whom (e.g. professional system architects) where (e.g. the location) when (e.g. the time related to the life-cycle of a product) and why (e.g. the reasoning of why the architectural style works well).

The context awareness and adaptation were always considered relevant technologies to the development of mobile and ubiquitous systems (Barbosa, 2015) in the area of health (Damasceno Vianna and Barbosa, 2014), commerce (Barbosa et al., 2016), accessibility (Tavares et al., 2016; Barbosa et al., 2018), learning (Wagner et al., 2014; Wiedmann et al., 2016), competences management (Rosa et al., 2015) and well-being (Vianna et al., 2017). All these works used the definition of Dey et al. (2001a) as the basis for their context models. The definition is generic enough to be applied across different domains. In this sense, all these works referred to the same "context", that is, the one defined by Dey, Abowd and Salber. In contrast, each computational model that applied this definition to create a ubiquitous system needed to specify some aspects related to the specific domain. For example, the health domain has considered vital signs and health resources available in the environment, and the accessibility has used users' deficiencies and accessible resources. The definition of Dey et al. (2001a) allows this flexibility through a specific information category called "Status" or "Activity", what according to them "identifies intrinsic characteristics of the entity that can be sensed." All contexts used in specific ubiquitous applications have differences, but the definition used as the basis is the same. In addition, the literature has discussed the role of the context concepts in general software development (Latoza et al., 2014; Leano et al., 2014; Martie and Hoek, 2014). In this sense, a research challenge is to model generic tools to support the use of contexts in software development.

Some authors use the concept of context histories (Rosa et al., 2015, 2016). In general, these approaches have in common the fact of dealing with records of sequences of events, ordered chronologically and tied to an identifiable entity. The difference is the type of information that is described in these sequences. Some of these studies treat sequences describing the location (Driver and Clarke, 2004; Li et al., 2012) or user's activities (Driver and Clarke, 2004, 2008; Smith, 2008), while Silva et al. (2010) offer support to generic entities. This generic approach corresponds to the categories proposed by Dey et al. (2001b) for describing the context of an entity.

Temporal series are sets of observations sequentially ordered (Fu, 2011). Based on this definition, Wiedmann et al. (2016) adopted the premise that the sequences of contexts visited by an entity can be described as a temporal series, since they are chronologically ordered and relevant data for the similarity analysis are quantifiable variables, i.e., temperature, velocity, heart rate, speed, costs, sales volume, and others. The temporal series are also used to commerce support (Barbosa et al., 2016) and to assist wheelchair users (Barbosa et al., 2018).

2.2. Context and Software

Briand et al. (2017) highlighted the importance of context in software engineering. According to the authors, software engineering solutions' applicability and scalability depend largely on contextual factors, such as human (engineers' background), organizational (such as cost and time constraints), or domain-related (such as the level of criticality and compliance with standards) .

Petersen and Wohlin (2009) proposed a checklist for the context description in software maintenance. This checklist identifies some context facets as a product, processes, practices and techniques, people, organization, and marketing. Besides that, the authors define the object of study that interacts with the context. For example, when an agile process is considered as the object of study, the process is used to develop a product, it is run by people, interact with other processes, and it is supported by practices, tools and techniques. In addition, the object of study is embedded in an organization that is operating within a market.

Each context facet comprises a set of context elements, such as:

- **Product:** The product is the software developed with the help of the object of study. The context elements considered for product have information like maturity, quality, size of the product, system type and programming language;
- **Process:** The process describes the development workflow considering the activities and their artifacts;
- **Practices, Tools, Techniques:** Practices, tools, and techniques describe systematic approaches that are used in an organization, and are interacting with the object of study;
- **People:** The human factor is important when studying software development, as it has a major impact on the success of this activity. The context elements considered for people have information regarding roles and experience;
- **Organization:** The organization describes the company structure in which the other context facts and the solution are embedded in;
- **Market:** The market represents the customers and competitors. Context elements describing the market are number of customers, market segments, strategy and constraints.

Antunes and Gomes (2009) propose an approach that follows some categories of context described by Petersen and Wohlin (2009). However, the object of study considered in this case is the developer. The contextual information should be captured considering every work environment that supports the developer's work. It means that contextual information must be retrieved from all applications that the developer uses, as an IDE, project management tools, human capital management tools and even the operating system itself. Many of these tools already provide data extraction interfaces via plug-in or APIs. The information obtained through the applications should then be centralized and integrated coherently in a context model, so the user context model is instantly available anywhere in real time.

Murphy (2018) argues that the lack of context in the software engineering tools limits the effectiveness of developers and compromises the software development practices. The development of a software system requires the orchestration of many different people using many different tools. Considering the need of developers to understand the contexts in which the tasks are undertaken, the tools would be helping them to work within the appropriate context.

Antunes et al. (2011) consider, in turn, that software context takes into account all dimensions that characterize the work environment of the developer. These dimensions were represented as a layered model with four main layers: personal, project, organization and domain. The *personal layer* represents the context of the work a developer has at hands at any point in time, which can be defined as a set of tasks. In order to accomplish these tasks, the developer has to deal with various kinds of resources at the same time, such as source code files, specification documents and bug reports. The *project layer* focuses on the context of the project, or projects, in which the developer is somehow involved. A software development project is an aggregation of a team, a set of resources and a combination of explicit and implicit knowledge that keeps the project running. The team is responsible for accomplishing tasks, which end up consuming and producing resources. The *organization layer* takes into account the organization context to which the developer belongs. Similarly to a project, an organization is made up of people, resources and their relations, but in a much more complex network. The *domain layer* takes into account the knowledge domain, or domains, in which the developer works. This layer goes beyond the project and organization levels and includes a set of knowledge sources that stand out of these spheres. Nowadays, a typical developer uses the Internet to search information and to keep informed of the advances in the technologies.

Baysal et al. (2013) and Augustine et al. (2017) suggested that supplementing quantitative dashboards with more developer-specific qualitative data can improve developers' situational awareness of their working context. This awareness will enable developers to keep better track of the ever-increasing number of issues involved in complex software systems. Situational awareness is a term from cognitive psychology referring to a state of mind in which people are aware of the elements of their immediate environment, have an understanding as to the environment's greater meaning, and can anticipate (or plan to change) these elements in the near future (Endsley, 1995). The term is used in engineering, for example, to describe how air traffic controllers work, more specifically, as they track and route air traffic. It is also an apt description of how software developers must maintain awareness of what is happening on their projects and as they manage a constant flow of information and react accordingly.

2.3. Software Analytics

Software engineering is a data rich activity. Many aspects of development from code repositories can be measured with a high degree of automation, efficiency, and granularity. Projects can be measured throughout their life-cycle from specification to maintenance. Over the past few decades, researchers have used analytics techniques on such data to better understand software quality (Zhang et al., 2013; Menzies, 2018). The analytic approaches strive to provide actionable real-time insights and can be both quantitative and qualitative in nature. The quantitative analytics can highlight high-level data trends, while qualitative analytics enable real-time decision making for day-to-day tasks (Baysal et al., 2013).

Menzies and Zimmermann (2013) defined software analytics as the use of analysis and systematic reasoning on software data for managers and engineers to gain and share insights from their data to make better decisions. Software analytics enables software practitioners to perform data exploration and analysis in order to obtain insightful and actionable information for data-driven tasks around software and services (Zhang et al., 2011). In another study, Zhang et al. (2013) define that software analytics aims to obtain insightful and actionable information from software artifacts that help developers accomplish tasks related to software development.

Software analytics utilizes data-driven approaches to enable software practitioners to perform data exploration and analysis to obtain insightful and actionable information for completing various tasks around software systems, software users, and software development process (Lou et al., 2013). Buse and Zimmermann (2012) provided the following definition about software analytics: “Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions”.

There are also many potential advantages to the application of analytics to software development and project management. Analytics can help: monitor a project; know what is really working; improve efficiency; manage risk; anticipate changes; evaluate past decisions (Buse and Zimmermann, 2012).

Port and Taber (2017) described the use of analytics to monitor more widely the development process of system (Monte). Using these data it is possible, for example, to evaluate/mitigate the risks related to the maintenance activities and forecast the effort need to add new features to Monte. This system has with over 800,000 lines of code and supports NASA’s Jet Propulsion Laboratory (JPL) in critical missions. The authors presented some examples of how the usage of a robust software metrics and analytics enables actionable maintenance management of Monte system in a timely, economical, and risk-controlled fashion.

3. Experiment Planning

This section describes the main decisions considering the design of the controlled experiment to grasp the impact of the use of contextual data on the software maintenance. All methodological steps described in this section are based on well-known guidelines about empirical studies (Wohlin et al., 2012; Kitchenham et al., 2008; Sjøberg et al., 2002).

3.1. Contextual Information

Section 2.2 discussed the use of contextual information to support the software development, mainly describing specific characteristics that should be considered as Contexts to software. The work of Petersen and Wohlin (2009) cited software dependencies, last changes, performance requirements, software usage and design patterns as strategic aspects that should be considered in the contextual information.

Based on the study carried out through the background literature review (Section 2) and related works discussion (Section 6), we have chosen the information indicated by Petersen and Wohlin (2009) as the basis of the contextual information applied in the experiment. The information was improved and organized in a dashboard as suggested by Baysal et al. (2013) and Augustine et al. (2017), and provided to the subjects in a PDF file. The contextual information made available to users was as follows:

- **Design patterns**: This group of information aims to advice the developer about the current architecture of application. The developer should correlate the change requests with a possible software artifact (e.g class or method) in terms of architecture (e.g. factory) and extend this pattern avoiding degradation (Latoza et al., 2014).
- **History of last changes**: By checking the history of last changes, the developers would identify similar previous changes and more easily find the source code artifact to be changed or discover some code examples (Sawadsky and Murphy, 2011).

- **Customer usage of features** and **Performance execution**: These groups were created as simulation regarding the feature usage and application response time. The aim of that is to get the developers aware about the possible impacts of a wrong implementation, for example, for the customers that are using the application (Zhang et al., 2013). At this point, the developer can correlate the variation of time responses with the history of changes and more easily identify the artifacts involved.
- **Non-functional requirements**: This information usually represents quality patterns or restrictions defined for the application under maintenance. Depending on development method and technology, these factors impact the software architecture and must be considering during the maintenance activities (Carlson et al., 2016).

3.2. Objective and research questions

This research aims to evaluate the effects of contextual information on three variables: correctness of source code, effort to implement change requests, and maintainability. These variables are explored in Section 3.8. These effects were investigated based on realistic maintenance scenarios involving the realization of change requests, thereby generating empirical knowledge about the benefits of contextual information to improve the developer's situational awareness. The participants (Section 3.4) used the contextual information represented using a dashboard as a prove of concept to improve their situational awareness about the application to be changed.

The objective of this study is stated based on the GQM template (van Solingen et al., 2002) as follows:

Analyze **the use of contextual information**
for the purpose of **investigating its effects**
with respect to **correctness, effort and maintainability**
from the perspective of **students**
in the contexts of **performing assigned maintenance activities**.

The effort is measured considering the time expended by the subjects to perform the requested changes. Some manual tests were defined to validate the correctness of each maintenance task. The maintainability variable is ensured considering the place where the subjects have implemented the changes requests, keeping the application's architecture. These three variables are described in detail in Section 3.8. Thus, we focus on three research questions, as follows:

RQ1: What is the correctness of source code after undergoing two software maintenance activities using contextual information and not using contextual information?

RQ2: What is the effort of implementing two software maintenance activities using contextual information and not using contextual information?

RQ3: What is the maintainability of source code using contextual information and not using contextual information?

3.3. Hypothesis formulation

To address the research questions described before, three hypotheses were formulated. The first refers to evaluate the correctness of changed code by using contextual information through a dashboard. The second hypothesis considers the usage of the same dashboard and its effects on the effort on implementing the change requests. The third hypothesis considers the usage of the same dashboard and its effects on the maintainability of the changed code. The formulation of these hypotheses was done based on the assumption that, the contextual information improves the developer's awareness regarding the application under changes, and the environment related. The formulation of three hypotheses is described as follows.

Hypothesis 1. In practice, developers usually go to source code having no qualitative information whatsoever to support the changes to be done, relying only on an arsenal of mentally-held indicators about the source code (a.k.a "experience"). Even though this strategy to implement the change requests is often used in practice, we conjecture that it is not effective enough to correctly support the implementation of complex change requests. In part, because usually developers need to maintain non-functional requirements, e.g., performance and security, whose implementations are widely known as crosscutting concerns. That is, their implementations end up scattering and spreading over many modules of the system, giving rise to code duplication and significant dependencies between modules. Moreover, it is very difficult to gather qualitative information about how these requirements are implemented, apart from that found in source code, to properly realize the change requests. Consequently, we hypothesize that developers tend to produce a higher number of correctly changed source code by making use of contextual information rather than using their personal experiences and knowledge. Therefore, the first hypothesis evaluates whether the use of contextual information with qualitative information produces a higher number of correctly changed source code compared to the use of mentally-held indicators only. Based on this statement, we state the null and alternative hypotheses as follows:

- **Null Hypothesis 1,** H_{1-0}: The use of contextual information produces a lower (or equal) number of correctly changed source code than the use of mentally-held indicators only (no-Context).

  H_{1-0}: $Corr(Code)_{Context} \leq Corr(Code)_{No-Context}$
- **Alternative Hypothesis 1,** H_{1-1}: The use of contextual information produces a higher number of correctly changed source code (Context) than the use of mentally-held indicators only.

  H_{1-1}: $Corr(Code)_{Context} > Corr(Code)_{No-Context}$

Hypothesis 2. As previously mentioned, fast-changing and unpredictable customer requirements in most software development projects have given rise to an elevated number of change requests. Developers are often challenged to accommodate such requests into evolving source code. In practice, developers aim at implementing the required changes without understanding the side-effects of the changes and the software architecture itself. Moreover, developers often work under pressure to deliver the source code modified more rapidly. Due to time constraints and lack of qualitative information about the source code (e.g., design patterns used, design decisions and non-requirements to be considered), developers end up implementing the change requests considering only information found in the source code (Lavallée and Robillard, 2015). With this in mind, we conjecture that the use of contextual information may reduce the implementation effort significantly. That is, we suspect that the effort to implement the change requests tends to be lower if developers use qualitative information about the source code, apart from the developers' experience. Therefore, the second hypothesis evaluates whether the use of contextual information with qualitative information reduces significantly the effort of the change requests implementation, compared to the use of developers' mentally-held indicators and needed source code analysis. Based on this statement, we state the null and alternative hypotheses as follows:

- **Null Hypothesis 2,** H_{2-0}: The use of contextual information (Context) does not reduce significantly the effort of implementing the change requests compared to the use of developers experience only (no-Context).

  H_{2-0}: $Effort(Code)_{Context} > Effort(Code)_{No-Context}$
- **Alternative Hypothesis 2,** H_{2-1}: The use of contextual information (Context) reduces significantly the effort of implementing the change requests compared to the use of developers experience only (no-Context).

  H_{2-1}: $Effort(Code)_{Context} \leq Effort(Code)_{No-Context}$

Hypothesis 3. The maintenance activities usually came from urgent issues, legal requirements, environment changes and others. Developers need to adapt the system to these changes and try to keep the software architecture according to the best practices and previous design considering some organization factors, like lack of documentation, with a potential impact on the success or failure maintenance activities. According to nature of the requests, developers need to go changing the code with the focus on solve the issue, as soon as possible, without considering properly the architecture of the software and the side effect of the changes to the customers, users and module/system integration. Besides that, the customer pressure may often reduce the time that developers have to analyze and implement the needed system changes. To deal with this complex network of factors, developers perform the changes based on the their own experience and knowledge about a particular module (Lavallée and Robillard, 2015). It is very difficult to get qualitative information about the architecture adopted, design patterns and software dependencies apart from that found in source code, to properly perform the maintenance activities in the right place, it means, in the correct class, include, method, function or procedure. So, we conjecture that the maintainability of a modified software code can be different if there is available contextual information with qualitative information regarding this artifact to the process of changing it. We suspect that the code may have a higher maintainability if the change is carried out using contextual information with qualitative information about the source code and not only considering the developer's experience. Therefore, the third hypothesis evaluates whether the use of contextual information with qualitative information improves the maintenance of source code compared with the use of developer's experience only. Based on this statement, we state the null and alternative hypotheses as follows:

- **Null Hypothesis 3,** H_{3-0}: The use of contextual information (Context) does not improves significantly (or even jeopardize) the maintainability of evolving source code compared to the use of developers experience only (no-Context).

  H_{3-0}: $Maint(Code)_{Context} \leq Maint(Code)_{No-Context}$

- **Alternative Hypothesis 3,** H_{3-1}: The use of contextual information (Context) improves significantly the maintainability of evolving source code compared to the use of developers experience only (no-Context).

  H_{3-1}: $Maint(Code)_{Context} > Maint(Code)_{No-Context}$

These three hypotheses consider the variables correctness, effort and maintainability which will have measurement method detailed in Section 3.8.

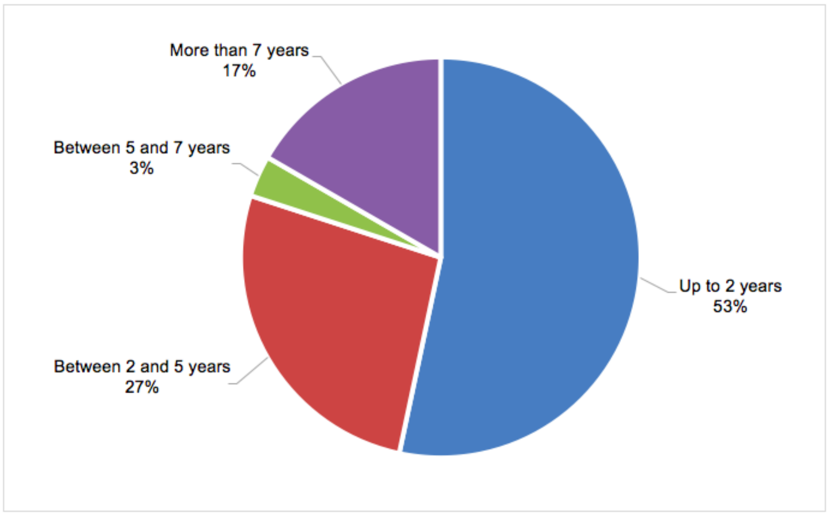
3.4. Context and Subject Selection

The subjects used the Eclipse IDE (Integrated Development Environment) (Eclipse, 2016) to perform the software maintenance activities extending the scope of an existing application. The maintenance activities were tasks on a prototype Java application developed by Deitel and Deitel (2010), where different types of salary are calculated. This application is well structured and implements the basic concepts of object oriented paradigm. Besides that, this application is widely used in learning of the Java programming language and this factor would facilitate the execution of experiment activities. The application is not large due to its educational purpose as detailed by the following metrics extracted using *Google CodePro AnalytiX* tool[1]: 491 lines of code (LOC), 5 packages (NOP), 10 classes (NOC) and 66 methods (NOM).

In total, 30 subjects were recruited by convenience (Wohlin et al., 2012). The experiment was conducted with 4 professionals from Brazilian companies and 26 students with professional experience. They were recruited considering the level of theoretical knowledge and practical experience. Majority of the subjects had experience with software development, which were acquired from previous software development projects. Figure 1 shows the subjects distribution regarding their experience in software development.

[1]https://marketplace.eclipse.org/content/codepro-analytix

Figure 1: Subjects distribution by experience



All participants were familiar with the Eclipse IDE and Java programming language. Based on that, we are confident that they had a proper training, theoretical knowledge and practical experience to get rid of any threat to the vitality of our results. All subjects held a Master's degree, Bachelor's degree or equivalent, and had a considerable knowledge to participate in the experiment.

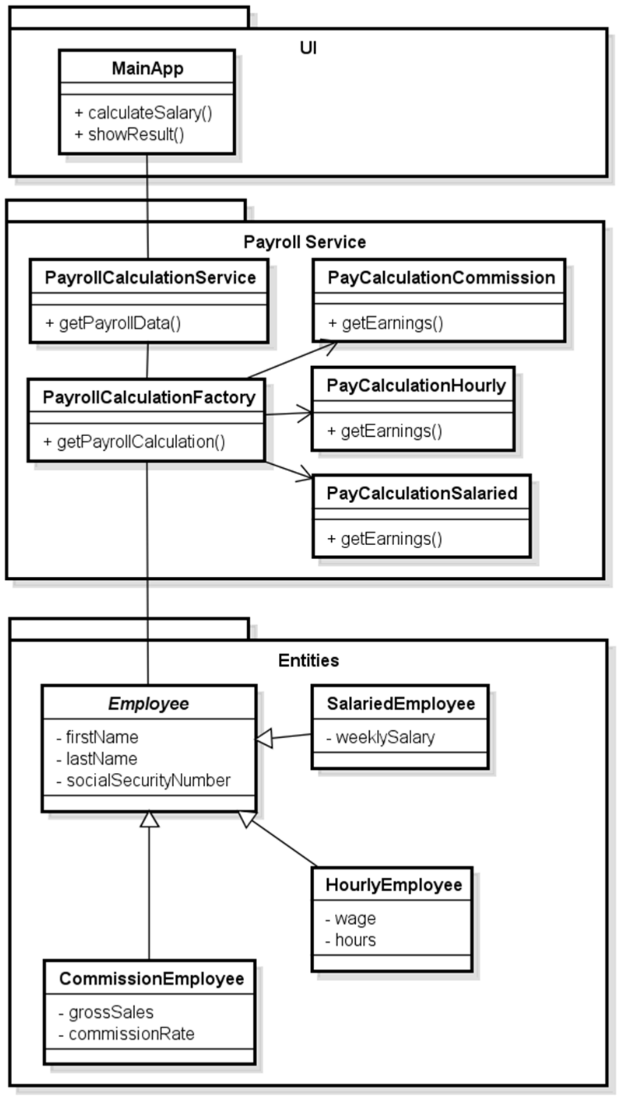
3.5. Experiment Design and Object

Completely randomized design. Our study is a human-oriented experiment, in which humans are the subjects, applying two treatments (with or without dashboard) to objects, which are programs to be maintained. Each subject was submitted to one treatment (either maintenance of a program with or without dashboard). In this sense, the study adopted an experiment design that allowed us to compare two treatment means. The design setup allowed the subjects to make use of the same object for both treatments. This means that half of subjects performed maintenance tasks on one program without dashboard support, while the rest had dashboard support to carry out maintenance tasks. The assignment of the subjects to each treatment was done randomly. However, the experience of subjects was considered to have balanced groups in terms of software-development knowledge. The number of subjects per treatment was equal. Each subject was assigned to the treatments randomly. Therefore, our experiment design can be considered as *balanced*.

Object. The object of the experiment consists of the program that the subjects had to change when executing the experimental tasks. This program is a Java application, whose its main feature is to calculate salary in three different ways. It is based on applications found in Deitel and Deitel (2010) and follows design patterns and good programming practices. Subjects interact with the application through a graphical user interface, which shows the calculated salary. Figure 2 shows a class diagram of the application. These classes are responsible for implementing the feature related to the salary calculation:

- *Entities* package contains classes that refer to the types of existing employment contract, i.e., Commission, Hourly and Salaried employee.
- *Payroll Service* package has classes that are responsible for implementing services to compute salary according to the different types of existing employment contract.
  1. *PayCalculationCommission*: It computes the commission-based salary. In this sense, the method *getEarnings()* multiples the *grossSales* and *commissionRate* attributes of *CommissionEmployee* class.
  2. *PayCalculationHourly*: It calculates hourly-based salary. The method *getEarnings()* multiples the attributes *HourlyEmployee.wage* and *HourlyEmployee.hours*.
  3. *PayCalculationSalaried*: It calculates incomes of paid employment. The method *getEarnings()* multiples the attribute *SalariedEmployee.weeklySalary* by the number four.
- *UI* package contains the MainApp class, which is responsible for displaying the computed salary to the end user.

Figure 2: The main class diagram of the application



3.6. Execution

This section presents each step followed to run the experimental design. We highlight that any deviations from the plan occurred. Jedlitschka et al. (2008) and Jedlitschka and Pfahl (2005) reinforce the need to discuss how the experimental design was enacted.

Preparation. Participants were allocated to two experimental groups. During group formation, we were concerned to keep groups balanced in terms of participants' level of knowledge. This action was taken to prevent more experienced participants from staying in a group. This could interfere with the results, since the participants did not act on the two treatments. This preparation stage lasted 2 hours and was performed in conjunction with the training, which will be explained in Section 3.7.

Data collection performed. The data collection related to variables (described in

Section 3.8) was followed so that deviations were avoided. Effort was calculated in minutes. Each participant recorded the time invested to perform each activity on a questionnaire. This effort record was followed closely to avoid any incorrect record. Correctness and maintainability were computed considering the source code changed by participants.

Validity procedure. Throughout the experiment the authors had a great concern to ensure the correct execution of the experimental project. In this sense, the defined experimental process was explained to all participants. This explanation was important for all participants to be aware of all experimental tasks. This avoided deviations during the execution of the studies.

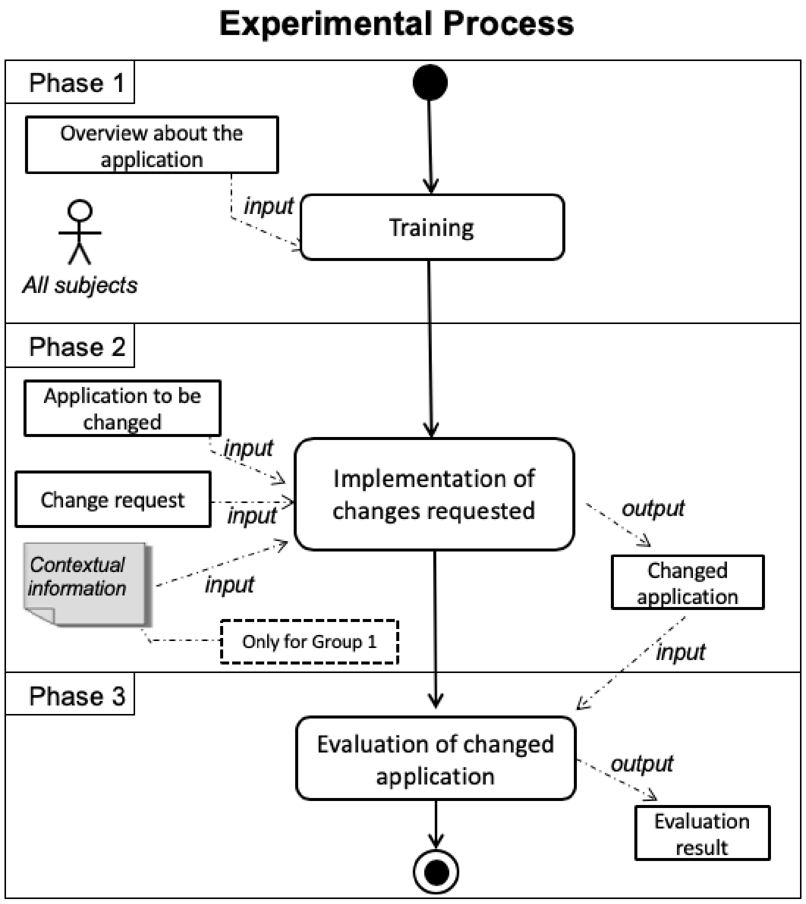
3.7. Experimental Process

Experimental process. Figure 3 presents the process adopted to run the experimental study. This experimental process is composed by three tasks, which are organized in 3 phases as follows:

- **Phase 1: Training.** All participants received training on the program used in the experiment. At this moment of familiarization, the participants were able to perform, analyze and be environmentally friendly with the source code. Subjects received details on each feature concerning the program to be maintained (experiment object). The subjects were not aware of RQs to avoid bias during the experiment execution. The knowledge received by subjects was passed through a session where the main class diagram of application was explained in detail, the requirements already implemented were described and finally, a demonstration of current application behavior was performed.
- **Phase 2: Execution of experimental tasks.** The second phase concentrated on the execution of the maintenance tasks. For this, subjects had access to the source code of application in a laboratory environment, and change requests that needed to be applied in such an application. Only half of the subjects accessed contextual information through a dashboard support. To carry out the change requests, 30 participants were randomly divided into two groups of 15. The first group, named *Group 1*, performed the implementation of changes with the use of contextual information represented in a dashboard, which contains contextual information about the application. The *Group 2* made the same implementation without using the contextual information. The subjects individually performed the first two phases to avoid any threat to the experimental process.
- **Phase 3: Evaluation of results.** The subjects recorded the start and end times for each implementation task. This information was filled through a Google Form application and allowed to compare the implementation effort of both subject groups. Moreover, the correctness of each implementation task was evaluated by manual and automated testing. All 30 produced codes were tested in order to check their results against the expected values. In addition, some maintainability aspects were evaluated by code review. All produced codes were revised to evaluate the implementation

way adopted by the subjects. This review allowed to identify how the application architecture remained consistent after the changes.

Figure 3: The experimental process adopted.



Experimental tasks. The participants were requested to change the application by including two new features in a classroom prepared for the experiment. The implementation of both tasks was monitored by the authors to avoid any communication between the subjects. The first task was to compute a tax and net value based on the existing salary calculation. The second maintenance activity was to create a new type of salary composed by a base salary plus a commission. The second activity was considered harder to implement than the first task due to the amount of code needed to cover the requirement. Both tasks were defined to have no dependence on the order of execution. It means that the subjects can implement the second task even though they have implemented the first one wrongly. These two requirements define the change request to be implemented. To keep the architecture of application the participants should implement the features as follows:

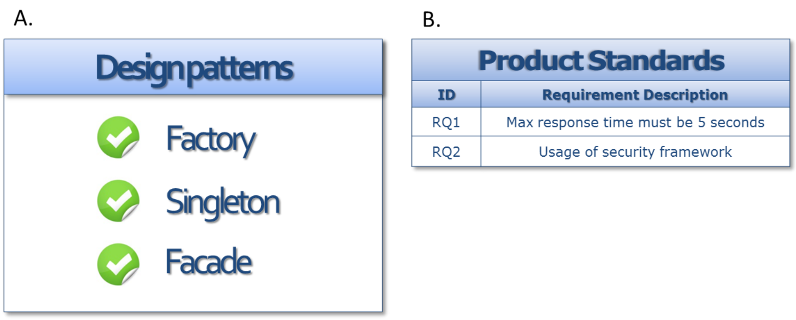
1. Tax calculation: The participants should use the *PayrollCalculationService* class and use the predefined variables;

2. New type of salary: The participants should extend the *CommissionEmployee* or *Employee* class and create a new pay calculation class, which must be called by the factory *PayrollCalculationFactory* class.

The dashboard of contextual information provided (only to Group 1) was created manually with quantitative and qualitative information regarding the application used in the experiment. The information available in this dashboard represent: the existing design patterns, a history of the last code changes, a graph representing the customers usage of features, a graph representing the performance execution and non-functional requirements.

Figure 4.A shows the design patterns implemented in the application. Thus, subjects can be aware about these aspects without interpreting the source code and keeping them in mind when implementing the change request. Figure 4.B represents the non-functional requirements implemented. This information aims to advise the developer that there are important aspects already considered in the application as security and performance needs.

Figure 4: Design patterns and Non-functional requirements



A.

Design patterns

✓ Factory

✓ Singleton

✓ Facade

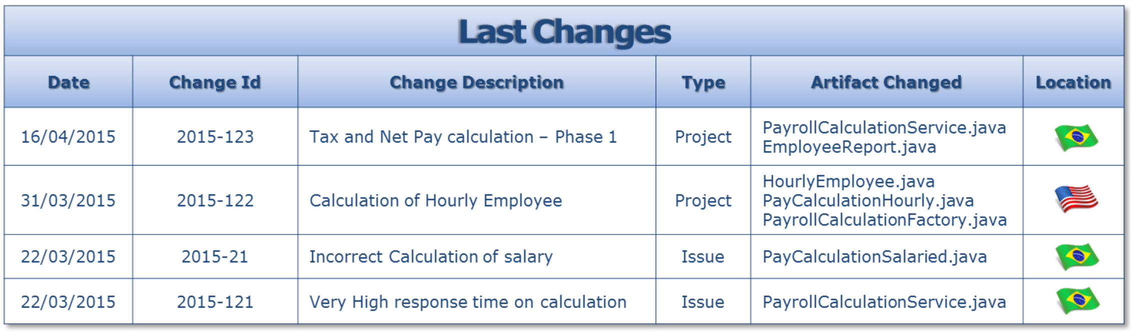
B.

Product Standards

ID	Requirement Description
RQ1	Max response time must be 5 seconds
RQ2	Usage of security framework

As mentioned by Augustine et al. (2017), developers often do not know whom to contact for guidance when modifying an unfamiliar part of the project. A history of last changes of the application is provided to the developer in the dashboard through a table. Figure 5 represents this information and the developer can be more effective implementing the changes requested by checking for a previous version.

Figure 5: Description of the last changes performed in the source code.



Last Changes

Date	Change Id	Change Description	Type	Artifact Changed	Location
16/04/2015	2015-123	Tax and Net Pay calculation – Phase 1	Project	PayrollCalculationService.java EmployeeReport.java	🇧🇷
31/03/2015	2015-122	Calculation of Hourly Employee	Project	HourlyEmployee.java PayCalculationHourly.java PayrollCalculationFactory.java	🇺🇸
22/03/2015	2015-21	Incorrect Calculation of salary	Issue	PayCalculationSalaried.java	🇧🇷
22/03/2015	2015-121	Very High response time on calculation	Issue	PayrollCalculationService.java	🇧🇷

Complementing the context of the application, the information regarding software operation is provided to participants. Figure 6 shows a graph containing the number of customer by feature usage. This information aims to alert the developer about the importance of functionalities and the impact of a possible error. Likewise, Figure 7 presents the application response times. This information supplies the developer about the application situation related to execution time and performance measurements.

Figure 6: Customer feature usage.

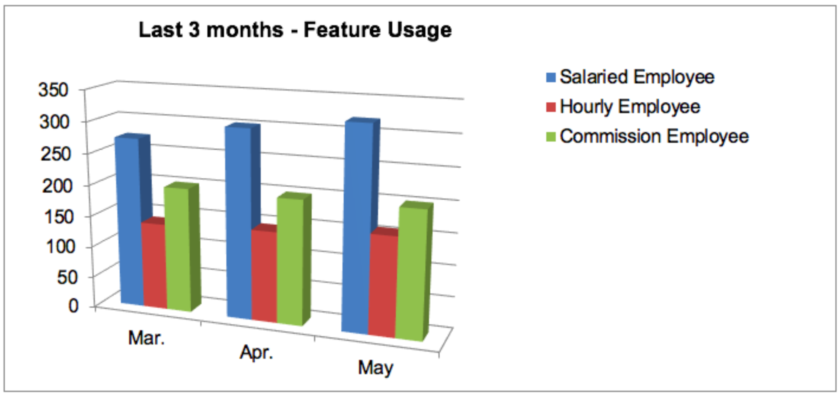
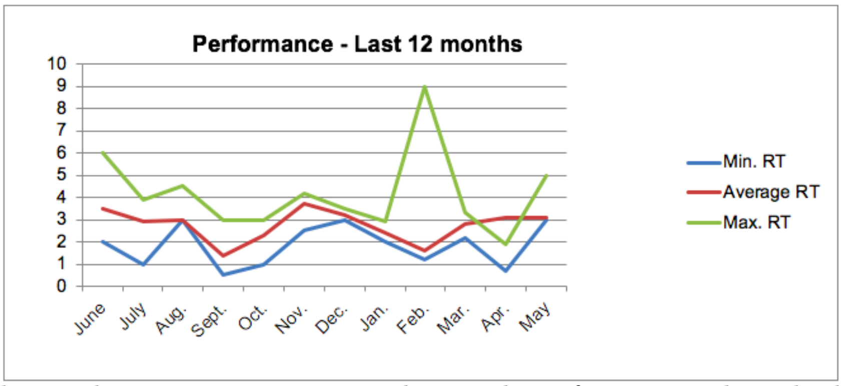


Figure 7: Performance metrics



By providing the application context, considering the information described above, the expectation is to leverage the developers' awareness about the application under development. This information aims to guide the developer to implement the change request considering the possible impacts of a wrong implementation, e.g., the number of customers that can be impacted by a bug.

3.8. Study Variables

The independent variable of this study is the usage of contextual information having qualitative information during the software maintenance activities. We investigated the impact of this independent variable in the following dependent variables:

- **Correctness:** The correctness of the maintenance activity is ensured when the output code produces the expected result according to the change request described in Section 3.5. This measurement is done by performing manual tests developed to each task and also automated unit tests previously created. These manual test cases validate the output report generated by the application where all employees are listed with they type of contract type, salary amount, taxes and net amount. For the first task, the manual test checked in the net amount was calculated properly according to the percentage defined. And, the second task was validated considering the new type of contract and the salary amount. If the result of the first task (Table 1) is correct, it is assigned the value 1. It is the same for the second task. Considering that the subject implemented improperly the two activities, then final score will be 0. By comparing the correctness, we can understand which approaches are more effective for producing code closer to the output intended result. We have measured this variable for each task performed.

- **Effort:** This variable measures the time expended by the subject to implement the change requests described in Table 1. Each implementation activity task considers different requirements and the effort to implement each one tends to be different. By comparing the values (in minutes) assumed by these variables, we can also grasp how the usage of contextual information outnumbers the other one considering a particular task.
- **Maintainability**: This variable measures if the subject implemented the code respecting the architecture of the application and the design patterns involved. Each implementation activity task considers different maintainability aspects to define the final score. This means that, there are expected points of source code where the developer should put the changes. These expected points of source code are based on the availability of the contextual information provided in the dashboard, more specifically the groups *Design Patterns* and *History of last changes*. The maximum score value that can be obtained by each subject is 5 points, if all changes are implemented in the expected class according to Table 1. In order to measure the maintainability variable, a maintainability rate was defined and it is calculated by dividing the sum of points obtained in both tasks by 5. Equation 1 shows how the maintainability rate is calculated.

Table 1: Accomplishment of assigned tasks.

Task	Requirement	Requested changes to the source code	Score
1	Tax Calculation	*Implement* the calculation in the PayrollCalculationService class	1
		Use the predefined variables	1
2	New Type of Salary	*Extend* the CommissionEmployee or Employee class	1
		Use the PayrollCalculationFactory to provide the new calculation	2

$$Maintainability = \left(\sum(ScoreTask1) + \sum(ScoreTask2)\right)/5 \qquad (1)$$

3.9. Analysis

We performed descriptive statistics to analyze the normal distribution (Wohlin et al., 2012) and statistical inference to test the hypotheses. The level of significance of the hypothesis tests was $\alpha = 0.05$. The analyses were performed to test the hypotheses individually considering the score obtained regarding the correctness, effort and maintainability for all tasks. To test the H_{1-1}, we applied the nonparametric Pearson's Chi-square test for the correctness score of the maintenance tasks. To test H_{2-1} and H_{3-1}, we applied the nonparametric Mann-Whitney test for the score of the two maintenance tasks.

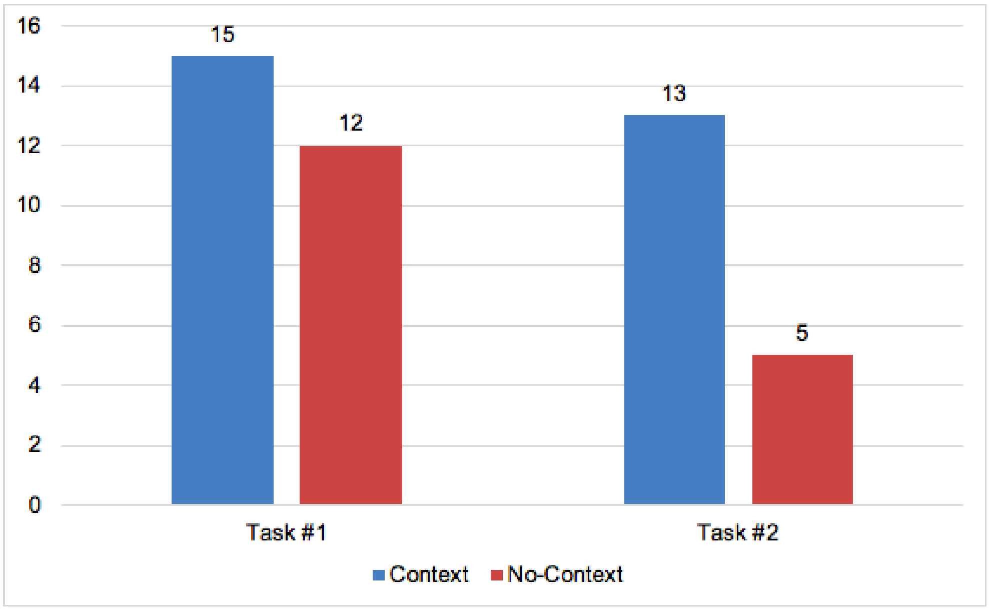
4. Study Results

This section analyzes the data set obtained from the experimental procedures described in Section 3. Our findings are derived from both the numerical processing of this data set and the graphical representation of interesting aspects of the gathered results. Section 4.1 describes the results regarding the correctness of code changed. Section 4.2 discusses the effort implementation results of change requests. Section 4.3 discusses the collected data and results related to the maintainability of code changed. Section 4.4 presents some findings and discussions.

4.1. RQ1: Correctness and Contextual Information

Figure 8 shows the number of correct modifications in of the maintenance activities implemented by the subjects. **Context** represents the sum of tasks performed correctly by the subjects of Group 1, in which the activities were performed with the usage of the dashboard. The **No-Context** represents the sum of tasks performed correctly by the Group 2 where the activities were implemented without the usage of the dashboard. The final score of Group 1 was 28 and the score of Group 2 was 17. Our initial expectation was that the sum of correct tasks of Group 1 could be higher than Group 2. This expectation was confirmed. We can consider that the usage of contextual information with qualitative information improved the correctness of the source code produce when compared to the counterpart.

Figure 8: Histogram of the correct modifications



We tested the H_{1-1} applying the Pearson's Chi-square test. Table 3 presents the contingency table, which provides the following information: the observed cell totals, (the expected cell totals) and [the chi-square statistic for each cell]. Table 2 shows the obtained results. The chi-square value obtained by the test was 10.756 and the *p-value* was 0,001. Considering that the *p-value* is lower than 0.05, then there is a statistically significant difference in the correctness measures associated with the usage of contextual information through a dashboard.

Table 2: The Pearson Chi-square test for the correctness

Task	**Statistic**	**Correctness**
All	*p-value*	0.001
	X^2	10.756

X^2 = Pearson's Chi-square, $\alpha = 0.05$

Table 3: The Pearson Chi-square test for the correctness

	Correct	**Not Correct**	**Marginal Row Totals**
Group 1	28 (21.82) [1.75]	2 (8.18) [4.67]	30
Group 2	12 (18.18) [2.1]	13 (6.82) [5.6]	30
Marginal Column Totals	40	15	60 (Grand Total)

4.2. RQ2: Effort and Contextual Information

Table 4 shows the effort (in minutes) invested by the subjects in the maintenance activities. In this table, the line **Context** represents the statistics of effort time of Group 1, in which the activities were performed with the usage of the dashboard. The line **No-Context** contains the statistics of effort of the Group 2 where the activities were implemented without the usage of the dashboard. The mean time expended for Group 1 to implement both maintenance activities was 19.67 minutes, while the Group 2 spent, on average, 48.47 minutes. We perceived a significant difference between the groups in terms of effort. The Group 1 finalized the maintenance activities, on average, 28.28 minutes (40.58%) faster than the Group 2. Our initial expectation was that the effort invested by Group 1 could be lower than Group 2. This expectation was confirmed. We can consider that the usage of contextual information with qualitative information reduced the maintenance effort to perform the change requests when compared to the counterpart. The next step is to test whether this difference is statistically significant.

Table 4: Descriptive statistic for maintenance effort

	N	**Min**	**Med**	**IQR**	**Max**	**Mean**	**SD**
Context	30	2	17	20.25	70	19.67	16.12
No-Context	30	3	30.35	47	180	48.47	47.87
Diff	0	1	13.50	26.75	110	28.28	31.75

N: number of tasks performed, Min: minimum, Med: median,
Max: maximum, SD: standard deviation, IQR: interquartile range

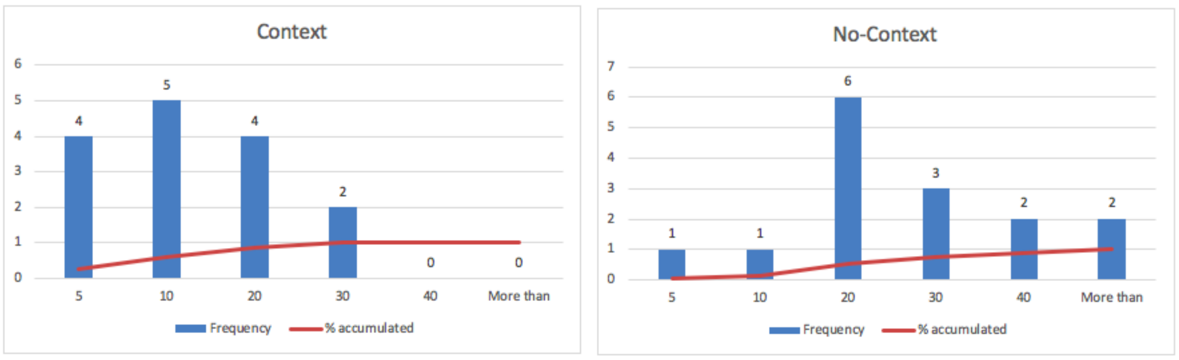
To apply the proper hypothesis test we used the Kolmogorov–Smirnov and Shapiro–Wilk tests (Devore and Farnum, 1999) to check the normal distribution of the data. Table 5 shows the results. Given that the p-values are lower than 0.05, then the collected data are not normally distributed. Thus, the Mann-Whitney test was applied to test the H_{2-1}. The *p-value* obtained by the test was 0.002. Considering that the *p-value* is lower than 0.05, then there is sufficient evidence to reject the null hypothesis. Therefore, the results suggest that the maintenance effort invested by subjects for implementing change requests using contextual information is significantly lower than the effort invested without using contextual information.

Table 5: Test of Normality

	Kolmogorov-Smirnov[a]			**Shapiro-Wilk**		
	Statistic	**df.**	**p-value**	**Statistic**	**df**	**p-value**
Context	0.161	30	0.046	0.860	30	0.001
No-Context	0.278	30	0.000	0.777	30	0.000

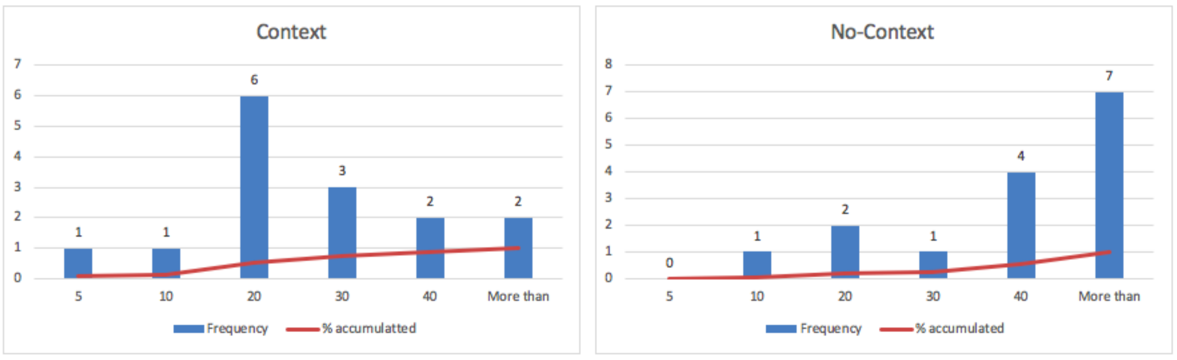
We also evaluated the effort invested by the subjects in each task. Figure 9 shows two histograms regarding the **task 1**. The first histogram (Context) represents the distribution of effort in minutes of Group 1, in which the maintenance task was performed with the usage of the dashboard. The majority (13 of 15) of subjects finished the implementation in up to 20 minutes. In contrast, just 8 of 15 subjects finished the activity in up to 20 minutes for the Group 2 (histogram No-Context). The impact of using contextual information with qualitative information in the implementation of task 1 was similar to the total effort time. It means the subjects invested less time to implement the changes when using the dashboard.

Figure 9: The effort frequency for task 1



The effort invested by the subjects in the **task 2** was also reduced for the subjects that used the contextual information with qualitative information. The Context histogram (Figure 10) shows that 11 of 15 subjects finished the maintenance task in up to 30 minutes. By contrast, just 4 of 15 subjects that implemented the task without the dashboard, finished the implementation in up to 30 minutes (histogram No-Context). Maybe, the effort difference of task 2 is higher than the first one due to the complexity. The change request of task 1 is simpler in terms of complexity and objects involved.

Figure 10: The effort frequency for task 2



4.3. RQ3: Maintainability

Table 6 shows the maintainability rate of the source code changed by the subjects during in the maintenance activities. The line **Context** represents the statistics of maintainability rate of Group 1, in which the activities were performed with the usage of the dashboard. The line **No-Context** contains the statistics of maintainability rate of the Group 2 where the activities were implemented without the usage of the dashboard. The average of source code maintainability rate changed by Group 1 was 0.8133 while the source code maintainability rate produced by Group 2 was, on average, 0.6133. We perceived a significant difference between the source code maintainability rate. The Group 1 produced a source code, during the maintenance activities, with an average rate 24.60% higher than the produced rate by Group 2. Our initial expectation was that maintainability of the source code produced by Group 1 could be higher than Group 2. This expectation was confirmed. We can consider that the usage of contextual information with qualitative information improved the maintainability of produced source code when compared to the counterpart.

Table 6: Descriptive statistic and test of normality

	Descriptive statistic					Shapiro-Wilk		Mann-Whitney test			
	N	**Min**	**Med**	**Mean**	**SD**	**Statistic**	**p-value**	**U**	**W**	**Z**	**p-value**
Context	15	0.40	0.8000	0.8133	0.15976	0.782	0.002	59	179	-2.338	0.026
No-Context	15	0.20	0.6000	0.6133	0.24456	0.914	0.155				
Diff	0	0.20	0.2000	0.2000	0.08480	0.132	0.153				

N #tasks, Min: minimum, Med: median
SD: standard deviation range

We have used the Shapiro–Wilk test (Devore and Farnum, 1999) to check the normal distribution of the data. Table 6 shows the results. Since the results indicated deviations from normality, the Mann-Whitney test was applied to test the H_{3-1}. We can be concluded that the maintainability rate in the contextual information group was statistically significantly higher than the group without contextual information (U = 59, p-value = 0.026).

4.4. Additional Discussion

This section summarizes our findings and outlines some implications for practitioners and researchers. The implications can help the researchers that intend to characterize software context in their empirical studies and also developers who deal with maintenance activities and need context information for that.

The conducted experiments allowed to conclude that the contextual information brought the following benefits for developers and researchers:

- **Uncovering context information for preventing critical bugs**. The study indicated that the developers' awareness about the application is increased through software contextual information. This indication comes from the results of *RQ1: Correctness and Contextual Information* where there is a statistically significant difference in the correctness measures associated with the usage of contextual information through the dashboard. Our perception is that, with more awareness about the possible impacts of a wrong implementation, the developers tend to change the source code more carefully and with more ownership about the changes.
- **The availability of context information for reducing effort**. We perceived a difference (on average, 28.28 minutes) between the groups in terms of effort. This indication resulted of *RQ2: Effort and Contextual Information* and suggested that the dashboard accelerated, for example, the searching time of source codes to be changed. Maybe, the history of last changes brought to the developers similar changes

previously performed and the objects changed. Without the contextual information, the searching time of objects to be changed would depend on the developer's experience and source code reading. Another aspect that reduced the implementation effort was the information about the design patterns, once one of the tasks demands changes of factory class.

- **Providing actionable information for preventing architecture degradation**. The information about design patterns and non-functional requirements implemented in the application advises the developer that there are important aspects already considered in the application, such as security and performance needs. These aspects are tight related to the application's architecture and the study indicated that the developers were aware about that when implementing the change request. The results of *RQ3: Maintainability* indicated that the usage of contextual information improved the maintainability of produced source code when compared to the group that did no use the dashboard.

However, to convince industry about the validity and applicability of the results of controlled software engineering experiments is challenge. Sjøberg et al. (2003) believe that the tasks, subjects and the environments should be as realistic as practically possible to achieve it. According to Cartaxo et al. (2015), some studies fail to present rigorous empirical evidence about their findings due to the lack of information about the context in which the studies were conducted. Additionally, the lack of contextual information is one of the main obstacles to replicate experiments.

Based on that, we also investigated whether the aforementioned results could be explained based on some information collected during the experiment. During the experiment, the subjects provided some qualitative data about themselves through a questionnaire. For instance, the subjects informed their current employment, the experience in software development and the academic qualifications. Analyzing the answers, we notice that 53% (16/30) of the subjects have no more than 2 years of experience in software development. Considering this fact, we analyzed additionally how the usage of contextual information influenced the same variables evaluated for all subjects.

We built a new data set containing the correctness, effort and maintainability of the subjects that have up to 2 years of experience developing software. To analyze these data, we did not perform hypothesis testing or other advanced analysis. We summarized the correctness frequency in this group and calculated the average of maintainability rate as well as the average of the effort expended during the experiment. Table 7 shows that the subjects (with experience up to 2 years) obtained similar results in the experiment compared with all subjects. In other words, the usage of contextual information improved the correctness and maintainability as well as reduced the maintenance effort for the subjects with less experience.

Table 7: Additional analysis

	N	Correctness	Mean Maint	Mean Effort
Context	8	14	0.575	43
No-Context	8	10	0.475	80
Diff	0	4	0.100	37

N #subjects, Correctness: correctness score
Mean Maint: mean of maintainability rate
Mean Effort: mean of effort in minutes

Additionally, we have observed no significant differences in the maintainability metrics (lines of code and cyclomatic complexity for example) extracted from the produced source codes. These metrics were extracted from all produced code by the subjects using the *Google CodePro AnalytiX*. We suspect that, these metrics were not impacted due to the size of the application under experiment and the nature of the maintenance activities proposed. The application under experiment can be considered small, in terms of number of classes (10) and lines of code (491 lines), and the proposed changes have no potential to change significantly the maintainability metrics. As we had a short time to count with the subjects, the changes requested could not be so complex as to significantly modify the application.

5. Threats to Validity

This study has a number of threats to validity that range from statistical conclusion validity, construct, internal, and external threats. This section discusses the strategies used for managing these threats.

5.1. Statistical Conclusion Validity

We minimized this threat by checking whether the variables (independent and dependent) were submitted to suitable statistical methods. The evaluation checked (1) whether the presumed cause and effect covary and (2) how strongly they covary (Cook and Campbell, 1979). Considering the first inferences, we may wrongly conclude that there is a causal relation between the variables when, in fact, they do not. We may also incorrectly state that the causal relation does not exist when, in fact, it exists. With respect to the second inference, we may incorrectly define the significance of covariation and the degree of confidence that the estimate warrants (Campbell and Russo, 1998).

We minimized the threats to the causal relation between the research variables studying the normal distribution of the collected sample. Thus, it was possible to verify whether parametric or nonparametric statistical methods could be used. For this purpose, we used the Kolmogorov-–Smirnov and Shapiro–Wilk tests (Devore and Farnum, 1999) to check the normal distribution of the data. Hence, we are confident that the test statistics were applied correctly, as the assumptions of the statistical test were not violated.

We tested all hypotheses considering the significance level at 0.05 level (p-value ≤ 0.05). In addition, we followed some general guidelines to improve conclusion validity (Trochim, 2006). First, we tried to obtain a significant number of subjects to improve the statistical power. Second, the subjects used pieces of realistic software applications. These improvements reduced "errors" that could obscure the causal relationship between the variable under study. Consequently, the improvements brought a better reliability for our results.

5.2. Construct Validity

The construct validity refers the degree to which inferences are warranted from the observed cause and effect operations included in our study to the constructs that these instances might represent. Considering these aspects, we evaluated (1) whether the quantification methods of the dependent variables are correct, (2) whether the quantification was accurately done, and (3) whether the usage of the IDE tool to implement the maintenance tasks can face the validity of our results.

We quantified the dependent variable *Effort* based on the time (in minutes) invested by the subjects to perform each of maintenance activities, while the correctness and maintainability rate based on a suite of metrics. We quantified the correctness by executing the existing unit testing in the application and checking the success rate, while the effort was recorded at the beginning and at the end of each experimental task. The maintainability rate was quantified through cycles of code reviews and, according to the code review, a final score was assigned to each point of change.

The authors have worked together to guarantee that the quantification of the variables was correctly performed. We checked whether the collected data were aligned with the objective and hypotheses of our study. The quantification procedures were carefully planned, and followed well-known quantification guidelines (Wohlin et al., 2012; Kitchenham et al., 2008; Kitchenham, 2007).

Another threat that we have controlled is if the use of the implementation IDE might unintentionally influence the results. All subjects have used the Eclipse IDE (Eclipse, 2016) to implement the changes requested by the experiment. The use of the Eclipse tool might jeopardize the results; as specific resources of the tools might influence the subjects during the execution of the experimental tasks. Considering this scenario, we have taken the precaution of making experimental decisions and seeking a study design that does not affect the results. First, the nature of the change requests did not require that the subjects understood the resources/details of the IDE tools. Second, the size of the application under experiment and the complexity of the implementation tasks were managed so that the use of these tools might not intentionally reduce (or exacerbate) the implementation effort. Therefore, we believe that the use of the Eclipse IDE tool did not impose threats to the validity of our experimental results.

5.3. Internal Validity

Inferences between our independent variable (usage of contextual information through a dashboard) and the dependent variables (correctness, effort and maintainability) are internally valid if a causal relation involving these variables is demonstrated (Wohlin et al., 2012; Shadish, W., Cook, T., Campbell, 2005). Our study met the internal validity because: (1) the temporal precedence criterion was met, i.e., the implementation of maintenance tasks preceded the correctness and maintainability rate as well as the effort implementation; (2) the covariation was observed, i.e., the use of contextual information led to varying accordingly to the implementation effort; and (3) there is no clear extra cause for the detected covariation. Our study satisfied all these three requirements for internal validity.

We also analyzed if the internal validity can be supported by other means. First, we performed some cases for demonstrating how the dependent variables are being exclusively affected by the independent variable. Second, we have observed that the collected values for the correctness, maintainability rate and implementation effort were confidently caused by the usage of contextual information through a dashboard.

Next, usually the confounding variable is seen as the major threat to the internal validity (Mitchell and Jolley, 2012). That is, rather than just the independent variable, an unknown variable unexpectedly affects the dependent variable. Thus, a pilot study was carried out to make sure that the dependent variables were not affected by any existing variable other than the use of the contextual information. During this pilot study, we tried to identify which other variables could affect the dependent variables, such as the experience of the subjects in software development, which was evaluated in the Section 4.4.

5.4. External Validity

External validity refers to the validity of the obtained results in other broader contexts (Mitchell and Jolley, 2012). That is, to what extent the results of this controlled study can be generalized to other realities, for instance, with different qualitative information, complexity of the code to be changed, with more experienced developers and quantifying other inconsistencies. As this study was not replicated yet, we made use of the theory of proximal similarity (proposed by Campbell and Russo (1998)) to identify the degree of generalization of the results. The goal is to define criteria that can be used to identify similar contexts where the results of this study can be applied.

Some criteria are shown as follows. First, the developers must have a basic knowledge about the software's domain under the study. The maintenance tasks should be implemented for evolving software applications; more specifically, evolution based on addition, exclusion, derivation, and change features. It is important to highlight that the software application used was small. Given that these criteria may happen in mainstream software development, we conclude that the results of our study may be generalized, at some point, to other contexts that are more similar to these requirements (Farias et al., 2012, 2015).

6. Related Works

This section presents some related studies, which define context for software and use this definition on software-development process to improve the situation awareness of developers. Section 6.1 describes the related studies and their contributions. Section 6.2 discusses evaluation criteria and comparison of these criteria in order to figure out some research opportunities related to the objective of this study.

6.1. Related Works Using Context on Software Development

Kersten and Murphy (2006) defined task context as the information that a software developer needs to know to complete that task. Each element and relationship in the study model correspond to a weighting of its relevance to that task. The task context is used by Mylar Elipse plug-in (renamed currently to Mylyn project) to improve the developer productivity when he needs to change the current task for another. The plug-in slices the project vision showing only the relevant artifacts to complete the current task. The histories of the programmer's interactions with a source code and related artifacts are also used by Sawadsky and Murphy (2011) to discover relevant code examples from the web through the Fishtail Eclipse plug-in. This plug-in used as base the Mylar plug-in described before. The key pieces of Fishtail's architecture include: (1) a task context manager component for tracking the degree-of-interest of program elements; (2) a query generator component for constructing keywords for program elements in the task context; (3) a query executor component for sending queries to a search engine and prioritizing results; and (4) a result display component which surfaces in search results within the IDE.

In the same line of task management and recovery, Parnin and Görg (2006) proposed an approach for capturing the context relevant to a task from programmer's interactions with an IDE. This information is then used to aid the programmer recovering the mental state associated with a previous task and to support the development activities using recommendation techniques. Their approach is focused on analyzing the interactions of the programmer with the source code, in order to create techniques for supporting mental recovery and source code exploration.

Leano et al. (2014) defined a task context using a hybrid approach. The files that are going to be edited are considered as part of task context and developers spend a lot of time to identify these files. So, the idea of the hybrid technique is to use a combination of information retrieval (IR), data mining and textual analysis techniques to determine the files which should be edited to accomplish a task. For example, when the developer starts editing an artifact, similar stored traces related to the artifact will be analyzed in order to provide a possible set of files to be changed. To determine the full set of files that are likely to be changed for a task the following approaches can be used: Commit Graphs, CrowdSourcing and Expert inputs.

Strathcona Eclipse plugin (Holmes and Murphy, 2005) considers as structural context the same artifact elements of task context; however, the usage of this kind of context is to suggest code samples to a developer by querying a repository in order to find similar source code usage. The main idea of this study is to facilitate the usage of frameworks and APIs. Strathcona works by extracting the structural context of source code entities. This structural context includes the method's signature, the declared type and parent type, the methods called, the name of fields accessed, and the types referred by each method. The extracted structural context can be used in two ways: (1) as an automatic query that describes the source code fragment for which the developer requests support; and (2) to build a database containing the structural context of classes.

Gasparic et al. (2017) presented a context model which captures various situations in which developers interact with an IDE. This context model can be used to support and enhance user interaction with the IDE or to improve the accuracy and timing of recommendations produced. The suggested context model consists of thirteen contextual factors, namely, variables with precise domains of possible values to be used to identify the context. The authors characterize developer situations from several perspectives as described bellow:

- Who: The contextual factors in this category capture general information about the developer who is interacting with an IDE.
- What: The contextual factors in this category capture information about what developers are doing with an IDE and which project artifacts are used during or affected by their actions.
- When: The contextual factors in this category capture temporal aspects of developer's interaction with an IDE considering the time of the day and day of the week which describe when the work is being performed.
- Where: The contextual factors in the *where* category capture information about the environment with which a developer is interacting to describe which parts of the IDE are available and which ones the developer is using.

According to Latoza et al. (2014), software development tasks require many types of context. Developers must know context such as where features are implemented, how to implement changes consistent with an architecture and design, and which developers to ask questions. For example, when writing a function, developers must understand the context in which it is used. When calling a function, developers must understand what assumptions it makes the system's state and the effects that this execution may cause. According to the authors, much of the context required in common programming tasks can be captured in the interfaces of functions, enabling tasks to be performed modularly on functions in isolation.

Requirements, dependent tasks, discussions and knowledge exchanges about tasks and artifacts are considered as part of software context. Tillmann et al. (2014) identified the use of this context information in Code Hunt tool. Code Hunt[2] is a web-based serious gaming platform in which players write code to advance through levels. The types of context identified in Code Hunt were categorized as follows: earlier solved coding duels, the secret code segment of the coding duel, the playing history of the coding duel, the coding duel being modified by the player, the input-output pairs reported to the player, and the hint reported to the player.

Martie and Hoek (2014) used contexts to help developers on code searching activity. Searching for source code on-line is a common activity in programming. Algorithms, language examples, and API usage examples are all searched during programming. They have developed a new code search engine called CodeExchange to directly support query reformulation. CodeExchange's interface presents meta-data about the code results, highlights semantic properties of the code results, and allows the programmer to use these meta-data

[2]https://www.codehunt.com/

and properties directly to reform the ongoing query. As such, the programmers can incrementally reform their queries not just with keywords but with the results returned, i.e., they search in context.

StackMine (Zhang et al., 2013) is a Microsoft project that identifies high-impact program execution patterns from a large number of trace streams based on sequences of function calls that happen during program execution. This project helps developers to identify a performance-bottleneck through analytics techniques reducing the human investigation effort by 90 percent. The build process of StackMine was done using an iterative flow between researchers and practitioners where the constant feedback was a valuable information to product improvements and usability.

Haron and Syed-Mohamad (2015) proposed the TDCA model, which focuses on integrating test coverage and defect coverage data to assist managers to make decisions. These data are extracted from source code artifacts under test, unit test cases from test suit tools and reported defects from a defect tracking database. After analyzing these data, the model shows the result through a dashboard named Test Analytics Dashboard. This dashboard demonstrates the information using a Bubble chart applying a qualitative approach. The model was implemented as an Eclipse plugin and its evaluation was performed by a case study on the Apache POI. The proposed model was compared to some test coverage tools such as Clover[3] and SonnerQube[4].

Some studies define context widely and not only to software artifacts. Petersen and Wohlin (2009) proposed a checklist for the description of context, consisting of context facets (product, process, practice, tools, people, organization, market) and related context elements based on a literature review. This checklist aims to help researchers to take informed decisions on what to include and not to include.

Antunes and Gomes (2009) proposed an approach for capturing contextual information from developers where it should take into account the whole environment that supports their work. This means that contextual information should be retrieved from all the applications which the developer typically deals with, such as an IDE, a set of office tools, a PIM (Personal Information Manager) and even the operating system itself. Most of the tools referred provide some kind of plug-in integration, which largely facilitate information retrieval. The information gathered across the various applications should then be centralized and coherently integrated into the context model, so that a snapshot of the context model of the user is available at any point in time.

In another study Antunes et al. (2011) considered that software context developer takes into account all dimensions that characterize the work environment of the developer. These dimensions can be represented as a layered model with four main layers: personal, project, organization and domain. The authors consider that the main sources of this of contextual information are project management tools. These tools store a big amount of explicit and implicit information about the resources produced during a software development project, how the people involved relate with these resources and how the resources relate to each

[3]https://confluence.atlassian.com/display/CLOVER/About+Clover+code+metrics
[4]http://www.sonarqube.org/

other. They have developed a prototype, in the form of an Eclipse plug-in, to show how the context information can be integrated into an IDE and used to help developers.

Carlson et al. (2016) proposed a context model that identifies which context properties are important for efficient and effective decision making and enable documentation of architectural decisions. Initially, five categories structure the model, namely organization, product, stakeholder, development methods & technologies and market & business. These are considered essential for studying the contextual elements of the phenomenon of architectural decision making and the effectiveness of modern, complex, strategic, tactical and operational decisions. The five main categories of the context model were described in the following way:

- Organization: Information that characterizes the organization type in which the implementation is carried out or intended for, i.e., structure, model of management, distribution, etc.
- Product: It describes the properties of the system developed, and all contextual information related to the current state of product.
- Stakeholder: It describes the type of organizations or people that might affect an architectural decision or that are affected by the decision. These are not taking part in the decision making process directly, but can represent the end-users of the decision, and thus are opt to be differentiated from the decision makers.
- Development method & technology: This category covers any systematic approach or technology used in or by the organization and is affecting the development of the product.
- Market & business: It represents the current state of the market and the business in general, outside the organization boundaries, i.e., involves customers, competitors, partners, ecosystems, etc.

6.2. Comparative Analysis of the Related Works

In this section, a comparison of related work is described considering six groups of characteristics, highlighting the most important, namely, the **Context Definition**. Table 8 organizes these characteristics in evaluation criteria and presents a comparison of the studies described before. These criteria were researched in order to understand the state of the art regarding the studies related to the usage of context information on software-development process.

The comparison criteria are:

1. **Context Definition**: This group of criteria evaluates how the study has defined formally the context information, and what kind of information was used on this. The criteria of this group are:
   - *Context definition/formalization*: this criterion evaluates if the study describes a formal definition of context applied to software development;

- *Context related to artifacts*: this criterion evaluates if the study considers information captured from the interaction between developers and IDEs and the artifacts' relationship as part of the context;
- *Context related to developers*: the developer has to deal with various kinds of resources at the same time, such as source code files, specification documents, bug reports, organization culture, software-development methodologies and others to accomplish determined tasks. The studies are evaluated by this criterion, considering this information as part of the context;
- *Context related to projects*: a software-development project is an aggregation of a team, a set of resources and a combination of explicit and implicit knowledge that keeps the project running. Usually, projects have tight deadlines, aggressive scopes and other restrictions and assumptions, which are so relevant to a software development. This criterion evaluates the usage of this data as part of the context;
- *Context related to software operations*: information, such as the number of clients who use a certain feature, the interaction between the feature and the computing infrastructure, performance and others can help developers to have a better understand about the context and the code under development. The usage of this information by the studies is evaluated by this criterion.

2. **Type of study**: This criterion considers the type of researched study. The types considered are survey, case study, controlled experiment and mapping study;
3. **Type of Model**: The type of model proposed by the researched studies is evaluated by this criterion. The type of models considered are architecture, conceptual model, ontology and algorithm;
4. **Tool support**: This criterion evaluates if the researched studies propose some tool. Prototype, IDE plug-in or heuristic were considered in this criterion;
5. **License**: The proposed tool license is evaluated by this criterion considering as open source or private;
6. **Visualization**: This criterion evaluates the approach of providing context information. The considered approaches are analytics, dashboards, structured data and views.

Table 8: Related work comparison

	Context Definition					Type of Study				Type of Model				Tool Support				License		Visualization		
Study	**Definition/Formalization**	**Related to artifacts**	**Related to developers**	**Related to projects**	**Related to software *operations***	**Survey**	**Case Study**	**Controlled Experiment**	**Mapping study**	**Architecture**	**Conceptual model**	**Ontology**	**Algorithm**	**Prototype**	**Tool**	**IDE Plugin**	**Heuristic**	**Open Source**	**Private**	**Analytics dashboards**	**Structured data**	**Views**
GASPARIC et al. (2017)	∼	√	√	–	–	–	√	–	–	–	√	–	–	–	–	√	–	∅	∅	∅	–	√
CARLSON et al. (2016)	–	√	√	√	√	–	–	–	–	–	–	–	–	–	–	–	–	∅	∅	∅	–	–
HARON; SYED-MOHAMAD (2015)	–	√	√	–	–	–	√	–	–	√	–	–	–	–	–	√	–	∅	∅	√	–	–
LEANO; KASI; SARMA (2014)	∼	√	√	–	–	–	–	–	–	–	√	–	–	–	–	–	√	∅	∅	–	–	–
TILLMANN et al. (2014)	–	√	√	√	–	–	–	–	–	–	√	–	∼	–	√	–	√	√	–	–	√	–
MARTIE; HOEK (2014)	∼	√	√	–	–	–	–	–	–	√	–	–	–	∅	√	–	–	√	–	–	√	√
LATOZA; TOWNE; HOEK (2014)	∼	√	√	–	–	–	–	–	–	–	–	–	–	–	√	–	–	√	–	–	√	–
ZHANG et al. (2013)	∼	√	√	–	√	–	√	–	–	∅	∅	∅	√	∅	√	∅	∅	–	√	√	√	–
CLARKE; O'CONNOR (2012)	∼	√	√	√	√	–	–	–	√	–	√	–	–	∅	∅	∅	∅	∅	∅	∅	∅	∅
SAWADSKY; MURPHY (2011)	–	√	√	–	–	–	–	–	–	–	–	–	√	√	–	√	–	∅	∅	–	√	√
ANTUNES; CORREIA; GOMES (2011)	∼	√	√	√	–	∅	∅	∅	∅	∼	√	–	√	√	–	–	–	∅	∅	–	√	√
ANTUNES; GOMES (2009)	–	√	√	√	–	–	∼	–	–	∼	–	∼	–	–	–	–	–	∅	∅	∅	∅	∅
KERSTEN; MURPHY (2006)	√	√	√	–	–	–	–	√	–	√	–	∼	√	–	–	√	√	√	–	–	√	√
PARNIN; GÖRG, 2006	∼	√	√	–	–	–	√	–	–	–	–	–	√	∅	∅	∅	∅	∅	∅	∅	∅	∅
HOLMES; MURPHY (2005)	–	√	√	–	–	–	√	–	–	–	–	–	√	–	–	√	√	√	–	–	√	√

Notes: (√) Supported (-) Not supported (∼) Partly supported (∅) Not applicable

As shown in Table 8, all studies consider context information related to software artifacts and developers. Only the study of Kersten and Murphy (2006) defines formally context, but its definition is limited to the developer's interaction with development IDE, disregarding other aspects such as project information or software operations.

Gasparic et al. (2017) proposed enhancements in the developer interaction with the Eclipse IDE and the context model were evaluated their by a case study. The evaluation showed that context affects the usage of IDE commands on the individual level, as well as on the group level. The authors plan to evaluate the model also with a bigger and more diverse group of IDE users, to improve the external validity of their conclusions.

The study of Zhang et al. (2013), through the StackMine tool, considers information about software operations (the result of programs performance evaluation) as relevant to compose the software context. The proposed tool provides the performance evaluation results to speed problem analysis in production environments systems done by developers.

The studies of Carlson et al. (2016), Antunes et al. (2011) and Antunes and Gomes (2009) define context for software development more widely, considering the main aspects listed in the literature. The studies consider that the context information should contain more than the data extracted from the developer's interaction with the development IDE. This means, the context information should consider personal aspects, organizational factors and projects where the requirements, assumptions and constraints make up the framework of information capable of improving the productivity of developers. The software operations dimension is not considered by the studies cited, and there is no formal definition or context model in the form of ontology or class diagram.

With a similar approach, Clarke and O'Connor (2012) propose a reference framework of situational factors which affect the software development process. This framework is grouped in 8 classifications: Personnel, Requirements, Application, Technology, Organization, Operation, Management and Business. They resulted to the 44 factors and 170 subfactors, from which the authors acknowledge that their scope of domains was restricted and did not include among other topics. This study enables researches to access a broad, systematically developed initial framework, which can be used as a reference for the situational factors affecting the software development process. Additionally, software development practitioners can access the proposed study as a check list of the important considerations for their software development process.

Haron and Syed-Mohamad (2015), Zhang et al. (2013), Holmes and Murphy (2005) and Parnin and Görg (2006) have organized their studies as case studies. Kersten and Murphy (2006) conducted the study through the implementation of a controlled experiment. The remaining studies were organized with a concept different from the covered by the criterion type of study.

Considering the type of model criteria, it was found that none of the studies defined ontology or class diagram to represent software context. Most studies have proposed algorithms as the main feature of their proposal.

In terms of visualization, most studies deal with structured data and views in their models. The only exceptions are StackMine tool by Zhang et al. (2013) and TDCA by Haron and Syed-Mohamad (2015). These works provide information using the analytics approach, allowing developers real-time decision-making in the analysis of performance issues.

Finally, we see this paper as a first step in a more ambitious agenda to define and use software contextual information in a qualitative way so that developers can get better results in their maintenance activities. According to the studies of Zhang et al. (2013), Haron and Syed-Mohamad (2015) and Antunes et al. (2011), a useful model of software qualitative information should be structured considering more than artifact data in the context definition. In addition, the way to visualize this information is an important feature to be considered by a context qualitative model for software. As demonstrated in the Section 4, this combination has a high potential to help the developers during their maintenance activities becoming a research opportunity.

7. Concluding Remarks and Future Work

This paper reported a controlled experiment to grasp the effects of contextual information on correctness and maintainability of code changed as well as the effort on implementing these changes. It can be seen as first step to assess the effects of using contextual information through qualitative dashboards on the software maintenance in terms of correctness, effort and maintainability. The results of this controlled experiment suggest that the usage of software contextual information, through qualitative dashboards, improves the awareness of developers regarding the application and the development environment. This awareness improvement can be verified by the higher software correctness and maintainability as well as a lower implementation effort measured during the experiment. These results can be considered a first step on supporting the definition of a context model for software development. The entities, which compose this model, should consider the qualitative information provided in the experiment.

However, all development aspects were controlled by the experiment. All participants performed the same activities in the same versions of code in an artificial environment. On this way, further empirical studies are still required to investigate if our results can be confirmed in other contexts. As a future work, we plan to build a dashboard composed by contextual information of a real application. This dashboard will be used in a real development team, inside of a software company. In this second study, it will not be possible to control all variables. The change requests can occur according to a bug or a legal requirement without forecast. The developers will perform changes in different artifacts and versions. The main goal of this second study will be to evaluate how the usage of contextual information can be applied to real software maintenance activities helping the developers to do the right thing in a proper place. The understanding of the role that different kinds of contextual information impact in the decision-making process of developers also will be evaluated in this study. From the result of this second assessment, we will have a complete overview about this approach and how it can be useful in the industry.

8. Acknowledgments

This work was financed by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and CNPq/Brazil (National Council for Scientific and Technological Development - http://www.cnpq.br). We would like to acknowledge both institutions for their support. Finally, we would like to thank Unisinos (http://www.unisinos.br) for embracing this research.

References

Antunes, B., Correia, F., Gomes, P., 2011. Context Capture in Software Development. Arxiv preprint arXiv11014101.
URL http://arxiv.org/abs/1101.4101

Antunes, B., Gomes, P., 2009. Context-Based Retrieval in Software Development. Proc. of the Doctoral Symposium on Artificial Intelligence (SDIA 2009) of the 14th Portuguese Conference on Artificial Intelligence (EPIA 2009), 1–10.

Augustine, V., Hudepohl, J., Marcinczak, P., Snipes, W., 2017. Deploying Software Team Analytics in a Multinational Organization. IEEE Software 35 (1), 72–76.

Barbosa, J., Tavares, J., Cardoso, I., Alves, B., Martini, B., 2018. Trailcare: An indoor and outdoor context-aware system to assist wheelchair users. International Journal of Human-Computer Studies 116, 1 – 14.
URL http://www.sciencedirect.com/science/article/pii/S1071581918301381

Barbosa, J. L. V., Dec 2015. Ubiquitous computing: Applications and research opportunities. In: 2015 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC). pp. 1–8.

Barbosa, J. L. V., Martins, C., Franco, L. K., Barbosa, D. N. F., 2016. TrailTrade: A model for trail-aware commerce support. Computers in Industry 80, 43–53.
URL http://dx.doi.org/10.1016/j.compind.2016.04.006

Baysal, O., Holmes, R., Godfrey, M. W., 2013. Developer Dashboards: The Need for Qualitative Analytics. IEEE Software 30 (4), 46–52.
URL http://dx.doi.org/10.1109/MS.2013.66

Briand, L., Bianculli, D., Nejati, S., Pastore, F., Sabetzadeh, M., 2017. The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated. IEEE Software 34 (5), 72–75.

Buse, R. P. L., Zimmermann, T., 2012. Information needs for software development analytics. Proceedings - International Conference on Software Engineering, 987–996.

Campbell, D. T., Russo, M. . J., 1998. Social Experimentation. SAGE Classics.

Carlson, J., Papatheocharous, E., Petersen, K., April 2016. A context model for architectural decision support. In: 2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH). pp. 9–15.

Cartaxo, B., Almeida, A., Barreiros, E., Saraiva, J., Ferreira, W., Soares, S., 2015. Mechanisms to characterize context of empirical studies in software engineering. In: Experimental Software Engineering Latin American Workshop (ESELAW 2015). pp. 1–14.

Cazzola, W., Shaqiri, A., 2017. Context-aware software variability through adaptable interpreters. IEEE Software 34 (6), 83–88.

Clarke, P., O'Connor, R. V., May 2012. The situational factors that affect the software development process: Towards a comprehensive reference framework. Inf. Softw. Technol. 54 (5), 433–447.
URL http://dx.doi.org/10.1016/j.infsof.2011.12.003

Cook, T., Campbell, D., 1979. Quasi-experimentation: Design & Analysis Issues for Field Settings. Houghton Mifflin.
URL https://books.google.com.br/books?id=BFNqAAAAMAAJ

Damasceno Vianna, H., Barbosa, J., Sept 2014. A model for ubiquitous care of noncommunicable diseases. Biomedical and Health Informatics, IEEE Journal of 18 (5), 1597–1606.

Deitel, H. M., Deitel, P. J., 2010. Java: Como Programar. PRENTICE HALL BRASIL.
URL https://books.google.com.br/books?id=U5AyAgAACAAJ

Devore, J. L., Farnum, N., 1999. Applied statistics for engineers and scientists. Statistics Series. Duxbury Press.
URL https://books.google.com/books?id=450ZAQAAIAAJ

Dey, A., Abowd, G., Salber, D., 2001a. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. Human-Computer Interaction 16 (2), 97–166.

Dey, A. K., Abowd, G. D., Salber, D., December 2001b. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction 16 (2), 97–166.

Driver, C., Clarke, S., August 2004. Context-aware trails [mobile computing]. Computer 37 (8), 97–99.

Driver, C., Clarke, S., Oct. 2008. An application framework for mobile, context-aware trails. Pervasive Mob. Comput. 4 (5), 719–736.
URL http://dx.doi.org/10.1016/j.pmcj.2008.04.009

Dybå, T., Sjøberg, D. I. K., Cruzes, D. S., Sept 2012. What works for whom, where, when, and why? on the role of context in empirical software engineering. In: Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 19–28.

Eclipse, 2016. Eclipse eclipse. http://www.eclipse.org, accessed: 2016-03-19.

Endsley, M. R., 1995. Toward a theory of situation awareness in dynamic systems: Situation awareness. Human factors 37 (1), 32–64.
URL http://dx.doi.org/10.1518/001872095779049543

Farias, K., Garcia, A., Whittle, J., Chavez, C., Lucena, C., 2012. Evaluating the effort of composing design models: A controlled experiment. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7590 LNCS, 676–691.

Farias, K., Garcia, A., Whittle, J., von Flach Garcia Chavez, C., Lucena, C., 2015. Evaluating the effort of composing design models: a controlled experiment. Software and Systems Modeling 14 (4), 1349–1365.
URL http://dx.doi.org/10.1007/s10270-014-0408-2

Fu, T., 2011. A review on time series data mining. Engineering Applications of Artificial Intelligence 24 (1), 164 – 181, http://www.sciencedirect.com/science/article/pii/S0952197610001727.

Gasparic, M., Murphy, G. C., Ricci, F., 2017. A context model for IDE-based recommendation systems. Journal of Systems and Software 128, 200–219.

Haron, N. H., Syed-Mohamad, S. M., 2015. Test and defect coverage analytics model for the assessment of software test adequacy. In: 2015 9th Malaysian Software Engineering Conference (MySEC). IEEE, pp. 13–18.

Holmes, R., Murphy, G., 2005. Using structural context to recommend source code examples. Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., 117–125.
URL http://dx.doi.org/10.1109/ICSE.2005.1553554

Jedlitschka, A., Ciolkowski, M., Pfahl, D., 2008. Reporting experiments in software engineering. In: Guide to advanced empirical software engineering. Springer, pp. 201–228.

Jedlitschka, A., Pfahl, D., 2005. Reporting guidelines for controlled experiments in software engineering. In: International Symposium on Empirical Software Engineering. IEEE, pp. 95–104.

Kersten, M., Murphy, G. C., 2006. Using task context to improve programmer productivity. Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14, 1—-11.
URL http://dx.doi.org/10.1145/1181775.1181777

Kitchenham, B., 2007. Empirical paradigm - the role of experiments. In: Proceedings of the 2006 International Conference on Empirical Software Engineering Issues: Critical Assessment and Future Directions. Springer-Verlag, Berlin, Heidelberg, pp. 25–32.
URL http://dl.acm.org/citation.cfm?id=1767399.1767412

Kitchenham, B., Al-Khilidar, H., Babar, M. A., Berry, M., Cox, K., Keung, J., Kurniawati, F., Staples, M., Zhang, H., Zhu, L., 2008. Evaluating guidelines for reporting empirical software engineering studies. Empirical Software Engineering 13 (1), 97–121.

Latoza, T. D., Towne, W. B., Hoek, A. V. D., 2014. Harnessing the Crowd : Decontextualizing Software Work. Csd 2014, 2–3.

Lavallée, M., Robillard, P. N., 2015. Why Good Developers Write Bad Code : An Observational Case Study of the Impacts of Organizational Factors on Software Quality. IEEE/ACM 37th IEEE International Conference on Software Engineering Why, 677–687.

Leano, R., Kasi, B. K., Sarma, A., 2014. Recommending Task Context: Automation Meets Crowd. International Workshop on Context in Software Development.

Li, W., Eickhoff, C., de Vries, A. P., 2012. Want a coffee?: predicting users' trails. In: Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval. SIGIR '12. ACM, New York, NY, USA, pp. 1171–1172.
URL http://doi.acm.org/10.1145/2348283.2348524

Lientz, B. P., Swanson, E. B., 1980. Software maintenance management: A study of the maintenance of computer application software in 487 data processing organizations. Addison-Wesley.

Lou, J.-G., Lin, Q., Ding, R., Fu, Q., Zhang, D., Xie, T., 2013. Software analytics for incident management of online services: An experience report. 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 475–485.
URL http://dx.doi.org/10.1109/ASE.2013.6693105

Martie, L., Hoek, A. V. D., 2014. Context in Code Search.

Menzies, T., 2018. The unreasonable effectiveness of software analytics. IEEE Software 35 (2), 96–98.

Menzies, T., Zimmermann, T., 2013. Software analytics: so what? IEEE Software 30 (4), 31–37.

Mitchell, M., Jolley, J., 2012. Research Design Explained, 8th edn. Wadsworth Publishing.

Morse, D. R., Armstrong, S., Dey, A. K., 2000. The what, who, where, when, why and how of context-awareness. In: CHI '00 Extended Abstracts on Human Factors in Computing Systems. CHI EA '00. ACM, New York, NY, USA, pp. 371–371.
URL http://doi.acm.org/10.1145/633292.633518

Mostefaoui, G. K., Pasquier-Rocha, J., Brézillon, P., 2004. Context-Aware Computing: A Guide for the Pervasive Computing Community. ICPS '04: Proceedings of the The IEEE/ACS International Conference on Pervasive Services, 39–48.
URL http://dx.doi.org/10.1109/ICPS.2004.14

Murphy, G. C., 2018. The need for context in software engineering (ieee cs harlan mills award keynote). In:

Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE 2018. ACM, New York, NY, USA, pp. 5–5.
URL http://doi.acm.org/10.1145/3238147.3241987

Parnin, C., Görg, C., 2006. Building usage contexts during program comprehension. IEEE International Conference on Program Comprehension 2006, 13–22.
URL http://dx.doi.org/10.1109/ICPC.2006.14

Petersen, K., Wohlin, C., 2009. Context in industrial software engineering research. 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, 401–404.
URL http://dx.doi.org/10.1109/ESEM.2009.5316010

Port, D., Taber, B., 2017. Actionable Analytics for Strategic Maintenance of Critical Software: An Industry Experience Report. IEEE Software 35 (1), 58–63.

Rajlich, V., 2001. Software evolution: a road map. Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, 6.
URL http://dx.doi.org/10.1109/ICSM.2001.972705

Rosa, J., Barbosa, J., Kich, M., Brito, L., 2015. A multi-temporal context-aware system for competences management. International Journal of Artificial Intelligence in Education, 1–38.
URL http://dx.doi.org/10.1007/s40593-015-0047-y

Rosa, J., Barbosa, J. L. V., Ribeiro, G. D., 2016. ORACON: An adaptive model for context prediction. Expert Systems with Applications 45, 56–70.
URL http://dx.doi.org/10.1016/j.eswa.2015.09.016

Satyanarayanan, M., 2001. Pervasive computing: vision and challenges. IEEE Personal Communications 8, 10–17.

Sawadsky, N., Murphy, G., 2011. Fishtail: from task context to source code examples. Proceedings of the 1st Workshop on . . . , 48–51.
URL http://dx.doi.org/10.1145/1984708.1984722

Shadish, W., Cook, T., Campbell, T., 2005. Experiments and generalized causal inference. Experimental and quasi-experimental designs for generalized causal inference 100 (470), 1–81.
URL http://dx.doi.org/10.1198/jasa.2005.s22

Silva, J. M., Rosa, J. H., Barbosa, J. L., Barbosa, D. N., Palazzo, L. A., 2010. Content distribution in trail-aware environments. Journal of the Brazilian Computer Society 16 (3), 163–176.
URL http://dx.doi.org/10.1007/s13173-010-0015-1

Sjøberg, D. I., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanović, A., Vokáč, M., 2003. Challenges and recommendations when increasing the realism of controlled software engineering experiments. In: Empirical methods and studies in software engineering. Springer, pp. 24–38.

Sjøberg, D. I. K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Koren, E. F., Vokác, M., 2002. Conducting Realistic Experiments in Software Engineering (1325).

Smith, A., July 2008. Who controls the past controls the future - life annotation in principle and practice. Ph.D. thesis, University of Southampton.
URL http://eprints.soton.ac.uk/266554/

Tavares, J., Barbosa, J., Cardoso, I., Costa, C., Yamin, A., Real, R., 2016. Hefestos: an intelligent system applied to ubiquitous accessibility. Universal Access in the Information Society 15 (4), 589–607.
URL http://dx.doi.org/10.1007/s10209-015-0423-2

Tillmann, N., Halleux, J. D., Bishop, J., Xie, T., 2014. Code Hunt: Context-Driven Interactive Gaming for Learning Programming and Software Engineering. Engr.Illinois.Edu, 0–1.
URL http://web.engr.illinois.edu/{~}taoxie/publications/csd14-codehunt.pdf

Trochim, W. M., 2006. Research method knowledge base: improving conclusion validity.
URL http://www.socialresearchmethods.net/kb/concimp.php

van Solingen, R., Basili, V., Caldiera, G., Rombach, H. D., 2002. Goal Question Metric (GQM) Approach. John Wiley and Sons, Inc.
URL http://dx.doi.org/10.1002/0471028959.sof142

Vianna, H. D., Barbosa, J. L. V., Pittoli, F., November 2017. In the pursuit of hygge software. IEEE Software

34 (6), 48–52.

Wagner, A., Barbosa, J. L. V., Barbosa, D. N. F., Mar. 2014. A model for profile management applied to ubiquitous learning environments. Expert Syst. Appl. 41 (4), 2023–2034.
URL http://dx.doi.org/10.1016/j.eswa.2013.08.098

Wiedmann, T., Luis, J., Barbosa, V., 2016. RecSim : A Model for Learning Objects Recommendation using Similarity of Sessions 22 (8), 1175–1200.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A., 2012. Experiment process. In: Experimentation in Software Engineering. Springer Berlin Heidelberg, pp. 73–81.

Zhang, D., Dang, Y., Lou, J.-G., Han, S., Zhang, H., Xie, T., 2011. Software analytics as a learning case in practice. Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering - MALETS '11, 55–58.
URL http://dx.doi.org/10.1145/2070821.2070829

Zhang, D., Han, S., Dang, Y., Lou, J.-G., Zhang, H., Xie, T., 2013. Software Analytics in Practice. IEEE Software 30 (5), 30–37.
URL http://dx.doi.org/10.1109/MS.2013.94