PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Diogo Silveira Mendonça

Pattern-Driven Maintenance: A Method to Prevent Unhandled Latent Exceptions in Web Applications

Tese de Doutorado

Thesis presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Marcos Kalinowski
Co-advisor: Arndt von Staa

Rio de Janeiro, March 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Diogo Silveira Mendonça

Pattern-Driven Maintenance: A Method to Prevent Unhandled Latent Exceptions in Web Applications

Thesis presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the Examination Committee.

Marcos Kalinowski
Advisor
Departamento de Informática – PUC-Rio

Arndt von Staa
Co-advisor
Departamento de Informática – PUC-Rio

Alessandro Fabricio Garcia
Departamento de Informática – PUC-Rio

Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Guilherme Horta Travassos
UFRJ

Leonardo Gresta Paulino Murta
UFF

Rio de Janeiro, March 21th, 2019

Diogo Silveira Mendonça

Diogo Silveira Mendonça received his Bachelor degree in Computer Science from the Federal University of Rio de Janeiro (UFRJ) in 2006. He received his Master degree in Informatics from the Pontical Catholic University of Rio de Janeiro (PUC-Rio) in 2008. His main research interests are Software Engineering, Software Product Quality.

Ficha Catalográfica

Mendonça, Diogo Silveira

Pattern-driven maintenance : a method to prevent unhandled latent exceptions in web applications / Diogo Silveira Mendonça ; advisor: Marcos Kalinowski ; co-advisor: Arndt von Staa. – 2019.
128 f. : il. color. ; 30 cm

Tese (doutorado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2019.
Inclui bibliografia

1. Informática – Teses. 2. Manutenção de aplicações web. 3. Exceções não tratadas. 4. Confiabilidade. 5. Padrões. 6. Análise estática. I. Kalinowski, Marcos. II. Staa, Arndt von. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

To my parents, Aparecida and Eron,
my wife Taliha,
and my daughter Iasmin.

Acknowledgments

I would like to thank my advisors, without their dedication and hard work this thesis would not have been possible. To Professor Arndt von Staa, by his celerity, sense of practice, rigor, and his vast knowledge in software engineering that was applied during this thesis. To Professor Marcos Kalinowski by his ideas, technical advises, and detailed revisions that provided a different and differentiated perspective in this thesis.

I would like to thank my family, which supports me to achieve my objectives, including to conclude this thesis. To my parents, Aparecida and Eron, for always support, encourage, and help me whatever it takes. To my wife Taliha, by always being by my side, being my lifemate, and filling my heart with your love. To my daughter, Iasmin, to be the joy of my life. To my parents in law, Deise and Paulo, by staying with my daughter many times while I had been working in this thesis (to my mother and wife also for this). To my sister Michele, for the funny talks that we had, helping me cheer me up. To my sister in law Laiza, by always helping us when we have health problems.

I would like to thank the professional of CEFET/RJ, which from goodwill, directly contributed to this thesis. To Julliany Sales Brandão and Elielson Ribeiro, as chiefs of DTINF in different periods of time, allowed me to use of information systems of CEFET/RJ in this thesis. To the system analysists of DTINF that cooperated with this research, they are Daniel Oliveira, Tarcila Silva, Taiana Pereira, Enoch Cezar, and Marcio Ferreira.

I thank CNPq, for the financial support without which this work would not have been possible.

I thank all the members of the examining committee.

I thank all the professors of the informatics department of PUC-Rio, and the professor Guilherme Travassos from COPPE/UFRJ, for sharing their vast knowledge.

Abstract

Mendonça, Diogo Silveira. **Pattern-Driven Maintenance: A Method to Prevent Unhandled Latent Exceptions in Web Applications.** Rio de Janeiro, 2019. 128p. PhD Thesis - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Background: Unhandled exceptions affect the reliability, usability, and security of web applications. Several studies have measured the reliability of web applications in use against unhandled exceptions, showing a recurrence of the problem during the maintenance phase. Detecting automatically unhandled latent exceptions is difficult and application-specific. Hence, general approaches to deal with defects in web applications do not treat unhandled exceptions appropriately. **Aims:** To design and evaluate a method that can support finding, correcting, and preventing unhandled exceptions in web applications. **Method:** We applied the design science engineering cycle to design a method called Pattern-Driven Maintenance (PDM). PDM relies on identifying defect patterns based on application server logs and producing static analysis rules that can be used for prevention. We applied PDM to two industrial web applications involving different companies and technologies, measuring the reliability improvement and the precision of the produced static analysis rules. We also evaluated reuse of static analysis rules produced during PDM application on within- and cross-company software. Finally, we studied the effectiveness and acceptance of novice maintainers on applying the PDM method **Results:** In both industry cases, our approach allowed identifying defect patterns and finding unhandled latent exceptions to be fixed in the source code, enabling to eliminate the pattern-related failures and improving the application reliability completely. Some of the static analysis rules produced by PDM application were reused on within- and cross-company software. We identified knowledge and experiences that influence on effectively applying steps of the PDM method. Most of the novice maintainers find PDM useful, but not easy to apply, thus hindering PDM acceptance among novices. **Conclusions:** The results strengthen our confidence that PDM can help maintainers to improve the reliability for unhandled exceptions in existing web applications. We provide guidance on how to apply PDM, reuse the produced static analysis rules, and the knowledge and experiences needed to apply the PDM

method effectively.

Keywords

Maintenance of Web Applications; Unhandled Latent Exceptions; Reliability; Patterns; Static Analysis.

Resumo

Mendonça, Diogo Silveira. **Manutenção Orientada a Padrões: Um Método para Prevenir Exceções Latentes Não Tratadas em Aplicações Web.** Rio de Janeiro, 2019. 128p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Contexto: Exceções não tratadas afetam a confiabilidade, usabilidade e segurança em aplicações web. Diversos estudos mediram a confiabilidade de aplicações web em uso em relação a exceções não tratadas, mostrando a recorrência deste problema durante a fase de manutenção. Detectar exceções não tratadas latentes de forma automatizada é uma tarefa difícil e específica de cada aplicação. Assim, abordagens gerais para tratar defeitos em aplicações web não tratam exceções não tratadas latentes apropriadamente. **Objetivos:** Projetar e avaliar um método que possa suportar encontrar, corrigir e prevenir exceções não tratadas em aplicações web. **Método:** Nós aplicamos o ciclo de engenharia do *design science* para projetar o método chamada Manutenção Orientada a Padrões (PDM). PDM consiste em identificar padrões de defeitos se baseando nos logs do servidor de aplicação, produzindo regras de análise estática que podem ser utilizadas para a prevenção de defeitos. Nós aplicamos PDM em duas aplicações web na indústria envolvendo empresas e tecnologias diferentes, medindo a melhoria confiabilidade das aplicações e a precisão das regras de análise estática produzidas. Nós também avaliamos o reuso das regras de análise estática produzidas durante a aplicação do PDM em software da mesma empresa e de outras empresas. Finalmente, nós estudamos a eficácia e aceitação de mantenedores novatos aplicando o método PDM. **Resultados:** Nos dois casos industriais, nossa abordagem permitiu a identificação de padrões de defeitos e exceções não tratadas latentes para correção no código fonte, permitindo eliminar completamente as falhas relacionadas a exceções não tratadas latentes e melhorando assim a confiabilidade da aplicação. Algumas regras de análise estática produzidas pela aplicação do método PDM foram reutilizadas em software na mesma empresa e em outra empresa. Nós identificamos os conhecimentos e experiências que influenciam em aplicar os passos do método PDM de forma eficaz. A maior parte dos mantenedores novatos acharam o

método PDM útil, mas não fácil de aplicar, dificultando a aceitação do método entre novatos. **Conclusões:** Os resultados fortalecem nossa confiança que o PDM pode ajudar os mantenedores a melhorar a confiabilidade em relação a exceções não tradadas em aplicações web existentes. Nós disponibilizamos orientações sobre como utilizar o método, reutilizar as regras de análise estática produzidas, e quais conhecimentos e experiências são necessárias para aplicar o PDM com eficácia.

Palavras-chave

Manutenção de Aplicações Web; Exceções Não Tratadas; Confiabilidade; Padrões; Análise Estática.

Table of Contents

List of figures

List of tables

1 Introduction

1.1. Context and Motivation

Maintenance is the most costly phase in the software lifecycle (BOURQUE; FAIRLEY; OTHERS, 2014). Defect prevention and correction activities consume part of these resources. Additionally, the impact of failures in software in use can range from slight inconvenience to severe damage, including economic ones (JONES; BONSIGNOUR, 2011). Among those failures are the ones generated by exceptions that are not handled by the application, *i.e.*, *unhandled exceptions*.

Unhandled exceptions can affect software reliability, usability, and security. The reliability of a system is its ability to perform their required functions under stated conditions for a specific period of time (ISO, 2010). Reliability is affected when an exception is not handled correctly. Indeed, exception handling is a requirement for reliable web applications. Usability may also be affected; typically users do not receive proper messages to deal with the exceptional situation when it occurs. Furthermore, unhandled exceptions are listed as a common software weakness (CSW-248)[1], which if exploited by attackers may affect software availability and confidentiality[2].

In web applications, the web server logs into the error log, among other failures, those generated by unhandled exceptions. They can be identified by the HTTP return code 500 in the web server access log. Those logs have been previously used to measure the reliability of several web applications (KALLEPALLI; TIAN, 2001; GOŠEVA-POPSTOJANOVA et al., 2006), showing the recurrent occurrence of unhandled exceptions. However, the logs record only the unhandled exceptions thrown during software use. Hence, even if unhandled exceptions are not registered in the log, it is still possible for the web application to have source code that lacks exception handling, but that has not

[1] http://cwe.mitre.org/data/definitions/248.html

[2] http://capec.mitre.org/data/definitions/54.html

thrown exceptions yet. Throughout this thesis, we refer to this type of source code problem as *unhandled latent exceptions*.

There are some simplistic solutions to handle latent exceptions in existing web applications. In some technologies, such as Java Enterprise Edition, it is possible to implement a generic exception handler at the server-side of web applications. This type of handler catches all exceptions and presents a general error message within an error page. However, general exception handling does not provide proper error messages to the users, preventing them from taking recovering actions, thus affecting software usability. This kind of exception handling does not mitigate possible business losses caused by failures. Another solution would be handling all possible exceptions by adding specific handlers to all source code locations that might potentially throw exceptions. However, this solution may result in useless code (handlers that might never be used in practice) and have a high cost due to many source code locations to change and test. Additionally, changing code unnecessarily represents a waste of resources and might introduce new defects.

Training maintainers and inspecting source code to enforce exception-handling policies may be useful for dealing with unhandled latent exceptions. However, process-based approaches need continuous effort to be effective. People need to be retrained periodically, as well as new project members, and new source code that is coming from software evolution needs to be inspected. Automated approaches, on the other hand, might avoid this continuous effort expenditure by locating and alerting maintainers about unhandled latent exceptions. However, there are some challenges to locate unhandled latent exceptions automatically.

Identifying unhandled latent exceptions is an application-specific task. Each application has its own exception handling policies and architecture, which define when and where to handle exceptions. Some programming languages enable forcing exception handling at compile time, such as Java checked exceptions, assisting developers with this task, whereas other programming languages, such as Python, do not assist developers in handling exceptions. Additionally, the use of software libraries that may throw exceptions that are unknown to the developer increases the chance of unhandled latent exceptions. These application idiosyncrasies influence how exceptions should be handled in the application, thus influencing the identification of unhandled exceptions.

Consequently, it is difficult to detect unhandled latent exceptions automatically. Automated approaches for testing web applications (GAROUSI et al., 2013; DOGAN; BETIN-CAN; GAROUSI, 2014; LI; DAS; DOWE, 2014) and locating defects using static analysis (HECKMAN; WILLIAMS, 2011; MUSKE; SEREBRENIK, 2016) do not focus on unhandled latent exceptions, thus they are inadequate to treat this problem. Application-specific approaches (ERSOY; SÖZER, 2016) show only superficially how to create static analysis rules to find application-specific defects. They also do not inform the precision of the static analysis rules produced and how that precision can be improved.

This overall scenario motivates further investigations to help to pave the road towards effective prevention of unhandled latent exceptions in web applications.

1.2. Problem Statement and Research Methodology

To fail and to learn from failure are essential parts of the engineering discipline (PETROSKI; BARATTA, 1988). In this thesis, we aim to apply this principle to the (latent) unhandled exceptions problem, using logged failure information as the basis for learning how to prevent them.

Using the design science (WIERINGA, 2014) template, our problem can be stated as follows:

- **Improve** the reliability of web information systems that present failures caused by unhandled operational[3] exceptions
- **by** designing a method to automate the localization of unhandled (operational and latent) exceptions
- **that satisfies** high levels of precision and recall for localization
- **in order to** not only fix the existing defects (operational and latent) but also be used to prevent the reintroduction of the same type of defect during the software evolution.

[3] Unhandled operational exceptions are the exceptions which were exercised during software operation producing a failure, while the unhandled latent exceptions are the exceptions that may produce a failure but were not exercised in this way during software operation yet.

Our research methodology to address this problem is based on the design science engineering cycle (WIERINGA, 2014). The design science approach starts with idealized assumptions to produce an artifact that solves a practical problem. Afterward, engineering cycles are performed with controlled conditions, gathering experience to improve the artifact. Each engineering cycle relaxes the conditions of experimentation gradually by approximating them to practical conditions. Those cycles are performed until the artifact is ready to be used in practice.

In our case, we had some idealized assumptions drawn from our previous experience and knowledge of the unhandled exceptions problem and its related literature. Our assumptions at this early time were: (1) unhandled exceptions (operational and latent) form patterns in the source code of web applications, (2) each application has its own patterns, and (3) each specific defect pattern occurs several times throughout the source code.

We designed a method called Pattern-Driven Maintenance (PDM) to perform corrective and preventive maintenance of web applications against unhandled latent exceptions. In this method, the maintainer first uses the web server logs as sources to find software failures generated by unhandled exceptions; then an investigation is performed on the failures and in the application source code to identify source code patterns that trigger an unhandled exception, i.e., a defect pattern. Once such a pattern has been identified, the maintainer creates a static analysis rule that represents the defect pattern and uses a static analysis tool to locate its instances. After the pattern instances are found, they are evaluated by testing, revealing their latent defects. The testing activity not only enables correction of the defects but also assists in improving the precision of the static analysis rules, working as a learning cycle.

Once designed, we conducted investigations aiming to answer the following design science knowledge questions (WIERINGA, 2014) about PDM:

RQ1. (effect) What is the software reliability improvement achieved by fixing the located defects?

RQ2. (requirement satisfaction) What is the precision and recall of the automated defect localization?

RQ3. (sensitivity) Which factors influence the method application and precision of the automated defect localization?

RQ4. (sensitivity) In which scope rules created by applying PDM can be reused?

RQ5. (effect) What are the benefits of reusing rules created by applying PDM?

RQ6. (sensitivity) Which factors influence reusing rules created by applying PDM?

RQ7. (requirement satisfaction) How effective are maintainers applying PDM for preventing defects?

RQ8. (requirement satisfaction) Would maintainers accept to use PDM?

We performed three different studies to address those questions. First, we evaluated PDM effectiveness and sensitivity in preventing unhandled latent exceptions by applying it in two industrial cases. We measured the reliability against unhandled exceptions of both software before and after applying PDM (RQ1), evaluated the precision and recall of rules produced (RQ2), and reported our perceptions on the factors that influence PDM application (RQ3).

After applying PDM in two industrial software systems, we selected other similar software to evaluate the reuse of rules produced by the method. We selected three software systems, one within the same company and team that we applied PDM, and the other two with other companies and development teams. We evaluated in which ones the rules could be reused (RQ4), as well as the factors that influence rule reuse (RQ6). We also measured the precision of the reused rules and discussed the benefits found by reusing PDM-produced rules (RQ5).

Finally, we evaluated the effectiveness of novice maintainers in applying PDM and their acceptance of the method by making them apply PDM in an observational study. We measured the percentage of maintainers that correctly applied each step of PDM and compared the maintainers that correctly performed each step with others. Hence, we evaluated maintainers effectiveness (RQ7) and the skills needed to achieve it. We evaluated PDM acceptance by applying the technology acceptance model (TAM) questionnaire after maintainers used PDM (RQ8).

1.3. Thesis Outline

This thesis is organized as follows: Chapter 2 presents the background and related work. Chapter 3 presents the PDM method and a detailed example of the PDM application. Chapter 4 presents the evaluation conducted by applying PDM to two industrial web applications. Chapter 5 describes the evaluation of the reuse of rules defined by PDM in other web applications. Chapter 6 presents an observational study on the effectiveness of maintainers in applying PDM and their acceptance of the method. Finally, Chapter 7 concludes the thesis presenting our contributions and suggesting future work.

2 Background and Related Work

2.1.Introduction

In order to provide the background for understanding the rationale used in the PDM method, this section provides an overview on research related to unhandled exceptions in web applications, automated approaches for maintenance of web applications, and enforcement of exception handling policies. We discuss the applicability of these approaches to deal with the problem of unhandled latent exceptions. We restricted our scope of comparison to techniques and methods that do not use software documentation as a resource for automation. This restriction is due to practical reasons because software documentation usually suffers from problems such as nonexistence (SOUZA; ANQUETIL; OLIVEIRA, 2006), low quality (BRIAND, 2003; HUANG; TILLEY, 2003), or being outdated (FORWARD; LETHBRIDGE, 2002), thus typically not being a trustworthy resource during maintenance (SINGER, 1998; SOUSA; MOREIRA, 1998; DAS; LUTTERS; SEAMAN, 2007).

2.2.Exceptions and web applications

An exception is an event that causes the suspension of normal program execution (ISO, 2010). Exception handling is a program language mechanism that passes error information by throwing and catching exceptions (ISO, 2010). In an interactive information system, such as web applications, exceptional events must be handled avoiding abnormal termination of user interactions. In this way, throwing an unhandled exception is a defect. The thrown unhandled exception is an error – the consequence of exercising a defect. Catching the exception reports a failure since it corresponds to observing an error. A latent defect is the one that never has been exercised during test and operation resulting in a failure, while operational ones have already produced a failure during software operation or test.

Preventive maintenance is the modification of a software product after delivery to detect and correct latent faults (defects) in the software product before they become operational faults (defects) (ISO, 2010). In a web application, when an unhandled exception occurs, the event is typically registered in web server logs. In the web server access logs, an HTTP 500 return code[4] is registered in such situation, while error log registers the stack trace of the failure. Appendix B presents an example of an error log produced by a Java programmed (JEE) web application running in Tomcat[5] web server. However, each programming language may have a different format to print a stack traces, and each web server may have different fields and formats of logs. The practitioner should consult the documentation of programming language and web server used in the web application for more information about stack trace and logs formats, respectively.

2.3.Unhandled exceptions and reliability of web applications

Kallepalli and Tian (2001) propose the use of web server access logs to analyze the reliability and perform the statistical testing of web applications. In that work, the authors identified application failures using the HTTP response code recorded in the web server access logs. They used Nelson's (1978) model (2.1) and an approximation of mean time between failures (MTBF) (2.2) for reliability calculation. In both formulas, *f* is the number of failures and *n* the number of accesses, which represents the workload variable. Kallepalli and Tian also suggest directing the testing by the number of accesses to each URL. This strategy is called statistical web testing. However, the HTTP return code only informs the type of failure; therefore, further analysis is needed to identify the related defect. Statistical web testing is dependent on the operational profile of the software, failing to find defects in the less frequently accessed areas of the application.

$$R = 1 - \frac{f}{n} = \frac{n - f}{n} \qquad (2.1)$$

[4] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

[5] http://tomcat.apache.org/

$$MTBF = \frac{n}{f} \quad (2.2)$$

Tian et al. (2004) experimented other variables as the workload for calculation of reliability based on web server logs. They used bytes transferred, number of users and number of sessions as workload. However, as the number of users and number of sessions was derived from the internet protocol address (IP) field of the web server logs, they were imprecisely calculated.

Huynh and Miller (2005) presented an analysis of the return HTTP codes neglected by previous studies. Specifically, one of these codes was the HTTP 500 code, which is returned when a server-side script execution fails. This code is logged when an exception is thrown but not handled by the web application.

Goševa-Popstojanova et al. (2006) conducted a study on reliability and presence of defects in eight web applications. The access and error logs of the applications were analyzed to identify failures and the unique errors that originated those failures. The authors defined the concept of unique errors as a combination of the error message and the source file that generated the message. However, error messages can present parameters, such as variable names and values, and vary according to these parameters. The unique errors were used to assess the number of defects empirically and they did not consider identifying defect patterns.

Ma and Tian (2007) presented an adaptation of orthogonal defect classification (ODC) (CHILLAREGE, 1995) for classifying and analyzing errors of a web application with the intent of identifying problematic areas for focused reliability improvement. The method proposed combines attributes extracted from web server logs to classify defects. The attributes used were the response code, the file type, referrer type, agent type and observation time. The authors also introduced an analysis procedure to assess the risk and leverage of sub-classes of problems by evaluating their error rate and share. The error rate is defined as the ratio between the number of failures and the number of accesses of a particular class, while the error share is the percentage of a given class of errors. Although the authors presented new analyses, they did not evaluate causes in web application source code, thus they did not identify defect patterns.

Huynh and Miller (2009) partially replicated the work of Tian et al. (2004) considering other web applications. One of the applications studied had a strong reliability requirement, presenting few failures. Hence, reliability metrics were calculated by month and not by day. They also observed that Nelson's model had low representability for analysis when high-reliability is needed. Thus they used MTBF instead for analysis.

Banerjee et al. (2010) investigated the suitability of reliability measures with respect to their relevance in the context of service level agreements (SLA) of software as a service (SaaS). They concluded that web server logs filtration is essential for proper SaaS reliability calculation and agreement, since the counting of accesses to static files, such as images, masks the actual reliability of the web application.

Jaffal and Tian (2014) performed a reliability assessment of the transactions of a web application. A transaction is a sequence of information exchange and related work that is treated as a unit for the purposes of satisfying a request and for ensuring the data integrity (JAFFAL; TIAN, 2014). The reliability was assessed using transactions as workload, thus calculating the chance of one transaction being completed without errors. The authors used different data sources for calculating the reliability, including user sessions retrieved from the application database and unique failures extracted from application server logs. However, the intent of the study was only to calculate transactions reliability, and no further investigation was presented to discover failures causes.

Alannsary and Tian (2016) proposed time taken for a request completion as a workload variable for measure the reliability of SaaS web application. The time taken variable is typically available in the platform as a service (PaaS) or infrastructure as a service (IaaS) providers, since it is used for the cost of service calculation. The time is taken in requests that result in failure also have infrastructure cost; Alannsary (2016) showed how to calculate this cost.

2.4. Automated approaches for the maintenance of web applications

Many primary and some secondary studies addressing automated testing of web applications have been conducted. Among the secondary studies, Li et al. (2014) explained the main techniques found in the literature for testing web

applications, whereas Garousi et al. (2013) and Doğan et al. (2014) presented a systematic mapping and review on the theme, respectively. Hereafter we present only the automated, or semi-automated, approaches that act on the server-side of the application and that do not depend on software documentation.

Session-based testing (ELBAUM; KARRE; ROTHERMEL, 2003) uses the web server access logs to identify access sequences performed by one user, which are the sessions, and re-execute them to reproduce failures and perform regression testing. This approach does not deal with latent defects, acting only on defects that have already produced a failure.

Scanning and crawling (BAU et al., 2010) are techniques used to perform security testing of web applications. The scanners produce specific entries to exercise common vulnerabilities of web applications, such as SQL Injection (HARTLEY, 2012). The crawlers navigate through web pages finding points where the scanners will act to identify vulnerabilities. However, to use these techniques, the vulnerability must be previously known, as well as how to find it and exercise it.

Reverse engineering of interface specification with the server-side application (HALFOND; ORSO, 2007; HALFOND; ANAND; ORSO, 2009; SOHAN; ANSLOW; MAURER, 2015), also known as web APIs, may be followed by testing as a preventive maintenance approach for web applications. Some techniques use static analysis (HALFOND; ORSO, 2007) or symbolic execution (HALFOND; ANAND; ORSO, 2009) of the source code to recover the web API. Sohan et al. (SOHAN; ANSLOW; MAURER, 2015) used an HTTP proxy to collect examples of the web API usage and generate its specification based on them. Reverse engineering of web API depends on sophisticated tools (HALFOND; ORSO, 2007; HALFOND; ANAND; ORSO, 2009) and good examples (SOHAN; ANSLOW; MAURER, 2015) of application usage to achieve a reasonable level of precision. The imprecision of the generated interfaces combined with randomly generated testing may not be sufficient to identify latent defects with a low probability of occurrence (LI; DAS; DOWE, 2014). Additionally, in cases of dispersion of defects throughout the application, many interfaces would need testing, which would increase the cost of applying this technique.

There are some studies in the literature addressing defect prevention during development using static analysis (HECKMAN; WILLIAMS, 2011; MUSKE; SEREBRENIK, 2016). The software suites used to search for defects in source code using static analysis are called Linters (AYEWAH et al., 2008). Open source tools, such as SonarQube (SONARSOURCE, 2008), can identify defects in programs written in more than twenty programming languages. However, Linters check only defects associated with the inadequate use of a programming language or the use of error-prone constructions. Thus they do not find application-specific defects. Moreover, the level of precision of defect detection is a determining factor in the practical adoption of Linters (JOHNSON et al., 2013). Furthermore, fewer than 5% of the static analysis rules used in open source projects are custom rules (BELLER et al., 2016), i.e., application-specific rules. This low usage may indicate difficulty to work with custom rules to detect application-specific defects.

Ersoy and Sözer (ERSOY; SÖZER, 2016) presented an automated approach to detecting application-specific defects using traces of unhandled exceptions as input. In this procedure, four tools are used to automate the process: a log parser, a root cause analyzer, a checking rule generator, and a static analysis tool. The method was evaluated only by its recall, showing that it can find latent defects, but the method precision on this task was not informed.

In this thesis, different from that of Ersoy and Sözer (ERSOY; SÖZER, 2016), we propose a method that not only develops but also evaluates and improves custom static analysis rules using commonly available resources and tools. Our study specializes in the corrective and preventive maintenance of web applications. In contrast to other approaches to the same purpose and application domain (ELBAUM; KARRE; ROTHERMEL, 2003; HALFOND; ORSO, 2007; HALFOND; ANAND; ORSO, 2009; BAU et al., 2010; SOHAN; ANSLOW; MAURER, 2015), it focuses on unhandled latent exceptions in web applications, including specialized activities to detect and fix them.

2.5.Exception handling policies and its enforcement

Software developers often adopt an *ignore-for-now* approach when dealing with exception handling (SHAH; GÖRG; HARROLD, 2008). They neglect exception handling until there is an error or until they are forced to address it

(SHAH; GÖRG; HARROLD, 2010). This tendency is more common among novice developers than expert ones (SHAH; GÖRG; HARROLD, 2010). A way to force developers to deal with exception handling is to establish an exception handling policy and check it with some verification approach.

An exception handling policy of a software project is the set of design decisions that govern the use of its exceptions (BARBOSA et al., 2016). Most software projects currently do not even define an explicit exception handling policy (EBERT; CASTOR, 2013). In existing software with unhandled exceptions present in logs, these policies may not be known by maintainers.

Some studies define domain-specific languages (DSL) for exception handling policies, with related tools to enforcement (TERRA; VALENTE, 2009; GURGEL et al., 2014; BARBOSA et al., 2016). These DSLs focus on defining which modules have permission, obligation or prohibition to raise, handle, propagate, re-map, or re-throw certain types of exceptions (BARBOSA et al., 2016). However, they focus on exceptions raised by the application source code, lacking ways to deal with exceptions raised by third-party libraries. Furthermore, their implementation is limited to the Java programming language, which provides static typing and other mechanisms that facilitate locating which methods throw checked exceptions. It is not clear whether those solutions can be used with unchecked exceptions and scripting languages, which do not oblige the developer to declare the exceptions thrown by a method.

Finally, there are studies on assisting developer for repairing violations in exception handling (TERRA et al., 2015; BARBOSA; GARCIA, 2018). Those studies present recommendation systems that instruct developers on how to fix defects related to exception handling. However, it is not clear what the precision and recall of unhandled exception localization of these tools are, a task that should be performed before they can recommend a repair strategy for the defect. Additionally, they may need explicit exception handling policies defined to achieve a reasonable level of precision in the repairing recommendations (BARBOSA; GARCIA, 2018).

2.6.Concluding Remarks

Although reliability against unhandled exceptions in web applications has been subjected of investigations, the studies focus mainly on measuring reliability using server logs (KALLEPALLI; TIAN, 2001; TIAN et al., 2004; HUYNH; MILLER, 2005; GOŠEVA-POPSTOJANOVA et al., 2006; MA; TIAN, 2007; BANERJEE; SRIKANTH; CUKIC, 2010; JAFFAL; TIAN, 2014; ALANNSARY; TIAN, 2016;). Studies that consider practical conditions of maintenance, when the documentation is not reliable, propose automated solutions to deal with unhandled exceptions, but do not focus on locating the latent ones (ELBAUM; KARRE; ROTHERMEL, 2003; HALFOND; ORSO, 2007; HALFOND; ANAND; ORSO, 2009; BAU et al., 2010; SOHAN; ANSLOW; MAURER, 2015). On the other hand, linters can find a myriad of defects directly in the source code (HECKMAN; WILLIAMS, 2011; MUSKE; SEREBRENIK, 2016), but they do not find application-specific defects, such as unhandled latent exceptions.

Policies for handling exceptions can be defined and checked using DSLs, thus possibly locating unhandled latent exceptions. However, the proposed solutions (TERRA; VALENTE, 2009; GURGEL et al., 2014; BARBOSA et al., 2016) do not deal with unchecked exceptions or unhandled exceptions in scripting languages.

Within this scenario, novel approaches are needed for helping maintainers to automate the localization of unhandled latent exceptions in web applications. In the next section, we propose a new method called Pattern-Driven Maintenance (PDM).

3 The PDM Method

3.1.Introduction

In this section, we explain the proposed Pattern-Driven Maintenance (PDM) method and present a comprehensive example of its application. Figure 1 shows the activities collapsed into steps along the control flow of the method. Two primary paths can be observed: the maintenance path (steps 1, 2, and 3) and the defect pattern improvement cycle (steps 4, 5, and 2).

The maintenance path includes activities to process the server logs and identify defect patterns (step 1), to develop static analysis rules to detect the latent defects (step 2) and to verify the detected instances and correct the defects (step 3). The execution of the maintenance path occurs when the web server error logs contain new records. The web server logs must be monitored periodically to identify those new records by performing the first step of the method (failure analysis and defect pattern identification). Eventually no defect pattern will be identified in step 1, and in this case, no further step of PDM need to be performed. The maintainer should perform the typical corrective maintenance in cases when failures are present in logs, but no pattern were identified. For simplicity, we did not represent this case in PDM workflow (Figure 1).

The defect pattern improvement cycle is performed when the evaluation of the rules (step 4) (e.g., based on precision and recall) does not reach acceptable levels to alert during development. These levels vary according to the static analysis rule and depend on factors such as the impact on software reliability. Each company or maintenance team also has its own tolerance levels to false positive and negative alerts. Thus, we do not prescribe the thresholds for these levels. Further information on how we establish those levels in our industrial evaluations and benchmarks are provided in Section 3.5.

When precision or recall levels are not acceptable, the source code context of the detected defects is analyzed to improve the static analysis rules (step 5). Finally, there are two exit steps in the exit path of the method – rule deployment

for defect alerting (step 6) and rule contingency (step 7) –, which includes using the rules and patterns only in a limited way. Further details on the seven depicted steps are provided hereafter.

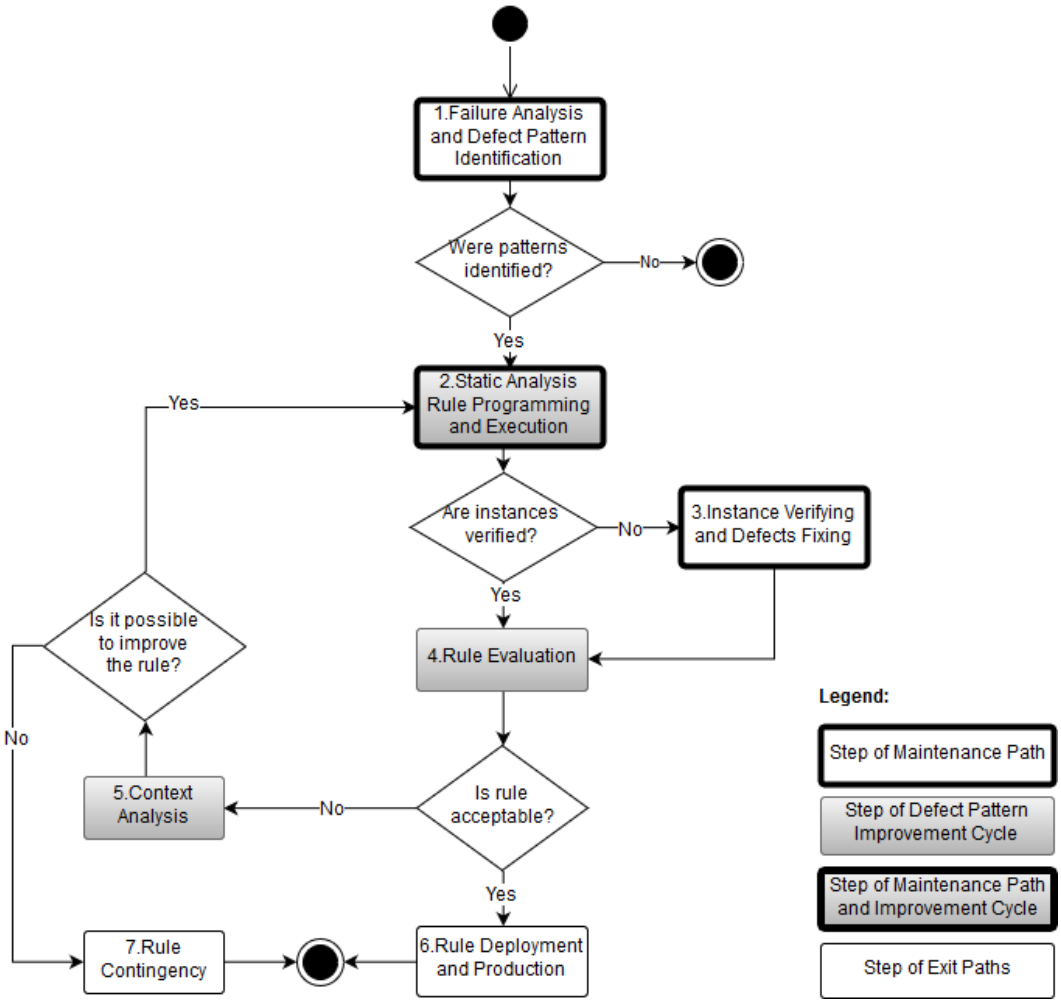

Figure 1: The PDM method control flow

3.2. Failure Analysis and Defect Pattern Identification

The first step of the PDM method is to perform failure analysis and defect pattern identification. The web server error log contains records of the failures generated by unhandled exceptions in web applications. Each record contains the exception type, the error message, and the source code file and line where the unhandled exception occurred. The log processing activity includes the extraction of these data from the logs and groups them by similarity. We define two failures as similar if their error types or messages are equal or vary only in parameter values.

Similar failures that occur in different parts of the source code suggest the existence of a defect pattern, typically introduced by systematic errors by the developers (KALINOWSKI; CARD; TRAVASSOS, 2012). Defect pattern identification consists of visually inspecting the source code context. During this step, the maintainer should focus on broadly identifying the defect pattern, trying to capture all instances of the related unhandled exception present in the application. This recommendation is due to the fact that latent defects are unknown and might slightly vary in the source code. The PDM method verifies and fixes only the defects identified by the pattern, thus it initially need a broader pattern for subsequently refining it. On the other hand, the pattern should not be defined too broadly, avoiding retrieving different kinds of defects. In this way, some experience is needed for defining the first version of the pattern. After finding a pattern, the maintainer must document it. Table 1 presents a template for defect pattern documentation.

Table 1: Defect pattern identification form

Field	Description
Defect Name	A descriptive name for the defect
Description	Situation that triggers the failure
Exception Type and Failure Message	What exception is thrown when the failure occurs, and which failure message is presented in the logs
Parameters in Failure Message	Variables that are present in the failure message
Example of Failure Message	An example of a failure message found in the log for this defect pattern
Class and Method of Throw	The class and method or group of them where the exception is thrown
Defect Characterization	Description in the natural language of the source code that leads to the defect
Defect Code Example	An example of the source code that leads to the defect.
Fixed Code Example	An example of source code that fixes the defect.

3.3.Static Analysis Rule Programming and Execution

After defect pattern identification, a static analysis rule is programmed and executed to locate the instances of the pattern. Some static analysis tools, such as SonarQube (SONARSOURCE, 2008), provide extension languages with which the maintainers can program their own rules to locate defects. Those languages use elements of the program represented into an abstract syntax tree (AST) and navigation operations to traverse the AST. Maintainers should develop a static analysis rule and test it using the defective and fixed code examples from the pattern documentation. The developed rule must alert the defective code, and not the fixed code to be accepted.

Thereafter, the static analysis tool can be used to execute the programmed rules locating the instances of the patterns, which are also called alerts or warnings. The located instances must include at least the defect that generated the failures present in the logs. The other instances are candidates for latent defects. The results of the static analysis, i.e., the source file and the line number of each alert, are stored in the static analysis tool database or exported to a file to support the other activities of the method.

3.4.Instance Verifying and Defect Fixing

In step 3, the maintainer verifies each instance of the pattern located and corrects the defects found. System level tests or source code inspection should be conducted for each alert instance with the intention to verify whether the target exception is thrown. If the maintainer chose to use testing for verification, test cases need to be defined for each instance of the pattern. The maintainer may choose to automate these test cases for further verification after defect fixing or perform them manually. It is noteworthy that, inherent to the testing activity, test case design to throw the unhandled exception may be complicated.

Furthermore, as we observed in our industrial evaluations (see Chapter 4), interrupting the verification step as soon as a false positive is found for performing a defect pattern improvement cycle may reduce the effort of performing PDM. This effort reduction occurs due to an improvement of the defect pattern. The static analysis rule may automatically discard several false positives that previously were matched by the pattern and will not be matched

after improvement. In this way, the discarded false positives do not need to be verified in step 3, reducing the effort of PDM application.

However, the preemptive execution of step 3 turns unfeasible the measurement of relative recall (see step 4, Section 3.5), since there will be no verified defect candidates. This way of performing PDM should be used in conditions where time for performing the method is more important than measurement of relative recall, as typically occurs in practical conditions.

3.5.Rule Evaluation

The pattern improvement cycle diverges from the maintenance path by the steps of rule evaluation (4) and context analysis (5). The maintainer uses the alerts, defects, and false positive alerts to evaluate the rule, for instance by calculating the precision and eventually relative recall of the static analysis rule (see Table 2). In the formulas of Table 2, *true defects alerted* refers to the number of true defects (i.e., true positives) alerted by a rule. It is noteworthy that the recall is relative to the defects matching the predetermined pattern, and its formula includes defects not alerted by the pattern (i.e., false negatives). A recall is recommended for evaluating the effectiveness of a pattern only for research purposes, since the effort to calculate it might involve inspecting the entire application source code. In PDM, to reduce this effort, we used a relative recall that approximates the expected recall. In a relative recall, false negatives are located by inspecting or testing the candidate defects retrieved by the initial and relaxed defect pattern version. The initial version should be broad enough to retrieve all defects but may include false positives. The purpose of the PDM learning cycle is to improve the precision without harming the relative recall.

Table 2: Metrics used in rule evaluation.

Precision	$\frac{True\ defects\ alerted}{True\ defects\ alerted + False\ defects\ alerted}$
Relative Recall	$\frac{True\ defects\ alerted}{True\ defects\ alerted + Defects\ not\ alerted\ (false\ negatives)}$

Calculating these metrics requires verifying all defect instances, which might require significant effort; thus, organizations may opt to conduct more informal rule evaluations, i.e., using only the precision. Nevertheless, precision and relative recall of the static analysis rules provide information about the quality of the developed rule. The practical use of a rule for defect detection during development depends on the acceptable values of these metrics.

A low precision value indicates that the rule will frequently alert the developers when there is no defect (false positive), which often induces developers to ignore warnings (JOHNSON et al., 2013). Low precision affects the confidence of developers in defect detection, thus causing them to eventually abandon this feature (JOHNSON et al., 2013). In contrast, low recall indicates that the rule will frequently miss defects (false negative), not alerting the developers about newly introduced defects. The threshold levels of precision and recall for accepting a rule for defect prevention depend on the tolerance to accept these situations. The definition of these thresholds may differ depending on characteristics of the company, application and the type and impact of the defect pattern.

In our industrial studies (see Chapter 4), we used a threshold level of 80% of precision and 100% of recall. These thresholds were established by asking the specialized opinion of each software project manager involved in the evaluations. A secondary study (HECKMAN; WILLIAMS, 2011) consolidated the precision of Linters. The precision reported in the studies vary from 3% to 98%, with a mean of 34% and median of 25%, showing the need for further research in this field. Additionally, less than 5% of the static analysis rules used in open source projects are custom rules (BELLER et al., 2016), i.e., application-specific rules. To the best of our knowledge, there is no benchmark published in academic literature for precision and recall of application-specific static analysis rules.

3.6.Context Analysis

Maintainers perform source code context analysis to improve the precision or relative recall when their thresholds for a rule are not acceptable. The alerts that do not reveal defects during verification, i.e., false positive alerts, provide information related to the context in which the static analysis rule fails (e.g., a

fixing alternative, which is a source code control flow that will not allow throwing the exception). For example, when a throw statement is executed inside of third-party library function, such as a string to integer conversion, it represents a misuse of the function. Although source code of third-party libraries may not be available, the failure produced in this situation is logged, allowing its identification. The control flow present in application source code may prevent in several ways that the instances of the same function call from being misused. The maintainer inspects the source code looking for the reason why the alert is a false positive, i.e., the control flow structures that prevent the exception from being thrown. The identified fixing alternatives are added to the defect pattern documentation.

After documenting the fixing alternative, the maintainer decides whether it is feasible[6] to automatically identify the fixing alternative, allowing such false positives to be ignored by making modifications on the static analysis rules. In the case of feasibility, the maintainer improves the rules (to consider such contexts) by performing the steps of programming and executing (2) and evaluating (4). Finally, improved rules can be accepted or rejected. If the rule gets accepted, the maintainer performs the step of rule deployment and production (step 6); otherwise, an improvement cycle can be performed to refine the rules. However, if there is no confidence that the rule can be improved, the maintainer should not deploy the rule. In this case, the maintainer performs the rule contingency step (7).

3.7.Deployment and Production of Rules

The rule deployment and production step (6) involves activities to make the Integrated Development Environment (IDE) present the alerts to the developers for defect prevention. Static analysis tools typically have configurations to choose which rules will present alerts to the developers. The rule deployment activity includes the setup of these configuration variables into the tool and the

[6] The evaluations of Chapter 4 showed that features (or their absence) of static analysis tools may make it unfeasible to improve a rule. In addition, Chapter 6 presents the main difficulties of maintainers in applying PDM steps, which can also hinder or make it unfeasible to improve a rule.

configuration of IDE plugins for presenting the alerts. After deployment, the static analysis tool, IDE, and plugin work together to present the alerts to the developers.

3.8.Contingency of Rules

The rule contingency step involves the development or improvement of an application programming guideline and the training of maintainers to use it. The documentation of rules that could not be automated by PDM presents information and examples about the defect patterns. The programming guidelines consolidate the defect patterns with explicit instructions on how to prevent them manually. The maintainers and developers should receive training on the defect patterns and on how to use the programming guideline.

3.9. Example of PDM application

In this subsection, we present a step by step example of the PDM method application in a comprehensive and straightforward case. We applied PDM in an open source JEE software named Employment and Internship Management System[7] (SisGEE). Students developed this software during an undergraduate course at CEFET/RJ, and CEFET/RJ employees use it. The selected example is the same one used during the observational study described in chapter 6.

The inputs for PDM method application are the logs of SisGEE (the excerpt of this log used for this example can be found in Appendix B) and a specific version of SisGEE[8] system that generated those logs. The first PDM step is failure analysis and defect pattern identification (see subsection 3.2). Performing failure analysis, we extracted data from the logs in order to compare the failures against each other and search for defect patterns. Table 3 presents data extracted from the logs. We can observe that exception type, and error message are equal for failures 1 and 3, as well as for failures 4 and 5. Similar failures should have the related source code inspected together for defect pattern identification. Figure 2 and Figure 3 show the source code that originated failure 1 and 3, respectively. We

[7] https://github.com/diogosmendonca/sisgee

[8] https://github.com/diogosmendonca/sisgee/tree/d06207f

can observe that both exceptions were thrown by the *Integer.parseInt* method. A simple solution to handle those failures is surrounding *Integer.parseInt* method call with a try/catch construct. In this way, we documented the pattern identified and the proposed solution using Table 1 form for supplying information for static analysis rule programming. Table 4 presents the documentation of the defect pattern identified, while Table 1 presents the explanation for each form field.

Table 3: Data extracted from logs during failure analysis

#Failure	File Name	Line	Exception Type	Error Message
1	BuscaTermoAditivoServlet	49	java.lang.NumberFormatException	For input string: ""
2	IncluirTermoEstagioServlet	60	java.lang.ClassCastException	java.lang.Double cannot be cast to java.lang.Float
3	VisualizarTermoEAditivo	43	java.lang.NumberFormatException	For input string: ""
4	RenovarConvenioServlet	44	java.lang.NullPointerException	
5	VisualizarTermoEAditivo	50	java.lang.NullPointerException	
6	index.jsp	4	org.apache.jasper.JasperException	File [import_head.jspf] not found

```
39 protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
40         Locale locale = ServletUtils.getLocale(request);
41         ResourceBundle messages = ResourceBundle.getBundle("Messages", locale);
42 
43         String msg = null;
44         String idAluno = request.getParameter("idAluno");
45          String mat = request.getParameter("matricula");
46 
47         Integer id = null;
48          System.out.println("Aqui >>> "+idAluno);
49         if(Integer.parseInt(idAluno)!=-1){
50         msg = ValidaUtils.validaObrigatorio("Aluno", idAluno);
51         if(msg.trim().isEmpty()) {
```

Figure 2: BuscaTermoAditivoServlet source code near line 49

```
32 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
33
34     String ide = req.getParameter("ide");
35     String ida = req.getParameter("ida");
36     String matricula = req.getParameter("matricula");
37     UF[] uf = UF.asList();
38     TermoEstagio termoEstagio=null;
39     TermoAditivo termoAditivo=null;
40
41     Aluno aluno=AlunoServices.buscarAlunoByMatricula(matricula);
42     if(ide!=null)
43     termoEstagio=TermoEstagioServices.buscarTermoEstagio(Integer.parseInt(ide));
44     if(ida!=null)
45     termoAditivo=TermoAditivoServices.buscarTermoAditivo(Integer.parseInt(ida));
```

Figure 3: VisualizarTermoEAditivo source code near line 43

Table 4: Documentation of the defect pattern identified

Defect Name	Unchecked Integer
Description	The application throws an exception when a string parameter is parsed to an Integer.
Exception Type and Failure Message	java.lang.NumberFormatException, For input string: "<value>"
Parameters in Failure Message	<value> - the value of the parameter passed
Example of Failure Message	java.lang.NumberFormatException, For input string: ""
Class and Method of Throw	Integer, method parseInt
Defect Characterization	A call to the parseInt method not surrounded by a try/catch

Defect Code Example	String intParam =
	request.getParameter("intParam");
	...
	//unchecked exception
	Integer intValue = Integer.parseInt(intParam);

Fixed Code Example	Integer intValue = null;
	try{
	intValue = Integer.parseInt(intParam);
	}catch(NumberFormatException e){
	//handle the exception
	}

The next step of PDM is static analysis rule programming. Using the documentation of the defect pattern, the maintainer uses some static analysis tool to implement a rule that locates unhandled latent exceptions that match the defect

pattern. In our example, we used SonarQube[9] for this task. SonarQube provides access to the abstract syntax tree (AST) of a Java program using the Visitor design pattern (GAMMA et al., 2002), allowing programming custom rules. Figure 4 presents a simplified version of the static analysis rule for the defect pattern documented in Table 4. This rule locates all *Integer.parseInt* function calls that are not surrounded by a try/catch. The method *visitMethodInvocation* is called for each method call in the software being analyzed. This method checks whether the name of the class is *Integer* and method name is *parseInt*. After locating one instance of the *parseInt* method, *isInsideTry* checks its parents recursively in search of a try block or a null value, returning if the method is inside a try block. For further information on how to program custom static analysis rules using SonarQube can be found in its documentation[10].

Table 5: Alerts produced by the first version of the static analysis rule with the results of their verification

#Alert	File	Line	Result of Verification
1	BuscaTermoAditivoServlet.java	49	Defect (True Positive)
2	BuscaTermoAditivoServlet.java	54	No Defect (False Positive)
3	FormTermoAditivoServlet.java	115	No Defect (False Positive)
4	FormTermoAditivoServlet.java	225	No Defect (False Positive)
5	FormTermoAditivoServlet.java	265	No Defect (False Positive)
6	VerTermoAditivoServlet.java	48	No Defect (False Positive)
7	VisualizarTermoEAditivo.java	43	Defect (True Positive)
8	VisualizarTermoEAditivo.java	45	Defect (True Positive)
9	FormTermoEstagioServlet.java	214	No Defect (False Positive)
10	FormTermoEstagioServlet.java	600	No Defect (False Positive)
11	FormTermoEstagioServlet.java	646	No Defect (False Positive)
12	FormTermoEstagioServlet.java	749	No Defect (False Positive)
13	FormTermoRescisaoServlet.java	81	No Defect (False Positive)

After programming a static analysis rule, maintainers should use it to locate unhandled latent exception candidates. Table 5 presents the alerts produced by running the static analysis rule together with the results of the verification of the alerts (achieved by inspecting the source code). Table 6 presents additional

[9] https://www.sonarqube.org/

[10] https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101

potential defect candidates, located using an IDE search, which was not alerted by the static analysis rule with the results of their verification. It is noteworthy that none of them concerned true defects (i.e., the rule indeed should not have alerted them). We calculated the precision and relative recall using the formulas described in Table 2, resulting in a precision of 23% and a relative recall of 100%.

```
@Override
public void visitMethodInvocation(MethodInvocationTree tree) {
    if(tree.methodSelect().firstToken().text().equals("Integer") &&
       tree.methodSelect().lastToken().text().equals("parseInt")){
        if(!isInsideTry(tree)){
            context.reportIssue(this, tree, "Surrond with try/catch");
        }
    }
    super.visitMethodInvocation(tree);
}
private boolean isInsideTry(Tree t){
    if (t == null){
        return false;
    }else if(t.is(Kind.TRY_STATEMENT)){
        return true;
    }else{
        return isInsideTry(t.parent());
    }
}
```

Figure 4: Example of static analysis rule implemented using SonarQube

Table 6: Defect candidates not alerted by the first version of the static analysis rule

#Alert	File	Line	Result of Verification
1	PrincipalTermo.java	378	No Defect (True Negative)
2	ValidaUtils.java	232	No Defect (True Negative)
3	ValidaUtils.java	233	No Defect (True Negative)

The 23% precision of the rule is unacceptable. Hence, we started a defect pattern improvement cycle by conducting context analysis. In the context analysis, we inspect false positive alerts searching for fixing alternatives different from the ones already included in the defect pattern. Table 7 presents a fixing alternative found during context analysis. After documenting the fixing alternative, we modified the static analysis rule to include it and executed the new version of the rule. The modified version of the rule eliminated all except for one false positive alert (Alert #12) of Table 5. In this way, the new precision of the rule is 75%, and

the relative recall remains 100%. Alert #12 has a different fixing alternative from the others, but the effort to include it in the defect pattern was not worthwhile to eliminate one single instance. The new rule version was accepted and deployed into a production environment for alerting developers.

Table 7: Documentation of a fixing alternative found during context analysis

Defect Pattern	**Unchecked Integer**
Context Name	Inside Integer Validation
Context Description	It is not possible to throw an exception in *Integer.parseInt* call when the call is inside an if block that checks its parameter for integer format
Context Cause	Control flow avoids exception throw
Context Characterization	An *Integer.parseInt* call inside an if block that uses the result of *ValidaUtils.validaInteger* in its expression and *ValidaUtils.validaInteger was* called with the same parameter of *Integer.parseInt*.

Code Example	campo = "Aluno";
	idAlunoMsg = ValidaUtils.validaInteger(campo, idAluno);
	if (idAlunoMsg.trim().isEmpty()) {
	  Integer idAlunoInt = Integer.parseInt(idAluno);
	  ...
	}

3.10. Concluding Remarks

Pattern-Driven Maintenance (PDM) and its learning cycle guide maintainers to produce static analysis rules that precisely locate unhandled latent exceptions in web applications. PDM indicates how to use server logs and application source code to identify similar failures and defect patterns. Those patterns are documented and programmed in some static analysis tool for locating unhandled latent exceptions. The located instances are verified for defect confirmation. The confirmed defects are fixed, and the developed rule is evaluated regarding its precision and its relative recall. If the rule does not achieve acceptable levels in those metrics, PDM indicates how to improve them by excluding fixing

alternatives observed in the false positives. After improving a rule, its precision and recall are re-evaluated. If it is needed and possible, new improvement cycles are performed until the precision and recall are acceptable. Afterward, the static analysis rule is deployed for alerting developers directly in their IDEs, preventing the reintroduction of the same defect pattern. Otherwise, if the rules are unacceptable and improving them is unfeasible, the prevention involves training maintainers to avoid the identified patterns of defects by using the defect pattern documentation.

PDM presents a novel approach to deal with unhandled latent exceptions in web applications. Software engineering solutions need to be evaluated to assess their benefits, risks, and conditions of application. In the next section, we present two industrial evaluations of PDM.

4 Industrial Evaluations

4.1.Introduction

Following the design science methodology (WIERINGA, 2014), after designing the solution concept, evaluations should be conducted. As described in the research design, we conducted two industrial evaluations, one under more controlled conditions, and a second one that further approximates practical conditions. In the first cycle, an initial version of PDM was designed and applied to a small size industrial web application, in controlled conditions (with the author of this thesis having complete access and previous knowledge about the application and the domain), with the purpose of an initial evaluation. In the second cycle, the PDM method was adjusted and applied to another small industrial web application using different technologies from the first one, with another industrial partner and without previous knowledge about the application and its domain, relaxing some controlled conditions and evaluating the sensitivity of the method. In this way, we aim to answer the following design science knowledge questions (*cf.* Section 1.2):

RQ1. (effect) What is the software reliability improvement achieved by fixing the located defects?

RQ2. (requirement satisfaction) What is the precision and recall of the automated defect localization?

RQ3. (sensitivity) Which factors have an impact on the method application and precision of the automated defect localization?

We use the metrics presented in Table 2 and the Nelson (1978) model (Formula 2.1) to answer these questions. The reliability is measured using the number of failures produced by unhandled exceptions during a period, in comparison with the number of accesses of the application in the same period. Sensitivity is discussed based on the experience of applying the method to two independent and different industrial applications.

4.2.First Evaluation

In our first evaluation of PDM, we applied the method to a financial web application marketed as a software-as-a-service (SaaS) in Brazil since 2010 (VITALJOB SOFTWARE, 2010). Although this software had been in use for eight years and a significant part of its defects had already been fixed, it was still being evolved, under active maintenance and eventually presenting failures. Some of these failures were caused by errors of use of the service; however, the application should not raise unhandled exceptions in those situations.

The server-side of the application had 12 KLOC developed in Python with the Django Framework running on Apache HTTP Server. The application source code, 31 days of the HTTP server access log, and the same period of the Django Framework error log were available. The author of this thesis applied the method. We consider this evaluation under more controlled conditions, given that the author previously worked with this industrial partner and application, so he had full knowledge of the system behavior and technologies involved. SonarQube was being used to control the code quality of the software under study; therefore, it was selected as the static analysis tool for the method application. For this task, we used its support for custom rules written in XPath. Data analysis was performed using R scripts, and the test automation tool used the Django Framework support for unit testing.

The log processing activity extracted 65 failures related to exceptions from 31 days of Django Error log. Table 8 shows these failures grouped according to the type of exception thrown. Some of the error log entries were incomplete because the TransactionManagementError does not provide information about the source code line in which the exceptions are thrown. The incomplete entries do not allow applying the defect pattern identification activity, although they are included in the number of known failures. We grouped all entries according to the type of exception, source file, and line, thus resulting in seven complete entries for defect pattern identification.

As shown in Table 8, exceptions of type DoesNotExist and ValueError were responsible for 44 out of 65 of the failures present in the logs (68%). Given that defect analysis activities should focus on the most frequent types (KALINOWSKI; CARD; TRAVASSOS, 2012), we investigated the source code

where these exceptions occurred for defect pattern identification and identified three defect patterns, presented in Table 12. Each defect pattern relates to a function call that may throw an exception. We programmed the static analysis rules to locate these three patterns. All rules followed the same principle: the instances that were not surrounded by a try/except can be a defect. In this way, all possible defect instances could be located. These rules were executed in SonarQube to have their instances located and were tested for defect confirmation.

After testing each candidate defect instance identified by the pattern, the rule could be evaluated. Table 9 presents the rule evaluation results. Although the rules were defined to initially reveal all possible related defect instances of the identified patterns (aiming an initial recall of 100%, cf. Chapter 3), the levels of rules precision of 49% to 67% were unacceptable for the company for defect alerting within the developers' IDE. Hence we started a defect pattern improvement cycle.

Table 8: Number of failures and defects by exception type.

Exception Type	Number of Failures	Number of Operational Defects
DoesNotExist	33	3
TransactionManagementError	17	-
ValueError	11	2
MultiValueDictKeyError	3	1
TypeError	1	1
Total	**65**	**7**

We conducted the defect pattern improvement cycle only for the rule Django ORM get. The other rules had few examples; hence, the effort to improve their precision might not justify the investment because of their low frequency of occurrence. The contexts identified were the origin of data in the variable passed as a parameter to the function, which could be from the request, database or a constant. We programmed the rules to identify these contexts in SonarQube. Table 10 presents the results of rule evaluation considering the identified context. It is possible to observe that, for one of the contexts (*Django ORM get - Parameter is from the request*) the relative recall decreased, failing to detect some of the defects. Therefore, this adjusted rule was discarded. Although the assessment showed rule precision improvements, the levels of this metric still did not reach

the threshold of 80% established by the company for rule precision. Hence, we performed only the rule contingency step after the improvement of the rules.

Table 9: Results of the evaluation of the developed rules.

Rule	Alerts			Precision
	Total	Defects	No Defects	
Django ORM get	109	53	56	49%
Float Conversion	15	10	7	59%
Date Conversion	6	4	2	67%

Table 10: Evaluation of the rules enhanced with the context.

Rule	Contexts from Alerts	Precision	Relative Recall
Django ORM get	The parameter is not constant (A)	52%	100%
	The parameter is not from the database (B)	63%	100%
	The parameter is from the request (C)	74%	58%
	A and B	68%	100%
	A and B or C	65%	100%

We analyzed the logs from one month before and two months after the defect fix deployment. Table 11 presents the results of this measurement. We can observe that the number of failures caused by the identified defect patterns were significantly reducted after defect fixing deployment, reaching zero failures in the second month after deployment. The failure identified after the first month of deployment was related to an incorrect fix of a previously identified defect. However, for the final version of the rules, the precision ranged from 59% (Table 9, Float Conversion) to 68% (Table 10, Django ORM get A and B); thus the rules were not accepted by the company for alerting developers.

Regarding the lessons learned from applying PDM in this first industrial evaluation, they are twofold. First, concerning the precision, the main reason for not being able to improve further it was that the abstractions needed to represent the source code context of the defect patterns were not present in the selected static analysis tool, namely, SonarQube with the XPath plugin. These contexts could be detected with higher precision if data flow analysis was available in this tool. Second, a significant effort was invested in testing false positive instances, calling for a faster way to evolve the rules to eliminate false positives as soon as possible.

Table 11: Measurements performed before and after the deployment of defect fixing during the PDM method application.

Metric	Data used to PDM application	Month Before Deployment	First Month After Deployment	Second Month After Deployment
Time Range (days)	31	31	30	30
Number of accesses	140,162	138,221	117,536	117,141
Number of unhandled exceptions failures	65	50	45	19
Reliability for unhandled exceptions	99.954%	99.964%	99.962%	99.984%
Number of failures caused by the identified defect patterns	44	12	1	0
Reliability for failures caused by the identified defect patterns	99.969%	99.991%	99.999%	100.000%

Table 12: Defect Patterns Identified in the First PDM Validation of first validation

Defect Pattern Name	Description	Defect Code Example	Fixed Code Example

Django ORM get	The application does not catch the exceptions thrown when a database search is conducted by id using Django ORM (Object-Relational-Mapper), and the id does not exist in the database.	`django.db.models import Model`
		`class Account(Model):`
		`  …`
		`...`
		`account =`
		`Account.objects.get (id=id)`
		`...`
			`try:`
			`  account =`
			`Account.objects.get(id=id)`
			`except:`
			`  #handle the exception`

Float Conversion	The application does not catch the exceptions thrown when a string is converted to float.	`a = "217x"`
		`b = float(a)`
			`try:`
			`  b = float(a)`
			`except:`
			`  #handle the exception`

Date Conversion	The application does not catch the exceptions thrown when it converts a string to date.	`from datetime import datetime`
		`a = datetime.strptime(\`
		`'10/10/201a','%d/%m /%Y')`
			`try:`
			`  a = datetime.strptime(\`
			`'10/10/201a', '%d/%m/%Y')`
			`except:`
			`  #handle the exception`

4.3. Second Evaluation

The second software selected to apply PDM was a small-sized administrative web application of an educational institution, written in PHP. Developers responsible for the software and working for this institution provided access to the system source code and logs. The application web server logs had evidence of failures caused by unhandled exceptions.

Building on the lessons learned from the first evaluation, in which we did not achieve the expected levels of rule precision, we used different technologies for developing the rules. This time we used programmed rules from SonarQube written in Java instead of XPath. We chose to keep SonarQube and use Java programmed rules because we already had experience with SonarQube and because Java written rules are more expressive than XPath ones, giving us a better chance of achieving higher levels of precision.

We also changed the way we executed the PDM method, slightly relaxing the conditions of experimentation and approximating it more to realistic conditions, as suggested by the design science methodology (WIERINGA, 2014). In our first evaluation, we were concerned with rigor in the method application and its evaluation. Hence, we evaluated the precision and recall of each rule and its versions (see Table 9 and Table 10), which required the significant testing effort of several false positive instances revealed by initial versions of the rules. In the second evaluation, we were concerned with applying PDM in a fast and practical way, thus approximating our evaluation to typical conditions of industrial practice. Therefore, we chose a faster approach for evolving the rules, producing a new version of a rule as soon as a false positive was found at the testing step (3), relaxing the rule evaluation (4) and conducting the context analysis step (5) to evolve the rule. The new version of the rule was then built to discard other false positives with the same context of the one that was found, avoiding the testing effort of false positive instances. Therefore, we calculated the precision during the evaluation step only after evolving the rule to remove a reasonable amount of false positives. Unfortunately, this approach does not allow calculating the relative recall, for which all possible defect instances would have to be tested for the initial version of the rule (to reveal the reference value for the defects matching the predertemined pattern).

We first analyzed the web server logs of the application. As the selected software has few accesses, we chose to analyze the maximum period of logs available. In total, 434 days of logs were available, with 68,972 and 642 entries in access and error logs, respectively. From these logs, we extracted the failures caused by unhandled exceptions in the application, which counted 151 failures (99.78% reliability). We analyzed the related operational defects in the source code and identified three defect patterns. Table 13 presents these defect patterns. As in the first PDM evaluation, the identified patterns are related to lack of data validation.

After identifying the defect patterns, we started testing their instances and improving the precision of the rules. Table 14 presents the results of rule development and improvement. The first search for defect instances returned a high number of possible defects since it did not include several existing structures for handling or preventing exceptions from being thrown. The first version of the rules also did not check situations where the origin of data made the checking unnecessary, such as date conversion when the data comes from the database, once the database provides dates in a fixed format. After including those and other contexts found in the rules, we achieved a final number of latent defects and false positives and used them together with the operational defects to calculate the final precision of rules.

As shown in Table 14, this time the achieved precision level of the rules (89.5-100%) was considered sufficient to be used for alerting developers during software maintenance and evolution. Hence the rules were successfully deployed into SonarQube to support defect prevention.

At the time the evaluation was conducted, only 30 days of logs after deploying the defect fixes were available. In these logs, there were 2,808 records of accesses and 19 records of failures. Those failures were caused by an error in database configuration, and there was no evidence in the error log of failures produced by unhandled exceptions related to the defect patterns.

Regarding the lessons learned, this PDM application allowed identifying two of them. First, concerning the difficulty on rules implementation, the abstractions available in Java written SonarQube rules are the ones present in AST structure. Those abstractions are made possible through a visitor design pattern (GAMMA et al., 2002). However, other concepts, such as data flow analysis,

were needed and we had to develop them also using a visitor pattern. This approach was challenging, showing the need for more powerfull tools for rules implementation. Second, we perceived similar defect patterns in both evaluations, i.e., related to data validation. This perception raises the hypothesis that these defect patterns could be generalized for other applications and that further investigation in this direction should be conducted.

Table 13: Defect Patterns Identified in the Second PDM evaluation

Defect Pattern Name	Description	Defect Code Example	Fixed Code Example
Date Conversion	A date conversion returns false when it fails. When a member function is called in a Boolean an exception is thrown.	`$dt1 = \DateTimeImmutable :: createFromFormat( 'd/m/Y', $str1); $dt1 = $dt1- >sub(new DateInterval('P1D' ));`	`… if(!$dt1){ … } $dt1 = $dt1->sub(new DateInterval('P1D'));`
Unchecked Integer	Data Access Object (DAO) layer may throw an exception when a non-validated integer variable is passed as parameter to their member functions.	`$res = $someDao- >someMethod($int_v ar);`	`if (strval($int_var) != strval(intval($int_var))) { … } $res = $someDao- >someMethod($int_var);`
Unchecked Id	Data Access Object (DAO) layer may throw an exception when a non-validated identifier variable is passed as parameter to their member functions.	`$res = $someDao- >someMethod($id_va r);`	`if (!isset($id_var) \|\| empty($id_var) || !is_numeric($id_var)){ … } $res = $someDao- >someMethod($id_var);`

Table 14: Failures and defects according to defect patterns of the second evaluation.

Defect Pattern	Failures	Operational Defects	First Search of Instances	Final Latent Defects	Final False Positives	Final Precision
Date Conversion	17	2	32	8	0	100.0%
Unchecked Integer	15	2	11	8	0	100.0%
Unchecked Id	74	3	172	14	2	89.5%
Other defects that do not form a pattern	45	5	N/A	N/A	N/A	N/A

4.4.Discussion

In this section, we answer our design science knowledge questions about PDM based on the experience and the findings of our industrial evaluations.

Q1. (effect) What is the software reliability improvement achieved by fixing the located defects?

In both evaluations, given the observed failures, we were able to eliminate all defects that could generate failures with similar causes, helping to improve the overall application reliability. The reliability concerning those unhandled exceptions improved by 0.031% (99.969% to 100%) for the first and 0.22% (99.78% to 100%) for the second application. The expectation is to prevent the recurrence of the failures produced by defect patterns in a rate of 37 and 10 failures per month, respectively, considering a similar monthly access profile of the applications.

Q2. (requirement satisfaction) What is the precision and recall of the automated defect localization?

Although PDM allowed successfully improving the precision of the static analysis rules, without harming the relative recall, the rules did not reach the desired precision levels of 80% established by the company in our first evaluation. We faced problems to program static analysis rules in SonarQube to represent the contexts in which the application must handle the exceptions. The identified contexts could have been better programmed using data flow analysis, identifying the origin or type of variables. These features were challenging but possible to program with the technology selected in the second evaluation. In this case, the precision of rules was higher than the ones in the first evaluation, achieving 89.5-100% of precision, thus being accepted by the company for defect prevention.

Q3. (sensitivity) Which factors have an impact on the method application and precision of the automated defect localization?

As noticed in our lessons learned, the way in which PDM steps are performed influences the application effort. Indeed, the PDM variation applied in the second evaluation, considering the context of false positives as soon as possible, showed to reduce the method application effort.

Another factor that may have an impact on effort is the familiarity of the maintainer with the subject web application. Without this familiarity, extra effort and support from other developers may be required in order to identify defect patterns, perform testing and evolve the rules.

Regarding the precision, our findings indicate that there is an influence of the technology selection on the precision of the rules. During our experience, using data flow analysis besides control flow analysis features helped to improve the precision of the rules in the second application.

4.5. Threats to Validity

In this chapter, we reported on applying an application-specific method in two different industrial contexts to identify, treat and prevent unhandled latent exception,s and improve application reliability. Thus, given the application-specific nature, no further theoretical generalizations or claims were made beyond

the findings of our specific evaluation scenarios. Nevertheless, we report some threats to validity that could have influenced our results.

Internal validity. One threat to internal validity is that maintenance and evolution of both industrial applications were not stopped while PDM was applied. Thus, the maintenance and evolution activity could have influenced the effect measured in the number of failures after PDM application. To mitigate this threat, we inspected, in both applications, all software changes made during the period while PDM was being applied. No software change had introduced or fixed defects related to the treated defect patterns, besides the ones made by applying PDM. We also performed cause analysis of each software failure included in both studies, splitting them in a group of those caused by the defect patterns and a group of those caused by other problems (see Table 11 and Table 14), isolating this confounding factor. We also did not evaluate the relative recall in the second application of PDM. Hence unknown false negatives might exist.

Construct validity. A threat to construct validity that we observed was that our second evaluation did not check all possible instances of defects with testing. As mitigation for this threat, we inspected the discarded instances of possible defects as soon as their context was included in the rules, preventing defects from having been discarded.

Conclusion validity. In our second evaluation, the time range of logs available after PDM application may not be sufficient to measure the effect on software failures. Furthermore, the values of precision and recall calculated for the static analysis rules depend on the state of the application to which the method was applied. Thus, changes in the software, after applying PDM, could introduce new contexts that are not handled by the rules, thus affecting the precision and recall. As the relative recall is only an approximation of the actual recall, false negatives might exist. We mitigated this threat by carefully defining broader patterns for evaluating the relative recall and hence approximate it to the actual recall.

External validity. As is tipical with empirical studies conducted in industry, the method application results are specific to the software applications and their characteristics and should not be generalized. Finally, a single person (the author of this thesis) was responsible for applying the PDM method steps (while discussing them and adjusting decisions jointly with other researchers). While he was familiar with the first software application and had a senior level of

experience in the related technologies, he had no previous contact with the second one and had less experience with its technology.

4.6. Concluding Remarks

We applied the PDM method to two industrial web applications from different companies and using different technologies. In both evaluations, applying the method enabled identifying three defect patterns and locating their latent instances statically (using SonarQube (SONARSOURCE, 2008)). A total of 104 defects were tested and fixed. In order to assess the PDM method, we performed measurements of failures caused by those patterns before and after applying PDM. In both applications, the failures caused by the treated defect patterns were eliminated, improving the application reliability. We also evaluated the static analysis rules produced by the PDM method. The method iteratively improved the precision of the defect pattern static analysis rules achieving absolute levels of precision of the rules of 59-68% and 89-100% in each application. These results strengthen our confidence that PDM can help maintainers in improving the reliability of existing web applications.

5 Reuse of Rules

5.1.Introduction

In order to assess the reusability of the static analysis rules produced by PDM, we applied them to similar projects and measured its precision in finding defects. As we observed in our previous PDM evaluations, the architecture of web applications plays an essential role in PDM rule definition. In this way, we expect that the rules produced by applying PDM in one software to be reusable in other software with similar architecture. Hence, we conducted a study that aims to answer the following additional design science knowledge questions (*cf.* Section 1.2):

RQ4. (sensitivity) In which scope rules created by applying PDM can be reused?

RQ5. (effect) What are the benefits of reusing rules created by applying PDM?

RQ6. (sensitivity) Which factors have an impact on reusing rules created by applying PDM?

Our first industrial evaluation of PDM produced rules for Python/Django written web applications, while the second produced rules for PHP ones. As Django is a popular framework that defines its reference architecture, we were able to find software projects with a similar architecture beyond the company frontier where the Python/Django rules were produced. Hence, we performed a cross-company evaluation of rules for these technologies. On the other hand, the architecture of the software used in our second PDM evaluation was defined by the company, and software with similar architecture was available for evaluation only in the same company. Therefore, we performed a within-company reusability evaluation for PHP rules. The next two subsections present the evaluations conducted on the reusability of the rules for these two cases.

5.2. Cross-company rules reuse evaluation

The software selection for the cross-company evaluation considered both, the use of similar technologies and the maturity of the project. We selected two software projects. The first was an information system for undergraduate student's performance monitoring named CADD[11] (Student Performance Evaluation Commissions Support System). The server-side of the application had 4.8 KLOC written in Python/Django. The CADD system was developed over a one-year period by two CEFET/RJ[12] undergraduate students as a final course project. The testing of the CADD system was adhoc, without using a systematic procedure. The system tests presented several defects, thus reflecting a low level of maturity.

The second project was an agile project management software named Taiga[13]. Taiga back-end[14] had 30 KLOC written in Python/Django within a history of four years of releases. This project is actively maintained and has several branches and stars on Github. Hence, we considered that Taiga had a higher level of software maturity than CADD system.

The evaluations performed on both software are presented and discussed in the next two subsections.

5.2.1. CADD system

As an evaluation procedure of the reuse of Python/Django rules, we executed them for the CADD system and tested the alerts produced for defect confirmation. We also searched for all instances of the function call present in each defect pattern as a way to confirm that the rules work correctly and to measure the capacity of effort reduction of rules reuse. Table 15 presents the results of a CADD system evaluation.

The CADD system did not have any float or strptime function calls, thus float conversion and date conversion patterns were not applicable. Regarding the Django ORM get pattern, we found 64 instances of the get function call. Some of

[11] https://github.com/diogosmendonca/CADD

[12] http://www.cefet-rj.br

[13] https://taiga.io/

[14] https://github.com/taigaio/taiga-back

the function calls not alerted by the defect pattern were inspected for confirming that the static analysis rule works correctly. After confirming it, the other instances were not inspected, since the original rule evaluation presented a relative recall of 100%. In the case of the CADD system, we reduced by 62.5% the effort of checking the proper exception handling in the get function calls by applying Django ORM get pattern. This result was achieved because 40 function calls out of 64 did not need to be verified.

Table 15: Evaluation of the Python/Django rules in CADD system.

Defect Pattern	Total of Function Calls	Function Calls not alerted	Function Calls alerted	Defects	Precision
Django ORM Get (A and B)	64	40	24	18	75%
Float Conversion	0	0	0	N/A	N/A
Date Conversion	0	0	0	N/A	N/A

The function calls alerted by Django ORM get pattern were tested for defect confirmation. The precision level of 75% found is slightly superior to the precision found in the software in which the rule was produced (68%). We inspected the false positives of the CADD system for causal analysis and found similar contexts causing the pattern to fail from the ones in the software which originated the rule. As stated before, this precision could be improved by using a tool with more resources for rule programming than SonarQube.

The level of precision found (75%) within the number of defects discovered in the software (18) strengthened our confidence that the rules produced by PDM may be reused in a cross-company setup to find defects in less mature software with a reduced effort. With the intent to help researchers and practitioners to understand better and check our results we made the artifacts used in our evaluation available on the internet[15].

[15] https://github.com/diogosmendonca/CADD/issues/1

5.2.2.Taiga

Taiga is a 30 KLOC, Python/Django written mature software with several installations and users. We chose Taiga to evaluate rule reuse in more mature software than the CADD system.

As the evaluation procedure, we executed the Python/Django rules in Taiga, excluding automated tests and migrations (database creation scripts) from the analysis. We also searched for the function calls present in the rules for checking if the rules were working correctly. Table 16 presents the number of function calls and alerts found. As our study with Taiga intended to understand the reuse of rules and not to fix defects, we chose to inspect the alerts produced by the rules instead of testing them. After inspecting some alerts without finding any defects, we found three new fixing alternatives that prevent the defects alerted by the Django ORM get pattern. Those contexts are explained in Table 17.

After finding these contexts, we realized that a new defect pattern improvement cycle would have to be performed to reuse Django ORM get rule in Taiga effectively. As some of these contexts could be very complicated or even impossible to include in the Django ORM get rule using SonarQube we decided not to perform the improvement cycle. Furthermore, the effort to continue inspecting Taiga without an expectation of executing an improvement cycle of the rule would not be worthwhile for the study purpose. Thus, we decided not to continue inspecting Taiga and finished the evaluation.

We conclude from the Taiga evaluation that the rules produced by PDM in one software may not be reusable in other software without adaptation, even when both software use the same framework or reference architecture. Programming style and architecture could be different from one software to the other, and the execution of a defect pattern improvement cycle may be needed. Furthermore, as also observed in the CADD system, the use of previously defined defect patterns may reduce the effort of checking a system for a specific defect. In case of Taiga, the Django ORM get rule execution enabled to reduce the inspection effort, discarding the need for inspecting 81 out of the 139 ORM get function calls, representing an effort reduction of 58% (naive estimate considering that all function calls whould require the same effort).

Table 16: Evaluation of the Python/Django rules in Taiga.

Defect Pattern	Number of Function Calls	Alerts produced by the Defect Pattern	Alerts inspected	Defects	Precision
Django ORM Get (A and B)	139	58	32	0	N/A
Float Conversion	0	0	0	0	N/A
Date Conversion	0	0	0	0	N/A

Table 17: New contexts found for Django ORM get rule in Taiga.

Context Description	Code Example
The use of pk to access an identifier attribute passed to get method instead of an id attribute	`User.objects.get(id=otherObject.pk)`

Id validation using a validator and inheritance. ProjectExistsValidator checks if the project exists and is called through inheritance on DueDatesCreationValidator is_valid method.	`class ProjectExistsValidator:`
	`  def validate_project_id(self, attrs, source):`
	`  …`
	
	`Class DueDatesCreationValidator(`
	`  ProjectExistsValidator,`
	`validators.Validator):`
	`  project_id = serializers.IntegerField()`
	`  …`
	
	`validator = validators.DueDatesCreationValidator(`
	`data=request.DATA, context=context)`
	
	`if not validator.is_valid():`
	`  return`
	`response.BadRequest(validator.errors)`
	
	`project_id = request.DATA.get('project_id')`
	`project = models.Project.objects.get(id=project_id)`

Constant as a literal or attribute.	`class BaseEventHook:`
	`  platform = "Unknown"`
	`  …`
	`  def get_user(self, user_id, platform):`
	`  …`
	`  user = get_user_model().objects.get(`
	`  is_system=True,`
	`  username__startswith=platform)`

5.3. Within-company rules reuse evaluation

The software selected for within-company rules reuse evaluation was a 1.7 KLOC, PHP written application with architecture similar to the software that

originated PHP rules. This application had been recently developed over a four-months period by three developers. Its purpose was to help the employees of the company to register themselves in the new corporate email system. Hereafter we refer to this application as a Registration system.

As an evaluation procedure, we started by executing the PHP rules without any modification in the Registration system. In the first execution, no defect candidate was found by the patterns. With the intent of verifying this result, with the aid of an IDE, we searched the source code for the elements contained in each defect pattern. We did not find any createFromFormat function call or integer variables being passed to DAO layer, which was the elements of the date conversion and the unchecked integer patterns, respectively. However, we found many id variables being passed to the DAO layer, which are the elements of the unchecked id pattern. The rule was not able to find that function calls because the naming convention for the DAO instance variables changed from the original system to Registration system. Thus the rule was adjusted to reflect the new naming convention and was again executed in the Registration system. Table 18 presents the results of the adjusted rule execution and inspection together with the total number of all function calls.

The adjusted unchecked id rule found five defect candidates in a total of 50 function calls, which represents a checking effort reduction of 90%. We inspected the alerts produced, and two of them were confirmed as defects, thus reflecting a precision of rule of 40%. Although this precision is low, the company decided to use the rules in its production environment for the Registration system. This decision was based on the excellent result in the first experiment with those rules, which strengthened the confidence of the company practitioners that the rules are useful for finding defects. Additionally, the low absolute number of false positives (three) associated with the functionality of SonarQube of marking false positive alerts not to be shown withing developer's IDE made the effect of false positives irrelevant for the developers.

We conclude from the Registration system evaluation that it is possible to reuse rules produced by PDM in a within-company environment. We also find that the adoption of the rules, in this case, was influenced by the previous experience of the company with the rules and that the precision may have had a low influence on this decision.

Table 18: Evaluation of the PHP rules the Registration system.

Defect Pattern	Number of Function Calls	Alerts produced by the Defect Pattern after adjustment	Defects	Precision
Unchecked Id	50	5	2	40%
Unchecked Integer	0	0	N/A	N/A
Date Conversion	0	0	N/A	N/A

5.4. Discussion

RQ4.(sensitivity) In which scope rules created by applying PDM can be reused?

Our evaluations indicate that the rules produced by PDM might be reused in other software in within-company and cross-company environments. Table 19 shows a summary of our quantitative results. The defects found showed the potential reuse effect of rules produced by PDM on the web applications reliability. This potential can be achieved not only in the maintenance and evolution phases but also in the software development phase. The CADD system was developed recently and, at the time of the writing of this document, it was not in production stage yet. The rules produced by the application of PDM on other software helped to identify several defects in the CADD system before it was released to its customers.

Table 19: Summary of PDM reuse of rules evaluation

Software / Metric	Type of Reuse	Technology	Defects found	Precision	Defect Candidate Reduction
CADD	Cross-company	Python/Django	18	75%	62.5%
Taiga	Cross-company	Python/Django	0	N/A	58%
Registration	Within-company	PHP	2	40%	90%

RQ5.(effect) What are the benefits of reusing rules created by applying PDM?

The beneficial effects of the reuse of rules produced by PDM is finding defects and reducing the number of defect candidates for verifying a defect pattern in other software. Table 19 presents our results. The precision of reused rules ranged from 40-75% without compromising the relative recall of 100%. The defect candidate reduction for verification ranged from 58-90% in our studies, which might represent a significant effort reduction in verifying the presence of a defect pattern in a software.

RQ6. (sensitivity) Which factors have an impact on reusing rules created by applying PDM?

We found some influence factors for rule reuse and adoption. First, architecture plays an essential role in rules definitions in PDM and consequently in its reuse. However, architecture similarity is not enough for rule reuse. As we observed in the Taiga and Registration systems, differences in the way that architecture is implemented and programming style might cause rules not to work correctly. Hence, the adjustment of the rules might be needed to enable effective reuse. An influence factor for reused rules adoption in a within-company environment is the success of rules in finding defects on other software. Indeed, the influence of this factor overcame the low precision of rules achieved in the Registration system, and the company chose to deploy the rule for defect prevention.

5.5.Threats to Validity

Internal Validity. The verifications of the results produced by the rules were conducted by a single researcher. However, the artifacts (except the proprietary ones of the Registration system) used in our evaluation are available online[12], allowing the investigations to be replicated by others to confirm the obtained results. Additionally, the partial inspection of defect candidates, with the purpose of verifying the correctly working of a rule, during its reuse might have caused missing false negatives. Inspecting all defect candidates discarded by a

rule is not a practical solution for confirming whether it is correctly working. One of the expected benefits of reuse of rules is reducing the effort of inspection, and by inspection all defect candidates this benefit would not be achieved.

Construct Validity. We selected the software systems for the study by convenience. For instance, the selection of software that was developed by students, who have a novice level of experience in software development, may have influenced the evaluation on the reusability of the rules. It is noteworthy that the defects detected by the patterns in our study are more commonly introduced by novice developers than by experienced ones.

Conclusion Validity. The number of systems chosen for evaluation does not allow applying any more sophisticated statistical techniques. Instead of claiming for conclusion validity we addressed the knowledge questions using a qualitative approach, trying to gather an initial understanding of the reuse scope, effects and factors.

External Validity. We recognize that the evaluations and results presented in this chapter are only examples of reuse of rules produced by PDM applications. The quantitative results achieved cannot be extrapolated to any other software than the ones in which the evaluation was performed. Thus, our findings should be interpreted as preliminary results from a specific context.

5.6.Concluding Remarks

We found that rules produced by applying PDM might be reused in within- or cross-company environments, and not only for software in the maintenance phase, but also recently developed ones. We were able to find defects in other software by reusing rules, as well as to reduce the verification effort of a defect pattern. Nevertheless, as expected, the architecture and programming style played an essential role in successfully reusing rules produced by PDM application, thus being an influencing factor for reuse. This finding indicates the feasibility of PDM producing rules that are application architecture and coding style specific, and not only application-specific. In this way, the reuse of rules has the advantage of producing more robust rules and might reduce the effort of identifying similar patterns in other systems. We also observed that previous successful experience with PDM influences rule reuse adoption.

Based on our experience, we recommend some practices for the evaluation and implementation of reuse of rules produced by PDM. After executing a rule in another software, our advice is to inspect both the alerts produced and the potential defect candidates that were not alerted. The inspection of the former might show new contexts to include in the rule to avoid false positives, and the latter might present adjustable cases where the rules fail because of differences in the architecture implementation or programming style. After inspecting these cases, fully or incrementally, the rules can be adjusted and executed for performing the maintenance cycle of PDM. Furthermore, additional defect pattern improvement cycles can also be performed if needed.

6 Evaluation of the Acceptance of PDM

6.1. Introduction

The studies presented in previous chapters showed that PDM is effective for preventing unhandled latent exceptions. However, in those studies, only the maintainer that created PDM (the author of this thesis) applied the method. Nevertheless, their feedback was positive about PDM experience, strengthening our confidence that PDM could help other maintainers. However, this feedback is insufficient to know whether they would accept and effectively apply PDM. Aligned with the methodology for introducing software processes described by Shull et al. (2001), because at this point we had determined the feasibility of PDM, our next step was to conduct an observational study.

In this way, our research objective in this chapter is to evaluate PDM concerning the effectiveness and acceptance from the viewpoint of different maintainers, answering the following additional design science knowledge questions (cf. Section 1.2):

RQ07. How effective are maintainers applying PDM for preventing defects?

a. How effective are maintainers in identifying and documenting defect patterns?
b. How effective are maintainers in programming a static analysis rule?
c. How effective are maintainers in identifying and documenting fixing alternatives present in false positives of a defect pattern?

RQ08. Would maintainers accept to use PDM?

d. How do maintainers perceive PDM regarding its ease of use?
e. How do maintainers perceive PDM regarding its usefulness?
f. Do maintainers intend to use PDM after experimenting it?

We conducted an observational study of PDM application using three groups of novice maintainers with different experiences and knowledge. The first group served as a pilot for instruments validation and was composed of computer science graduate students (n=9). The other two groups were composed of computer science undergraduate students. Students from group A (n=27) had no previous experience with the software under investigation and limited experience with the involved technologies (JEE), whereas students from group B (n=18) had previous experience with the software and were more familiar with its technologies. Each group was trained[16] and applied the two main reasoning steps involved in PDM, concerning identifying defect patterns from logs and adjusting static analysis rules to detect such patterns precisely. Group B had an additional session to implement the static analysis rules. We collected the results of applying those tasks and their feedback on the difficulties found. We also used the Technology Acceptance Model (TAM) (DAVIS, 1989) to assess the acceptance of PDM by maintainers in its three dimensions: ease of use, usefulness, and intention of use.

The remainder of this chapter is organized based on a guideline for reporting experiments (JEDLITSCHKA; CIOLKOWSKI; PFAHL, 2008), as follows. Section 6.2 presents our experiment planning. Section 6.3 details the observational study execution. Section 6.4 presents the study results. Section 6.5 discusses the results. Finally, we present the threats to validity in Section 6.6 and concluding remarks in Section 6.7.

6.2. Planning

6.2.1. Goals

The research objective covered in this study is to evaluate PDM concerning the acceptance and effectiveness from the viewpoint of different maintainers. In this way, following the GQM template (BASILI; CALDIERA; ROMBACH, 1994) we have the following goal:

[16] Our training materials are available in our replication package (http://doi.org/10.5281/zenodo.2597220)

Analyze PDM *for the purpose of* characterization *with respect to* effectiveness on conducting its steps, perceived usefulness, ease of use, and intention of use *from the point of view of* maintainers *in the context of* computer science students applying the PDM steps on excerpts of artifacts from a real and specific software product.

6.2.2. Participants

We selected the subjects of the study by convenience. We had access to graduate and undergraduate students in courses related to software quality of two different Brazilian universities. The first group of students was composed of nine graduate students in informatics from the Pontifical Catholic University of Rio de Janeiro. We called this group Pilot since its primary purpose was to help validate our materials. The other two groups called A (n=27) and B (n=18), were respectively composed of undergraduate students in computer science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and the Federal Center of Technological Education Celso Suckow da Fonseca (CEFET/RJ). Students of Group A were enrolled in the discipline of software testing and measurement, which is in the second year of their course, whereas students of Group B were enrolled in the discipline of software engineering, which is in the third year of their course.

One relevant difference between groups A and B was that group B had previous experience with the software on which they would apply PDM. The previous experience was possible because that software was used in the final course assignment in which group B students were enrolled. At the time when the students performed the tasks of the study, the assignment had already been passed to the students.

6.2.3. Experimental Materials

The characterization of students was made by filling a characterization form with questions about their experience (in months) with software development and maintenance in different contexts (for their own use, in a course, and in the industry). We also included questions about the level of experience with techniques and technologies that could influence the results of the experiments. In

case, we asked about their level of experience in Java, JEE, stack trace reading, static analysis rule programming, and source code inspections, as well as their proficiency in the English language. The consent and characterization forms used in the study are available in Appendix A.

The software for applying PDM was selected by convenience. The selected Internship and Employment Management System (SisGEE)[17] is an information system developed by students as an assignment of a web programming discipline of a computer science course at CEFET/RJ. SisGEE was developed using JEE technology and contained some defect patterns of unhandled latent exceptions in its source code.

We exercised some of those unhandled exceptions to produce a log for PDM application. The produced log is available in Appendix B, and the version of SisGEE that was used for producing this log is available on Github[18]. The log contains two exceptions produced for invalid conversion from string to an integer (NumberFormatException), two exceptions produced by access in service layer that returns null and the null value is used without previously checking (NullPointerException), and other two failures that do not form any pattern.

The first task of the study (Task 1) consists of executing the first PDM step, i.e., failure analysis and defect pattern identification. The failure identification consists of extracting failure data from logs filling a provided form. All groups of maintainers received the same form for failure identification. This form is available in Appendix C and D. The data that should be extracted consists of a file name and line where the exception was thrown, as well as the exception type and error message contained in the failure. After performing failure identification, maintainers were instructed to use the extracted data to compare failures and identify similar ones.

The maintainers were instructed to inspect the source code related to similar failures to identify patterns formed by the defects. If a defect pattern was identified, maintainers should document it. The Pilot group received a form with separate fields for information that would be useful for identifying a defect pattern whereas groups A and B received training in using a pattern language and should

[17] https://github.com/diogosmendonca/sisgee

[18] https://github.com/diogosmendonca/sisgee/tree/d06207f

document the defect patterns using that language. The form provided for the Pilot group is available in Appendix C. The fields are described in Table 1 while Table 4 presents an example of this form filled. The pattern language used by groups A and B consists of the same syntax of the software programming language (Java), but including wildcards symbols and conventions for documenting the pattern. Table 20 presents the wildcards and conventions available in the pattern language while Table 21 presents an example of defect pattern documented using this language. The forms provided for group A and B to fill during the first task of the study is available in Appendix D.

Table 20: Pattern language wildcards and conventions

Description	Wildcard Symbol	Example
An element must be present as it appears in the defect instances to describe the pattern	Use the same elements present in the examples of the defect, typically structural element.	If, for, while, switch, assignments, operators, etc.
An element must be present to describe the pattern, but the identifier name can vary in each instance of the defect. An abstraction of identifier name is needed.	Prefix name convention: any, some, other; followed by the name of the element needed.	anyVariable, otherVariable, someMethod, someClass
An element or group of elements do not need to be present to form a pattern, and it can be fully abstracted in the pattern description	The symbol "…" is used where any code can be present.	if(someVariable){ … }, someClass.someMethod(…)

Table 21: Example of defect pattern documented using the pattern language

Defect Example	Defect Pattern

`String param = request.getParameter("param");`
`…`
`if(param.length() > 0){ //line where the //exception was thrown`
`…`
`}`
	`String someVariable = request.getParameter("someParam");`
	`...`
	`someVariable.someMethod(...);`
	`...`

`String other = request.getParameter("otherParam");`
`…`
`String msg = "invalid value: " + other.trim(); //line where the exception //was thrown`
`…`	

After performing each task of the study, the maintainers were asked to fill a follow-up questionnaire with questions about their strategies and perceptions on the task. The questionnaire used for the first task of the study was equal for all groups of maintainers, and it is available in Appendix C and D. The questions asked concerned: the strategy used by the maintainer to identify the defect pattern, the perception if the time was enough to complete the task, the confidence in the patterns reported, the ease of performing the task, and the difficulties found.

Task 2 consisted of programming a static analysis rule. For this task, it one defect pattern documentation was provided, and the maintainers were asked to program a static analysis rule that locates the instances of this defect pattern. The provided defect pattern documentation was the one presented in Table 4. The tool selected for static analysis rule programming was SonarQube, which supports rule programming in Java programming language using the Abstract Syntax Tree (AST) of the Java language and the Visitor design pattern. A SonarQube template project of a custom static analysis rule was provided to facilitate the task. After finishing the task, the maintainers should provide the source code of the programmed static analysis rule and fill the follow-up questionnaire, which

follows the same template of task 1. The form used in task 2 is available in Appendix E.

Table 22: TAM questions used in the study

Dimension	ID	Question
Usefulness	Q1	Using *PDM* would improve my performance in preventing unhandled latent exceptions (i.e.., prevent faster)
	Q2	Using *PDM* would improve my productivity in preventing unhandled latent exceptions (i.e.., prevent more and faster)
	Q3	Using *PDM* would enhance my effectiveness in preventing unhandled latent exceptions (i.e., prevent more)
	Q4	I would find *PDM* useful in preventing unhandled latent exceptions
Ease of use	Q5	Learning to operate *PDM* would be easy for me
	Q6	I would find it easy to get *PDM* to prevent an unhandled exception
	Q7	It would be easy for me to become skillful in the use of *PDM*
	Q8	I would find *PDM* easy to use
Intention to use	Q9	I intend to use *PDM* regularly at work

Finally, Task 3 comprised the PDM steps of rule evaluation and context analysis. To perform this task, we provided the documentation of one defect pattern, the source code of the application that contains this defect pattern, and a list of source code lines in this application that were alerted by a static analysis rule that implements the defect pattern. Table 4 presents the provided defect pattern documentation. The application source code was the same one of other tasks, thus being available on Github. The alerted source code lines were provided in a form provided for maintainers performing the task, which is presented in Appendix F. During Task 3, maintainers should classify the alerts provided as defects or false positives. If a false positive was found, they should inform which fixing alternative was present in the source code. In the case of finding new fixing alternatives, maintainers should document them. The pilot group documented the fixing alternatives using a form while groups A and B used the pattern language. The form used by the Pilot group is presented in Appendix F while the one used by groups A and B is presented in Appendix G. After performing the task, the maintainers were asked to fill the follow-up questionnaire, which is similar to the

follow-up questionnaire of other tasks, and is included in both forms of Pilot and groups A and B.

At the end of the study, the maintainers were asked to fill the TAM questionnaire. This questionnaire is composed of nine questions split into three dimensions: usefulness, ease of use, and intention to use. The answers are provided in a five-point Likert scale ranging from strongly disagree to agree strongly. The questions of TAM questionnaire adjusted to our study are presented in Table 22, and the form used for asking them to maintainers can be found in Appendix H.

6.2.4. Tasks

The study started with the proper preparation of a laboratory with computers and Netbeans IDE for students to be able to perform the tasks. As soon as maintainers came to the laboratory, they received the consent term and the characterization form (see Appendix A). After filling these forms, an introductory presentation of 30 minutes about the PDM method was held, followed by a training of 20 minutes on Task 1 activities.

This training includes learning how to identify the data that should be extracted from the error logs, how to compare this data to identify similar failures, and how to compare similar failures in the source code to identify and document a defect pattern. The training of the Pilot group was different from the one of groups A and B because the forms used for documenting failures and defect patterns were different. After training, they received a brief explanation about Task 1 and the materials of this task were distributed, i.e., the forms of task 1 (see Appendix C and D) together with the logs (see Appendix B) and the application source code. Participants had 40 minutes to perform Task 1, which consisted of extracting data of six failures from logs and identifying and documenting two defect patterns found in the application source code. In the end, they filled the follow-up form and uploaded it to a folder or send it by e-mail together with the digital version of the task form.

After performing Task 1, the same groups of maintainers performed Task 3. We expected Task 2 to be more difficult and time-consuming than Task 3 and less relevant for observing the effectiveness of the maintainers on PDMs main

reasoning tasks. Therefore, we decided to change the order of the tasks. The Pilot group received a 10 minutes break between Task 1 and Task 3, group A performed Task 1 and Task 3 in two different days, finally group B did not receive any interval between the tasks.

We started Task 3 by distributing the forms of the task (see Appendix F and G) and the defect pattern documentation presented in Table 4. After that, we applied a training session of 20 minutes regarding Task 3. In this session, we showed how to identify false positives and how to document new defect fixing alternatives. Thereafter, the maintainers had 40 minutes to inspect 16 alerts of a defect pattern for classifying them into a defect or a false positive. This set of alerts contains 3 defects and 13 false positives that include two new fixing alternatives for the defect pattern. Finishing Task 3, maintainers filled the follow-up questionnaire and uploaded it to a folder or send it by e-mail together with the digital version of the task form. At the end of Task 3, we asked the maintainers to fill the TAM questionnaire (see Appendix H) and upload it to a folder or send it by e-mail to us.

Group B was the only one to apply Task 2 because only this group had time for one more task in their course. Task 2 was applied on a different day of the other two tasks. We started distributing the form of the task (see Appendix E), then we applied 20 minutes of training on Task 2 including concepts of AST and SonarQube technology for custom rules programming. After that, the maintainers had 50 minutes for performing the rule. Finally, they filled the follow-up form and uploaded it to a folder or send it by e-mail together with the digital version of the task form.

6.2.5. Questions and Variables

In this section, we describe our questions and variables. The first knowledge question we wanted to answer (RQ7) concerned the effectiveness of maintainers, i.e., for each task of the study we want to understand how effective maintainers are on performing the task. One indicator of the effectiveness of a maintainer is to complete a task and perform it correctly, i.e., completing the task with success. The percentage of maintainers that completed each task with success might give us insights about how easy it is for a maintainer to perform the task effectively.

Having this in mind, we aim at answering the following more detailed questions regarding the effectiveness:

RQ07. How effective are maintainers applying PDM for preventing defects?

a. How effective are maintainers identifying and documenting defect patterns?
b. How effective are maintainers programming a static analysis rule?
c. How effective are maintainers identifying and documenting fixing alternatives present in false positives of a defect pattern?

Some knowledge and experiences may influence the application of PDM. Hence, we were also interested in having insights on how these pieces of knowledges and experiences affect the effectiveness of maintainers in applying PDM. Therefore, in our study, we additionally investigated whether knowledge and experience in Java, JEE, static analysis programming, stack trace reading, and source code inspection have an influence on applying PDM, as well as, maintainers' previous experience with software development, software maintenance, and with the software that was used in the study.

The second knowledge question we wanted to answer (RQ8) concerned the acceptance of PDM by maintainers. Therefore, we used the TAM questionnaire (see Table 22) to evaluate the acceptance of PDM by the maintainers. Based on the TAM constructs, we answer the following questions:

RQ08. Would maintainers accept to use PDM?

a. How do maintainers perceive PDM regarding its ease of use?
b. How do maintainers perceive PDM regarding its usefulness?
c. Do maintainers intend to use PDM after experimenting it?

As TAM makes positive questions about the technology (see Table 22), we want to know the frequency in which maintainers agree with the questions. Additionally, we wanted to understand better the difficulties found by maintainers during PDM application. The frequency of specific difficulties found by maintainers might indicate their importance and improvement opportunities for the PDM method.

Table 23 and Table 24 describe the set of independent and dependent variables together with their types and scales.

Table 23: Independent variables

Type	Variables name and definition	Scale
Level of experience of maintainers	(L-Java) in Java (L-JEE) in JEE (L-STR) in stack trace reading (L-SCI) in source code inspection (L-SARP) in static analysis rule programming	1 = No experience 2 = I studied in a classroom or in a book 3 = I actively practiced in a classroom project 4 = I used it in a project in industry 5 = I used it in several projects in industry
Time of experience of maintainers	(T-SD-I) in software development in the industry (T-SM-I) in software maintenance in the industry	Years

Table 24: Dependent variables

Type	Variables name and definition	Scale
Percentage of maintainers	(P-CI-DP) that correctly identified all defect patterns (P-CD-DP) that correctly documented all defect patterns (P-Diff-DP) per reported difficulty found during defect pattern identification and documentation (P-CP-SARP) that correctly programmed the static analysis rule (P-Diff-SARP) per reported difficulty found during static analysis rule programming (P-CI-FA) that correctly identified all fixing alternatives (P-CD-FA) that correctly documented all fixing alternatives	Percentage (0% to 100%)

	(P-Diff-FA) per reported difficulty found during identification and documentation of fixing alternatives (P-A-Q1) that agree or strongly agree in Q1 of TAM (P-A-Q2) that agree or strongly agree in Q2 of TAM (P-A-Q3) that agree or strongly agree in Q3 of TAM (P-A-Q4) that agree or strongly agree in Q4 of TAM (P-A-Q5) that agree or strongly agree in Q5 of TAM (P-A-Q6) that agree or strongly agree in Q6 of TAM (P-A-Q7) that agree or strongly agree in Q7 of TAM (P-A-Q8) that agree or strongly agree in Q8 of TAM (P-A-Q9) that agree or strongly agree in Q9 of TAM	
Number of defect patterns	(N-CD-D) correctly documented by a maintainer	Integer (0, 1 or 2)

6.2.6. Experiment Design

The study performed is characterized as an observational study. We had one treatment (PDM) that was applied by three different groups on one object. This design served our purpose since we wanted to observe how effective they were in applying the treatment and their acceptance of PDM. We also wanted to have insights about the characteristics of maintainers that influence the application of each step of the PDM method.

6.3.Execution

The execution procedure followed the experiment planning tasks almost strictly. Hence, it is possible to understand our execution procedure by referring to Section 6.2.4. A difference between our planning and execution was the pilot group had been discarded after we have found problems with the initial materials. Details about this problem are presented in the next section. The analysis procedures are described hereafter in the analysis section together with their results.

6.4.Results

After executing the study, the materials were analyzed for determining the value of the variables to answer the stated questions. Some variables must be determined by a researcher, such as the number of correctly identified or documented defect patterns. In these cases, one researcher inspected the forms and determined whether the maintainers correctly identified or documented each item. The procedure of determining whether a maintainer correctly identified a defect pattern or a fixing alternative include reviewing all fields in the respective form and determining whether she captured the general idea of the pattern or fixing alternative. In another way, the procedure for deciding whether the documentation of a defect pattern or fixing alternative is correct, we searched for errors in the specific field of documentation.

Additionally, qualitative data was analyzed to determine the most frequent difficulties of the maintainers. The qualitative analysis included open coding of the qualitative data using the constant comparative method (SEAMAN, 1999) and counting the most common codes.

The remainder of this section is organized by the tasks executed during the study, reporting on the effectiveness, the profile of most effective maintainers and their difficulties. The last section shows the analysis of PDM acceptance, presenting the analysis regarding questions stated for the whole method and not its isolated tasks. The answers and discussion of the research questions based on the analysis results follow in Section 6.5.

6.4.1. Task 1 – Failure Analysis and Defect Pattern Identification

Table 25 and Table 26 present the percentage and number of maintainers that correctly identified and documented none, one, or two defect patterns during task 1, respectively. 48 maintainers completed task 1 by sending the task 1 form to the researchers. As task 1 had two defect patterns, the percentage of maintainers that were able to identify all defect patterns (P-CI-DP) ranged from 12% to 30% whereas the percentual of maintainers that correctly documented them (P-CD-DP) ranged from 0% to 30%.

Table 25: Percentage of maintainers that correctly identified defect patterns

Correctly Identified / Number of Defect Patterns	n	0	1	2
Pilot	9	44% (4)	44% (4)	12% (1)
Group A	23	52% (12)	18% (4)	30% (7)
Group B	16	38% (6)	50% (8)	12% (2)

Table 26: Percentage of maintainers that correctly documented defect patterns

Correctly Documented / Number of Defect Patterns	n	0	1	2
Pilot	9	89% (8)	11% (1)	0% (0)
Group A	23	61% (14)	9% (2)	30% (7)
Group B	16	44% (7)	50% (8)	6% (1)

Regarding materials validation, we used two formats of defect pattern documentation. The Pilot (n=9) group used the form for defect pattern documentation while groups A and B (n=39) used the pattern language. We found that individuals using a pattern language document more defect patterns correctly than the ones using a form (Wilcoxon-Mann-Whitney test on N-CD-DP, $W=120.5$, $p < 0.05$, one-sided). As we have found this difference, hereafter we discarded Pilot group from our analysis, thus considering only Groups A and B (n=39), which used a pattern language for documenting defect patterns.

The profile consists of the level and time of experience (see Table 23) of maintainers. With this respect, we have an interest in the profile of maintainers that correctly identified and documented all defect patterns in task 1 (n=8). Figure 5 shows, in its left side, a boxplot of this profile. We can observe that most of the maintainers had at least practice in a classroom in Java and stack trace reading while they had experience in the industry in source code inspection. They had less experience in JEE and static analysis rule programming, corresponding in most of the cases to theoretical knowledge in JEE and no knowledge at all in static analysis rule programming. The experience time in the industry of most of the

maintainers was about one year in software development and less than one year in software maintenance.

To understand differences in their profiles, we split the maintainers into two groups, the ones that correctly identified and documented all patterns (success, n=8) and the complementary group (other, n=31). For each knowledge or experience variable (L-Java, L-JEE, L-STR, L-SCI, L-SARP, T-SD-I, T-SM-I), we compared their distribution using Wilcoxon-Mann-Whitney test with alternative hypothesis of success group have higher values than others. The results of the tests are presented in Table 27. We found a significant difference in L-Java, L-STR, L-SCI variables ($p < 0.05$). Figure 5 presents the boxplot of variables distribution split into success and others group.

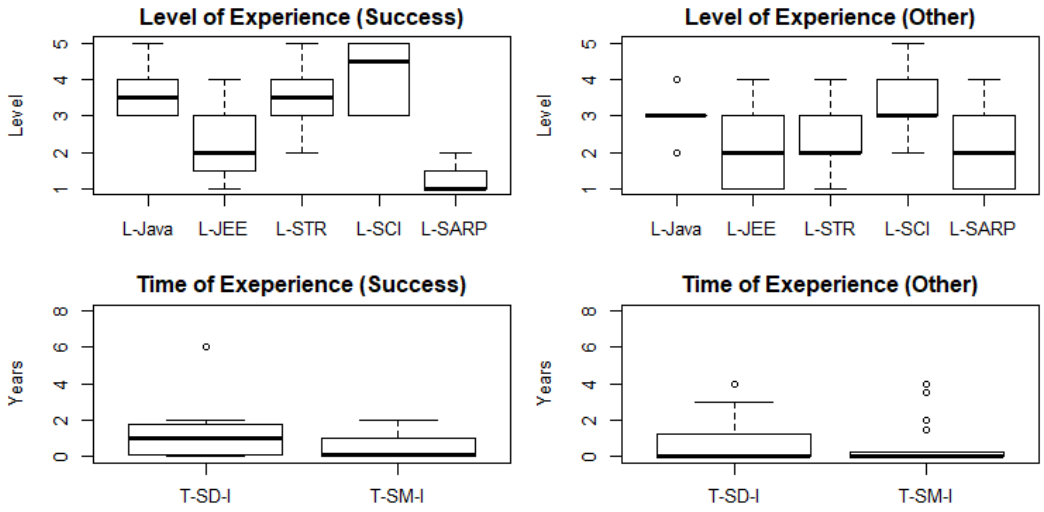

Figure 5: Boxplot of the profile of maintainers that correctly identified and documented all defect patterns in task 1 (Success) and complementary group of maintainers (other)

Besides analyzing the effectiveness and the profile of the effective maintainers, we also analyzed the maintainers' difficulties during task 1 based on the answers provided to an open question. Therefore, we open coded the qualitative data and counted the most common codes. Table 28 presents the results of this counting. As the groups of maintainers are different, we present the results separately per group. We can observe that the main difficulties of all groups involve somehow documenting and identifying defect patterns. Thereafter we also provide some examples of difficulties reported by the maintainers for Task 1.

Table 27: Results of Wilcoxon-Mann-Whitney tests between Success and Other groups in task 1 (n1=8, n2=31, one-tailed)

Variable	W	p
L-Java	169	0.026*
L-JEE	128	0.441
L-STR	202	0.002*
L-SCI	180	0.020*
L-SARP	82	0.946
T-SD-I	162	0.077
T-SM-I	138	0.284

Table 28: Most frequent difficulties of maintainers in Task 1

Group of Maintainers	P-Diff-DP	Difficulty description (code)
Group A	5 of 23 (22%)	Identifying the patterns
	5 of 23 (22%)	Documenting the patterns
	5 of 23 (22%)	Identifying a solution for the defect pattern
	4 of 23 (17%)	Lack of experience (Java, JEE and App Code)
Group B	10 of 16 (63%)	Documenting the patterns
	5 of 16 (31%)	Identifying the pattern

Examples of difficulties reported by group A and the related codes are provided hereafter:

"The main difficulty was confirming if the same exception types form a pattern in the source code." (Identifying the patterns)

"Representing the defect patterns in a generalized manner." (Documenting the patterns)

"To know the best way of fixing the code (with a try/catch or if/else)" (Identifying a solution for the defect pattern)

"No familiarity with the method, and the technology (servlets)" (Lack of experience)

Examples of difficulties reported by group B and the related codes follow:

"It was the first time I used this kind of form for documenting defect patterns. So, it took a while before it flows regularly" (Documenting the patterns)

"Recognizing patterns in different places of the source code and using the generic language to document the patterns." (Identifying the pattern and Documenting the patterns)

6.4.2. Task 2 - Static Analysis Rule Programming

Group B was the only one that performed task 2. This group had 18 maintainers, but only eight remained until the end of the task sending the materials for evaluation. No maintainer was able to program the proposed static analysis rule correctly. In case, the percentage of maintainers that correctly programmed the static analysis rule was 0%.

We also analyzed the maintainers' difficulties during task 2 based on the answers provided to a related open question. Therefore, we open coded the qualitative data and counted the most common codes. Table 29 presents the most common codes found. We can observe that the time to perform the task was the main difficulty reported followed by the concepts involved and difficulties on implementation. Thereafter we also provide some examples of the difficulties reported by the maintainers for task 2.

Table 29: Most frequent difficulties of maintainers on task2

P-Diff-SARP	Difficulty (code)
6 of 8, 75%	Time was not enough for performing the task
4 of 8, 50%	Understand the concepts
4 of 8, 50%	Difficulties on implementation
3 of 8, 38%	Lack of experience

Examples of difficulties reported by group B and the related codes are presented hereafter:

"The time was short for understanding the topic, and this topic is complicated…" (Time was not enough for performing the task)

"Understand the methods, classes and how to use them to solve the problem." (Understand the concepts)

"Unfamiliarity with the topic, difficulty to implement the methods, even after understanding I did not know how to program." (Difficulties on implementation)

"Low knowledge regarding static analysis causing serious difficulties for completing the task." (Lack of experience)

6.4.3. Task 3 - Rule Evaluation and Context Analysis

Table 30 and Table 31 present the percentage and number of maintainers who correctly identified and correctly documented zero, one, or two fixing alternatives during task 3, respectively. 28 maintainers completed task 3 by sending the form to the researchers. As task 3 had two fixing alternatives, the percentage of maintainers that were able to identify all fixing alternatives (P-CI-FA) ranged from 0% to 65% within the groups A and B whereas the ones that correctly documented all fixing alternatives (P-CD-FA) ranged from 0% to 37.5%.

Table 30: Percentage of maintainers that correctly identified fixing alternatives

Correctly Identified / Number of Fixing Alternatives	N	0	1	2
Group A	20	5% (1)	95% (19)	0% (0)
Group B	8	0% (0)	37% (3)	63% (5)

Table 31: Percentage of maintainers that correctly documented fixing alternatives

Correctly Documented / Number of Fixing Alternatives	N	0	1	2
Group A	20	65% (13)	35% (7)	0% (0)
Group B	8	25% (2)	37.5% (3)	37.5% (3)

The profile consists of the level and time of experience (see Table 23) of maintainers. In case, we have an interest in the profile of maintainers that correctly identified and documented all fixing alternatives in task 1 (n=3). Figure 6 shows, in its left side, a boxplot of this profile. We can observe that most of the maintainers had at least practice in a classroom in Java, JEE and source code inspection. They had less experience in stack trace reading and static analysis rule programming, corresponding in most of the cases a theoretical knowledge. All maintainers had no experience time in the industry. The three maintainers that correctly identified and documented all fixing alternatives are from group B.

Table 32: Results of Wilcoxon-Mann-Whitney tests between Success and Other groups in Task 3 (n1=3, n2=25, one-tailed)

Variable	W	p
L-Java	31.5	0.712
L-JEE	47.5	0.213
L-STR	50.5	0.682
L-SCI	36	0.547
L-SARP	23.5	0.865
T-SD-I	18	0.942
T-SM-I	24	0.887

We split maintainers into two groups, the ones that correctly identified and documented all fixing alternatives (success, n=3) and the complementary group (other, n=25). For each knowledge or experience variable (L-Java, L-JEE, L-STR, L-SCI, L-SARP, T-SD-I, T-SM-I), we compared their distribution using Wilcoxon-Mann-Whitney test with alternative hypothesis of success group have higher values than others. The results of the tests are presented in Table 32. We

did not find any significant difference between the two groups ($p < 0.05$). Figure 6 presents the boxplot of variables distribution split into success and others group.

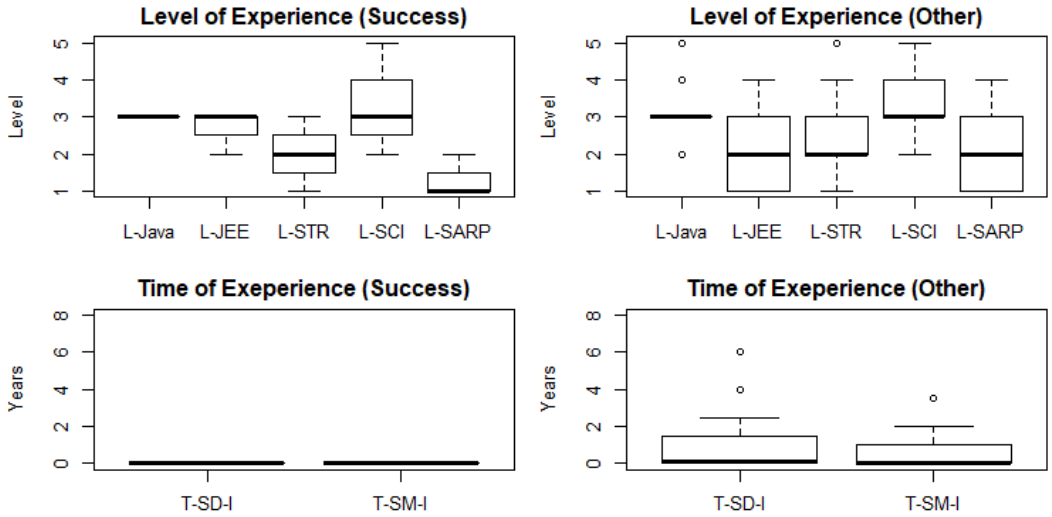

Figure 6: Boxplot of the profile of maintainers that correctly identified and documented all fixing alternatives in task 3 (Success) and the complementary group of maintainers (other)

As done for the other tasks, we also analyzed the maintainers' difficulties during task 3 based on the answers provided to a related open question. Therefore, we open coded the qualitative data and counted the most common codes. Table 33 presents the results of this counting. As the groups of maintainers are different, we present the results separately per groups. Thereafter, we also present some examples of the reported difficulties of maintainers.

Table 33: Most frequent difficulties of maintainers on task 3

Group of Maintainers	P-Diff-FA	Difficulty
Group A	6 of 20, 30%	Understand the app source code
Group B	3 of 8, 30%	Lack of experience with the task

An example of description of the difficulty reported by a participant of Group A:

"Sometimes the scope of the method can be really big, so inspecting every scope to check validation functions can become confusing." [19] (Understand the app source code)

An example of description of the difficulty reported by a participant of Group B:

"Lack of practice in activities like this." (Lack of experience with the task)

6.4.4.TAM – Technology Acceptance Model

Table 34: Percentage of maintainers that strongly agree or agree with TAM questions regarding PDM method.

TAM Dimension	Question	Percentage of Maintainers that Strongly Agree or Agree
Usefulness	Q1	77%
	Q2	74%
	Q3	81%
	Q4	70%
Ease of use	Q5	26%
	Q6	30%
	Q7	34%
	Q8	34%
Intention to use	Q9	15%

27 maintainers filled the TAM questionnaire and sent them to the researchers. Table 34 presents the percentage of the maintainers that agree or strongly agree with each question of TAM questionnaire (See Table 22). We can

[19] One point of inspection (PrincipalTermo.java line 378) is inside a long method (more than 400 lines of code). Although inspecting this method was a challenge for maintainers, a simple static analysis rule (see Figure 4) produced by PDM can correctly classify this line as having no defect.

observe that most the of maintainers found PDM useful, but not easy to use and that they do not intend to use PDM in their work. Further discussions of the results follow in the next section.

6.5.Discussion

In this section we answer and discuss our research questions, presenting the main related findings and insights.

RQ7. How effective are maintainers in applying PDM for preventing defects?

a. How effective are maintainers in identifying and documenting defect patterns?

Regarding the effectiveness of maintainers applying PDM, only 12% of the maintainers in group A and 30% in group B identified and documented all defect patterns. These results showed that after 20 minutes of training in task 1, few maintainers could effectively apply it. This fact could indicate that more training would be needed or that the application of task 1 of PDM needs to be facilitated in some way.

We collected and analyzed the difficulties found by maintainers during PDM application. During Task 1, the documentation format hindered the pilot group in documenting defect patterns. We could notice this problem by observing maintainer's comments on difficulties in this task and a statistically significant difference between the pilot group and the other groups. After changing the documentation format, groups A and B performed better than the pilot group in documenting defect patterns. However, groups A and B also complained about difficulties in identifying and documenting defect patterns, showing that this task could be indeed tricky.

The maintainers who correctly identified and documented defect patterns had superior experience in Java, stack trace reading and source code inspection. Thus, this type of knowledge possibly influences in performing task 1. The levels of experience of maintainers that successfully completed task 1 were in its majority industrial, while other maintainers had only academic experience. This

fact might indicate that an appropriate profile for performing task 1 is a maintainer with industrial experience in defect fixing using the technology of the software in which PDM will be applied. Defect fixing activity typically involves stack trace reading and source code inspection. It is unclear if the industrial expertise in those aspects might be replaced by better training of maintainers for performing task 1. Hence, further experiments are needed for investigating this hypothesis.

b. *How effective are maintainers in programming a static analysis rule?*

In task 2, no maintainer was able to program the proposed static analysis rule correctly. The main complaint of maintainers on this task was the limited time for programming (50 min), the difficulty to comprehend the concepts involved (Abstract Syntax Tree and Visitor Pattern), and difficulties for programming the rule using those concepts and SonarQube. Task 2 was the most challenging task to be performed by the maintainers selected for the study. This fact might indicate that programming static analysis rules are a particular task and that it might be difficult to find a professional that is already skillful on it. In the case of training professionals in static analysis rule programming, a single training session with 20 minutes of presentation and 50 minutes of exercises will not suffice.

c. *How effective are maintainers in identifying and documenting fixing alternatives present in false positives of a defect pattern?*

As well as in task 1, few maintainers were able to identify and document fixing alternatives during task 3 correctly. No maintainer of group A and 63% of the maintainers of group B correctly identified all fixing alternatives while no maintainer of group A and 37.5% of group B correctly documented them. The maintainers that correctly identified and documented all fixing alternatives had no significant difference in any knowledge and experience variables to other maintainers. As stated before, group B had previous experience with the software to which PDM was applied by using it in the course assignment. This fact might

indicate that previous experience with the software being maintained positively affects the performance of identifying fixing alternatives.

The main complaints of maintainers regarding task 3 for group A were related to understanding the application source code, and for group B to the lack of experience on the task. The group A complaint reinforces the insight of experience with software affects the performance in task 3. When the maintainer has this experience, as in the case of group B, the main complaint was not the software but the experience with the task.

RQ8. Would maintainers accept to use PDM?

a. How do maintainers perceive PDM regarding its ease of use?

b. How do maintainers perceive PDM regarding its usefulness?

c. Do maintainers intend to use PDM after experimenting it?

The TAM questionnaire was used to access the PDM acceptance by maintainers. Table 34 showed that most maintainers perceive PDM as useful, but not easy to use. This perception may have influence in the intention to use PDM since few maintainers answered agreeing with this question in the TAM questionnaire. These perceptions show that PDM application should be facilitated for improving its acceptance by maintainers.

6.6.Threats to Validity

In this section, we discuss the threats to the validity of the study in the four types described by Wohlin et al. (2012), i.e., internal, external, construct and conclusion.

Internal Validity. During the study, some maintainers did not perform all tasks. As an uncontrollable condition of mortality, some maintainers left the experimentation session before completing task 3, especially in group B. This group had classes at night, and as the end of the class come close some of them naturally left the class. This condition made the number of subjects vary significantly from task 1 (n=39) to task 3 (n=28). Hence, the results of task 3 might have been affected by this variation. Furthermore, only one researcher

performed the qualitative analysis, which could be influenced by the researcher point of view.

Construct Validity. The task selection, order of application, and time to complete the tasks could have affected the results of the study. We have selected to perform task 1 and task 3 in all groups, while we applied task 2 only in group B. This task to group assignment was made considering the constrained time in the classes of the groups. Additionally, group B received the tasks in a different order in which they were designed in PDM, receiving task 1, task 3 and thereafter task 2. As PDM steps interact, the changing in order or avoiding steps might affect the effectiveness in applying PDM and the perception of maintainers about the method. Furthermore, the time of training and for performing each task were also constrained by the time available in classes of the groups and might have affected the results of the study.

Conclusion Validity. Our purpose was to conduct an observational study to evaluate whether other maintainers would be able to apply the PDM steps. Given our limited sample size, we had no further aims regarding conclusion validity. Indeed, while some maintainers successfully completed the steps, their number was not enough to make claims about their characteristics. It is noteworthy that, although we used a large group of students in our research (n=54), they were split into three different groups and only a few of them correctly completed each task. Hence, while we analyzed the characteristics of those groups against their complementary groups, the confidence in the results is affected by our sample size.

External Validity. Our observational study considered a specific setting (e.g., software, technology, students). Hence external validity is limited. Furthermore, as is common with empirical studies conducted with students, the results concern novice maintainers and their characteristics, not being generalizable.

6.7. Concluding Remarks

We have evaluated PDM regarding maintainers' effectiveness in applying it and their acceptance of the method. In this way, we observed 54 novice maintainers applying PDM steps split into three tasks, i.e., failure analysis and

defect pattern identification (task 1), static analysis rule programming (task 2), and rule evaluation and context analysis (task 3). The maintainers had difficulties during PDM steps application, and few of them correctly completed the tasks. The difficulties found included the defect pattern documentation format, which was changed during the study, the identification of defect patterns and their fixing alternatives, the static analysis rule programming, as well as the understanding of subject software source code and difficulties caused by lack of experience with the tasks.

We analyzed the profile of maintainers that correctly completed the tasks. We found that the ones that correctly completed task 1 had superior experience in the subject software programming language (Java), stack trace reading, and source code inspection; while in task 3 the maintainers had previous experience with the software to which PDM was applied. No maintainer correctly completed task 2.

Finally, the maintainers answered a TAM questionnaire about PDM acceptance. Most of them found PDM useful but not easy to apply and do not intend to use PDM at work. However, the perceived ease of use of PDM could be hindered by the conditions of the limited time of an observational study, thus affecting the intention of use.

In this way, we had insights about the effectiveness of maintainers applying PDM and their acceptance. We also identified factors of influence that can help to identify appropriate professionals for applying each step of the PDM method. However, the results also indicate that proper training is needed for applying the method, especially on static analysis programming.

As future work, we intend to reproduce this study with more experienced maintainers and with more time of training in PDM. We also plan to develop support tools to facilitate a PDM application.

7 Conclusion

Failures generated by unhandled exceptions affect the reliability, usability, and security of web applications. Several studies showed the presence of unhandled exceptions in web applications (KALLEPALLI; TIAN, 2001; HUYNH; MILLER, 2005; GOŠEVA-POPSTOJANOVA et al., 2006; JAFFAL; TIAN, 2014;). There are some solutions for enforcing policies to handle exceptions (BARBOSA et al., 2016; BARBOSA; GARCIA, 2018). However, those solutions are limited, and fall short of dealing with unhandled exceptions generated by third-party libraries and unchecked exceptions. Additionally, it is not clear whether those solutions could be used with scripting languages, such as PHP and Python, which are commonly used in web applications.

Unhandled exceptions might be latent in the source code, thus not presenting failures in logs. Those unhandled exceptions need to be located for proper fixing. However, it is possible to avoid exceptions to be thrown in several ways, and each application has its approaches to deal with exceptions. This specificity hinders general linters to automatically locating unhandled latent exceptions. Furthermore, automated solutions for testing web applications do not focus on unhandled latent exceptions (GAROUSI et al., 2013; DOGAN; BETIN-CAN; GAROUSI, 2014;).

In this thesis, we proposed PDM, a method that iteratively uses static and dynamic analysis to find, correct, and prevents unhandled exceptions in web applications. PDM help maintainers to automate the unhandled exception localization by guiding them to find defect patterns and programming static analysis rules that locate those patterns. We successfully applied PDM in two industrial cases, reused the rules generated in other software within- and cross-company and evaluated PDM acceptance by maintainers.

7.1. Revisiting the Thesis Contributions

Initially, we aimed to support industrial partners in solving a recurrent problem of unhandled exceptions in a financial web application implemented in Python. As the application has no reliable documentation, no automated testing, and high people turnover, we realized that there was no appropriate approach to deal with unhandled exceptions in this context. An inspection could be performed, but the entire software would need to be inspected, which would represent a large effort. A software process improvement approach could be implemented, solving the problem during evolution, but not dealing with unhandled latent exceptions. A cost-effective solution was needed to locate and fix the unhandled latent exceptions and to avoid the reintroduction of the problem.

We noticed that failures in this software were similar and could represent the same error repeated several times, thus forming defect patterns. However, a systematic approach was needed to identify, document and locate those patterns, finding not only the unhandled latent exceptions but also informing maintainers when the same defect pattern has been reintroduced. Within this context, we proposed Pattern-Driven Maintenance (PDM), a systematic method to help maintainers dealing with defect patterns using automation.

We applied PDM in two industrial software systems, showing its effectiveness (RQ1), the precision and recall of automation produced (RQ2), and the influence factors (RQ3) for applicability not only for the software for which PDM was initially designed but also for other software. In this way, we state our first and foremost contribution:

> ***1st Contribution.*** *An empirically evaluated method for preventing unhandled latent exception in web applications.*

After investigating the effectiveness of PDM, a hypothesis on the reusability of the defect patterns found during the study raised. For checking this hypothesis, we selected some software systems with the similar architecture of the ones in which the patterns were found and checked patterns reusability. Some of the

defect patterns could be successfully reused. We evaluated the reusability in within- and cross-company environments (RQ4), showing that it is possible to reuse PDM produced defect patterns and static analysis rules. We investigated the benefits of reusing rules (RQ5) as well as factors of influence for reusing rules (RQ6). Table 35 presents the patterns identified during the thesis together with the precision and recall of the static analysis rules produced.

Table 35: Defect patterns found during the thesis with the evaluation of static analysis rules produced

Defect Pattern	Technology	Software	Precision	Relative Recall
Django ORM get	Python/Django	inFinance	68%	100%
		CADDs System	75%	N/A
Date conversion	Python/Django	inFinance	59%	100%
Float conversion	Python/Django	inFinance	67%	100%
Unchecked Id	PHP	SAD System	89.5%	N/A
		Register System	40%	N/A
Unchecked Integer	PHP	SAD System	100%	N/A
Date conversion	PHP	SAD System	100%	N/A
Integer conversion	Java	SisGEE System	75%	100%

The reuse of defect patterns and static analysis rules produced by PDM might not be immediate. The patterns and rules could need to be adjusted, and we present some recommendations on how to act to proper reuse the defect patterns produced. These recommendations involve how to check whether the static analysis rule reused is correctly working and how to adjust them to new software. Hence, we state our second contribution:

2nd Contribution. *Guidance on reusing rules produced by PDM.*

Table 36: Papers produced among the thesis

Paper	Chapter	Status
MENDONÇA, D. S.; Staa A.v. . Um Método Semi-Automatizado para Manutenção Corretiva e Preventiva de Sistemas Web. In: **XVI Simpósio Brasileiro de Qualidade de Software (SBQS)**, 2017, Rio de Janeiro. XV Workshop de Teses e Dissertações em Qualidade de Software, 2017. p. 80-88.	3,4	Published
MENDONÇA, Diogo S. et al. Applying pattern-driven maintenance: a method to prevent latent unhandled exceptions in web applications. In: **Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'18)**. ACM, 2018. p. 31.	3, 4	Published
MENDONÇA, Diogo S.; VON STAA, Arndt; KALINOWSKI, Marco. Pattern-driven maintenance: a method to prevent unhandled latent exceptions in web applications. **Journal of Systems and Software (JSS)**. Elsevier, 2019.	3,4,5,6	Submitted

After checking the reuse of defect patterns produced by PDM, we still had doubts if maintainers would effectively apply (RQ7) and accept (RQ8) PDM. This doubt was justified because only the maintainer that created the method (the author of this thesis) have applied PDM and he have a senior level of experience. Hence, we decided to evaluate PDM with novice maintainers. Our findings showed characteristics of maintainers that successfully applied each step of PDM, thus reflecting the knowledge and experiences needed. This finding could help to proper select or training maintainers for applying the method. The evaluation of acceptance showed that most of the maintainers found PDM useful, but not easy to apply. One hypothesis for justifying this finding is that most of them did not have the knowledge and experiences needed to apply the method effectively. This hypothesis needs further investigation and motivates our future work. In this way, we state our third contribution:

***3rd* Contribution.** *Guidance for effectively selecting or training maintainers for applying PDM.*

Table 36 presents the papers produced throughout the thesis. The studies presented in chapters 5 and 6 were conducted in the second semester of 2018, and for this reason, are not yet published.

7.2. Limitations

There are some limitations to the PDM method. First, the pattern identification depends on the ability of the maintainer in comprehending source code, comparing the multiple defects that produce the same failure and abstracting its common parts. As we presented in chapter 6, the minority of novice maintainers could correctly identify defect patterns. This limitation can be mitigated by proper training or selection of maintainers that will apply the PDM method. Another way to mitigate this limitation is to have support tools to facilitate maintainers to identify and document defect patterns.

The precision of automation depends on the static analysis tools selected for applying PDM. Some defect patterns had abstractions that could not be fully implemented in the selected static analysis tool, in our case SonarQube. Hence, the tool selection should consider the abstraction needed to implement rules. As we observed in chapter 4, the data flow and control flow analysis are needed for adequately implementing common rules to locate unhandled latent exceptions.

The PDM method starts to detect defect patterns by using application server production logs. Hence, defects related to patterns that were never exercised and did not produce a failure cannot be detected by the PDM method. Testing logs could be used instead of production logs to mitigate this problem; however, proper test cases need to be designed for the log to be representative.

Finally, the studies of the effectiveness of maintainers on applying PDM and their acceptance were conducted with novices. In this case, the external validity of the study is limited, needing replication of the study with more experienced maintainers for mitigating this limitation.

7.3. Future Work

As future work, we intend to evaluate PDM acceptance by experienced maintainers. We do believe that experienced maintainers could have different results on applying PDM than novice ones. Their feedback can help to improve the PDM method further and to evaluate PDM acceptance better.

We also intend to develop tools to facilitate a PDM application. The difficulties collected during PDM acceptance study showed that novice maintainers have difficulties in identifying and documenting defect patterns. They also have problems in implementing static analysis rules. Tools support on these activities could help novice maintainers in effectively applying PDM.

The PDM method should be modified to be used to locate other kinds of defects. Some necessary conditions to use PDM are the defects form patterns and must exist a way to initially identify those patterns. An example of a possible application of PDM is for dealing with architectural violations. Those violations may form anti patterns, and an initial inspection of the software would be used to identify them. As future work, we intend to experiment PDM with other kinds of defects than unhandled exceptions.

The PDM method can also be adjusted to work not only in the back-end of web applications but also in front-end. Web browser logs would be used to perform PDM in the front-end of web applications.

The PDM method can be more automated. In this thesis, we automated failure analysis, but there are possibilities of automating other steps of the method, such as patterns identification, static analysis rule programming, context analysis.

Finally, we proposed in this thesis a pattern language for documenting defect patterns. The pattern language proposed is tight to the programming language where the defect patterns occur. The specification of a programming-language-agnostic pattern language would allow defining defect patterns that matches defects in more than one programming language.

8 References

ALANNSARY, M. O. **Quality improvement of SaaS (Software as a Service) in the Cloud**. [s.l.] Southern Methodist University, 2016.

ALANNSARY, M. O.; TIAN, J. **Measurement and Prediction of SaaS Reliability in the Cloud**. Software Quality, Reliability and Security Companion (QRS-C), 2016 IEEE International Conference on. **Anais**...2016

AYEWAH, N. et al. Using Static Analysis to Find Bugs. **IEEE Software**, v. 25, n. 5, p. 22–29, 2008.

BANERJEE, S.; SRIKANTH, H.; CUKIC, B. **Log-Based Reliability Analysis of Software as a Service (SaaS)**. 2010 IEEE 21st International Symposium on Software Reliability Engineering. **Anais**...IEEE, nov. 2010Disponível em: <http://ieeexplore.ieee.org/document/5635046/>

BARBOSA, E. A. et al. Enforcing Exception Handling Policies with a Domain-Specific Language. **IEEE Transactions on Software Engineering**, v. 42, n. 6, p. 559–584, 2016.

BARBOSA, E. A.; GARCIA, A. Global-Aware Recommendations for Repairing Violations in Exception Handling. **IEEE Transactions on Software Engineering**, v. 44, n. 9, p. 855–873, 2018.

BASILI, V.; CALDIERA, G.; ROMBACH, H. D. Goal Question Metric (GQM) Paradigm. In: **Encyclopedia of Software Engineering**. [s.l: s.n.].

BAU, J. et al. **State of the art: Automated black-box web application vulnerability testing**. Proceedings - IEEE Symposium on Security and Privacy. **Anais**...2010

BELLER, M. et al. **Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software**. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). **Anais**...2016

BOURQUE, P.; FAIRLEY, R. E.; OTHERS. **Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0**. [s.l.] IEEE Computer Society Press, 2014.

BRIAND, L. C. **Software documentation: how much is enough?** Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on. **Anais**...2003

CHILLAREGE, R. Orthogonal Defect Classification, Ch. 9 of Handbook of Software Reliability Engineering, M. Lyu Ed. **IEEE Computer Society, McGraw-Hill**, 1995.

DAS, S.; LUTTERS, W. G.; SEAMAN, C. B. **Understanding Documentation Value in Software Maintenance**. Proceedings of the 2007

Symposium on Computer Human Interaction for the Management of Information Technology. **Anais**...: CHIMIT '07.ACM, 2007Disponível em: <http://doi.acm.org/10.1145/1234772.1234790>

DAVIS, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. **MIS quarterly**, p. 319–340, 1989.

DOGAN, S.; BETIN-CAN, A.; GAROUSI, V. Web application testing: A systematic literature review. **Journal of Systems and Software**, v. 91, p. 174–201, 2014.

EBERT, F.; CASTOR, F. **A study on developers' perceptions about exception handling bugs**. IEEE International Conference on Software Maintenance, ICSM. **Anais**...2013

ELBAUM, S.; KARRE, S.; ROTHERMEL, G. Improving web application testing with user session data. **Proceedings of 25th International Conference on Software Engineering**, p. 49–59, 2003.

ERSOY, E.; SÖZER, H. **Extending static code analysis with application-specific rules by analyzing runtime execution traces**. International Symposium on Computer and Information Sciences. **Anais**...2016

FORWARD, A.; LETHBRIDGE, T. C. **The relevance of software documentation, tools and technologies: a survey**. Proceedings of the 2002 ACM symposium on Document engineering. **Anais**...2002

GAMMA, E. et al. **Design Patterns – Elements of Reusable Object-Oriented Software**. [s.l: s.n.].

GAROUSI, V. et al. A systematic mapping study of web application testing. **Information and Software Technology**, v. 55, n. 8, p. 1374–1396, 2013.

GOŠEVA-POPSTOJANOVA, K. et al. Empirical Characterization of Session--Based Workload and Reliability for Web Servers. **Empirical Software Engineering**, v. 11, n. 1, p. 71–117, mar. 2006.

GURGEL, A. et al. **Blending and Reusing Rules for Architectural Degradation Prevention**. Proceedings of the 13th International Conference on Modularity. **Anais**...: MODULARITY '14.New York, NY, USA: ACM, 2014Disponível em: <http://doi.acm.org/10.1145/2577080.2577087>

HALFOND, W. G. J.; ANAND, S.; ORSO, A. **Precise interface identification to improve testing and analysis of web applications**. Proceedings of the eighteenth international symposium on Software testing and analysis. **Anais**...2009

HALFOND, W. G. J.; ORSO, A. **Improving test case generation for web applications using automated interface discovery**. Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. **Anais**...2007

HARTLEY, D. Chapter 1 - What Is SQL Injection? In: CLARKE, J. (Ed.). . **SQL Injection Attacks and Defense (Second Edition)**. Second Edi ed. Boston: Syngress, 2012. p. 1–25.

HECKMAN, S.; WILLIAMS, L. A systematic literature review of actionable alert identification techniques for automated static code analysis.

Information and Software Technology, v. 53, n. 4, p. 363–387, 2011.

HUANG, S.; TILLEY, S. **Towards a documentation maturity model**. Proceedings of the 21st annual international conference on Documentation. **Anais**...2003

HUYNH, T.; MILLER, J. **Further investigations into evaluating website reliability**. 2005 International Symposium on Empirical Software Engineering (ISESE), Proceedings. **Anais**...345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE, 2005

HUYNH, T.; MILLER, J. Another viewpoint on "evaluating web software reliability based on workload and failure data extracted from server logs". **Empirical Software Engineering**, v. 14, n. 4, p. 371–396, 2009.

ISO, I. E. C. IEEE, Systems and Software Engineering--Vocabulary. **ISO/IEC/IEEE 24765: 2010 (E)) Piscataway, NJ: IEEE computer society, Tech. Rep.**, 2010.

JAFFAL, W.; TIAN, J. **Defect Analysis and Reliability Assessment for Transactional Web Applications**. Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on. **Anais**...nov. 2014Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6983847>

JEDLITSCHKA, A.; CIOLKOWSKI, M.; PFAHL, D. Reporting experiments in software engineering. In: **Guide to advanced empirical software engineering**. [s.l.] Springer, 2008. p. 201–228.

JOHNSON, B. et al. **Why don't software developers use static analysis tools to find bugs?** 2013 35th International Conference on Software Engineering (ICSE). **Anais**...IEEE, maio 2013Disponível em: <http://ieeexplore.ieee.org/document/6606613/>

JONES, C.; BONSIGNOUR, O. **The economics of software quality**. [s.l.] Addison-Wesley Professional, 2011.

KALINOWSKI, M.; CARD, D. N.; TRAVASSOS, G. H. Evidence-Based Guidelines to Defect Causal Analysis. **IEEE Software**, v. 29, n. 4, p. 16–18, jul. 2012.

KALLEPALLI, C.; TIAN, J. Measuring and modeling usage and reliability for statistical web testing. **IEEE Transactions on Software Engineering**, v. 27, n. 11, p. 1023–1036, 2001.

LI, Y.-F.; DAS, P. K.; DOWE, D. L. Two decades of Web application testing-A survey of recent advances. **Information Systems**, v. 43, p. 20–54, 2014.

MA, L.; TIAN, J. Web error classification and analysis for reliability improvement. **Journal of Systems and Software**, v. 80, n. 6, p. 795–804, 2007.

MUSKE, T.; SEREBRENIK, A. **Survey of approaches for handling static analysis alarms**. Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on. **Anais**...2016

NELSON, E. Estimating software reliability from test data. **Microelectronics Reliability**, v. 17, n. 1, p. 67–73, 1978.

PETROSKI, H.; BARATTA, A. J. To Engineer is Humam—The Role of Failure in Successful Design. **The Physics Teacher**, 1988.

SEAMAN, C. B. Qualitative methods in empirical studies of software engineering. **IEEE Transactions on Software Engineering**, v. 25, n. 4, p. 557–572, 1999.

SHAH, H. B.; GÖRG, C.; HARROLD, M. J. Understanding exception handling: Viewpoints of novices and experts. **IEEE Transactions on Software Engineering**, v. 2, p. 150–161, 2010.

SHAH, H.; GÖRG, C.; HARROLD, M. J. Visualization of exception handling constructs to support program understanding. **Proceedings of the 4th ACM symposium on Software visuallization - SoftVis '08**, p. 19–28, 2008.

SHULL, F.; CARVER, J.; TRAVASSOS, G. H. **An empirical methodology for introducing software processes**. ACM SIGSOFT Software Engineering Notes. **Anais**...2001

SINGER, J. **Practices of software maintenance**. Software Maintenance, 1998. Proceedings., International Conference on. **Anais**...1998

SOHAN, S. M.; ANSLOW, C.; MAURER, F. **SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server (N)**. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). **Anais**...IEEE, nov. 2015Disponível em: <http://ieeexplore.ieee.org/document/7372015/>

SONARSOURCE. **SonarQube**. Disponível em: <https://www.sonarqube.org/>. Acesso em: 20 jul. 2018.

SOUSA, M. J. C.; MOREIRA, H. M. **A survey on the software maintenance process**. Software Maintenance, 1998. Proceedings., International Conference on. **Anais**...1998

SOUZA, S. C. B. DE; ANQUETIL, N.; OLIVEIRA, K. M. DE. Which documentation for software maintenance? **Journal of the Brazilian Computer Society**, v. 12, n. 3, p. 31–44, 2006.

TERRA, R. et al. A recommendation system for repairing violations detected by static architecture conformance checking. **Software - Practice and Experience**, v. 45, n. 3, p. 315–342, 2015.

TERRA, R.; VALENTE, M. T. A dependency constraint language to manage object-oriented software architectures. **Software: Practice and Experience**, v. 39, n. 12, p. 1073–1094, 2009.

TIAN, J. et al. Evaluating Web software reliability based on workload and failure data extracted from server logs. **Software Engineering, IEEE Transactions on**, v. 30, n. 11, p. 754–769, nov. 2004.

VITALJOB SOFTWARE. **inFinance**. Disponível em: <http://www.infinance.com.br/>. Acesso em: 20 jul. 2018.

WIERINGA, R. **Design Science Methodology for Information Systems and Software Engineering**. [s.l: s.n.].

WOHLIN, C. et al. **Experimentation in Software Engineering**. [s.l.] Springer Publishing Company, Incorporated, 2012.

Appendix A – Consent and Characterization Form

Consent Form

I declare to be over 18 years old and that I agree to participate in a study. This study aims at evaluating the acceptability and difficulty of application of patter-driven maintenance method.

The procedure

I understand that I will conduct an application of patter-driven maintenance method. The researchers will conduct the study consisting of collection, analysis and reporting of the exercise data. I understand that I have no obligation to contribute with information about my performance in this exercise and that I can request the removal of my experiment results at any time. I also understand that when data are collected and analyzed, my name will be removed from the data and it will not be used at any moment during the analysis or when the results are presented.

Confidentiality

All information collected in this study is confidential and my name won't be identified at any time. Similarly, I agree to maintain confidentiality of the requested tasks and documents, which are part of the experiment.

Benefits, Freedom to Quit

I understand that I am free to ask questions at any time or to request to not include my information in this study. I understand that I am participating in the empirical study by my own free will with the aim to contribute to the advancement of software engineering.

Name (capital letters):_____

Signature:_____

Characterization Form

Name:_____

Level(B.Sc/Ms.c/D.Sc.)_____

1) What is your experience with software development? (More than one option can be selected. Specify, next to the chosen option, how long the experience lasted)

☐ I've never developed software

☐ I've been developing for my own use _____

☐ I've been developing as team member, related to a course_____

☐ I've been developing as team member, in industry _____

2) What is your experience with software maintenance? (More than one option can be selected. Specify, next to the chosen option, how long the experience lasted)

☐ I've never maintained software

☐ I've been maintaining for my own use _____

☐ I've been maintaining as team member, related to a course_____

☐ I've been maintaining as team member, in industry ____ _____

3) Please, select for each topic the level of your experience following the 5 points scale (look at the subtitle):

Subtitle:
1 = No experience
2 = I studied in a classroom or in a book
3 = I actively practiced in a classroom project
4 = I used it in a project in industry
5 = I used it in several projects in industry

Experience Project					
Java	1	2	3	4	5

General experience with JEE programming (e.g., JSP, Servlets, EL, JSTL, JPA)	1	2	3	4	5
Experience reading java error logs (e.g., stack traces)	1	2	3	4	5
Experience inspecting source code for finding bugs	1	2	3	4	5
Experience programming static analysis rules (rules for alerting defects, such as linters)	1	2	3	4	5

4) How do you rate your English reading and comprehension skills?

▢ Basic

▢ Intermediate

▢ Advanced

Appendix B – Error Log of SisGEE

```
ERROR     2018-09-02 16:22:53,242       br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter       [http-nio-
8080-exec-81]       Exception não tratada no Filter
java.lang.NumberFormatException: For input string: ""
          at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
          at java.lang.Integer.parseInt(Integer.java:580)
          at java.lang.Integer.parseInt(Integer.java:615)
          at
br.cefetrj.sisgee.view.termoaditivo.BuscaTermoAditivoServlet.doPost(BuscaTermoAditivoServlet.java:49)
          at javax.servlet.http.HttpServlet.service(HttpServlet.java:660)
          at javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
          at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
          at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
          at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
          at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
          at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
          at br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter.doFilter(TodasRequisicoesFilter.java:37)
          at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
          at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
          at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:199)
          at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
          at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
          at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:139)
          at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)
          at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:651)
          at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
          at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:343)
          at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:412)
          at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
          at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:754)
          at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1385)
          at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
          at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
          at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
          at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
          at java.lang.Thread.run(Thread.java:748)
ERROR     2018-07-02 18:30:00,695       br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter       [http-nio-
8080-exec-85]       Exception não tratada no Filter
java.lang.ClassCastException: java.lang.Double cannot be cast to java.lang.Float
          at
br.cefetrj.sisgee.view.termoestagio.IncluirTermoEstagioServlet.service(IncluirTermoEstagioServlet.java:60)
          at javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
          at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
          at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
          at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
          at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
          at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
          at org.apache.catalina.core.ApplicationDispatcher.invoke(ApplicationDispatcher.java:712)
          at org.apache.catalina.core.ApplicationDispatcher.processRequest(ApplicationDispatcher.java:459)
          at org.apache.catalina.core.ApplicationDispatcher.doForward(ApplicationDispatcher.java:384)
          at org.apache.catalina.core.ApplicationDispatcher.forward(ApplicationDispatcher.java:312)
          at
br.cefetrj.sisgee.view.termoestagio.FormTermoEstagioServlet.doPost(FormTermoEstagioServlet.java:763)
          at javax.servlet.http.HttpServlet.service(HttpServlet.java:660)
          at javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
          at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
          at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
          at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
          at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
          at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
```

```
        at br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter.doFilter(TodasRequisicoesFilter.java:37)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:199)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:139)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)
        at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:651)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:343)
        at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:412)
        at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:754)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1385)
        at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:748)
ERROR   2018-10-13 17:10:45,087     br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter       [http-nio-
8080-exec-1]        Exception não tratada no Filter
java.lang.NumberFormatException: For input string: ""
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:592)
        at java.lang.Integer.parseInt(Integer.java:615)
        at
br.cefetrj.sisgee.view.termoaditivo.VisualizarTermoEAditivo.doGet(VisualizarTermoEAditivo.java:43)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:634)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter.doFilter(TodasRequisicoesFilter.java:37)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:199)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:139)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)
        at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:651)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:343)
        at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:412)
        at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:754)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1385)
        at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:748)
ERROR   2018-09-24 10:42:01,436     br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter       [http-nio-
8080-exec-17]       Exception não tratada no Filter
java.lang.NullPointerException
        at br.cefetrj.sisgee.view.convenio.RenovarConvenioServlet.doGet(RenovarConvenioServlet.java:44)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:634)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter.doFilter(TodasRequisicoesFilter.java:37)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
```

```
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:199)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:139)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)
        at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:651)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:343)
        at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:412)
        at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:754)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1385)
        at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:748)
ERROR   2018-10-13 17:29:19,662     br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter       [http-nio-
8080-exec-9]        Exception não tratada no Filter
java.lang.NullPointerException
        at
br.cefetrj.sisgee.view.termoaditivo.VisualizarTermoEAditivo.doGet(VisualizarTermoEAditivo.java:50)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:634)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter.doFilter(TodasRequisicoesFilter.java:37)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:199)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:139)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)
        at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:651)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:343)
        at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:412)
        at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:754)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1385)
        at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:748)
ERROR   2018-07-03 00:00:20,057     br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter       [http-nio-
8080-exec-19]       Exception não tratada no Filter
org.apache.jasper.JasperException: /index.jsp (line: [4], column: [1]) File [import_head.jspf] not found
        at org.apache.jasper.compiler.DefaultErrorHandler.jspError(DefaultErrorHandler.java:42)
        at org.apache.jasper.compiler.ErrorDispatcher.dispatch(ErrorDispatcher.java:292)
        at org.apache.jasper.compiler.ErrorDispatcher.jspError(ErrorDispatcher.java:98)
        at org.apache.jasper.compiler.Parser.processIncludeDirective(Parser.java:345)
        at org.apache.jasper.compiler.Parser.parseIncludeDirective(Parser.java:380)
        at org.apache.jasper.compiler.Parser.parseDirective(Parser.java:481)
        at org.apache.jasper.compiler.Parser.parseFileDirectives(Parser.java:1797)
        at org.apache.jasper.compiler.Parser.parse(Parser.java:141)
        at org.apache.jasper.compiler.ParserController.doParse(ParserController.java:244)
        at org.apache.jasper.compiler.ParserController.parseDirectives(ParserController.java:127)
        at org.apache.jasper.compiler.Compiler.generateJava(Compiler.java:202)
        at org.apache.jasper.compiler.Compiler.compile(Compiler.java:385)
        at org.apache.jasper.compiler.Compiler.compile(Compiler.java:362)
        at org.apache.jasper.compiler.Compiler.compile(Compiler.java:346)
        at org.apache.jasper.JspCompilationContext.compile(JspCompilationContext.java:603)
        at org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:369)
        at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:386)
```

```
at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:330)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at br.cefetrj.sisgee.view.filters.TodasRequisicoesFilter.doFilter(TodasRequisicoesFilter.java:42)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:199)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:139)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:92)
at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:651)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:343)
at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:412)
at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:754)
at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1385)
at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
at java.lang.Thread.run(Thread.java:748)
```

Appendix C – Forms of Task 1 – Failure Analysis and Defect Pattern Identification – Pilot Version

Task 1 Description

Together with this task description you received the following documents:

- A JEE error log (stack traces) of a system
- A system source code that generated the error log.

You are asked to inspect the software source code, searching for defect patterns, e.g., defects that occurred because of the same cause.

While doing so, please consider:

- This is an individual work and discussions with colleagues are not allowed.

After finishing this task please upload this document using the following link.
https://goo.gl/forms/YIiMRf1dAvbEVFdc2

Thank you very much for your participation!

Defect Pattern Reporting Form

Name: _____

Start time: _____

End time: _____

Please remember also registering the start and end time of any breaks you might have taken during the exercise.

Failure List

ID	File Name	Line	Exception Type	Error Message

Defect Patterns List

Defect Name	
Description	
Exception Type and Failure Message	
Parameters in Failure Message	
Example of Failure Message	
Class and method of throw	
Defect Characterization	
Defect Code Example	
Fixed Code Example	

Defect Name	
Description	
Failure Message	
Parameters in Failure Message	
Example of Failure Message	
Class and method of throw	
Defect Characterization	
Defect Code Example	
Fixed Code Example	

Follow-up Questionnaire Task1

Name:_____

1) Briefly describe your strategy for detecting defect patterns:

_____
_____
_____
_____
_____
_____

2) Did you consider the time sufficient to conclude your task (Yes/No): _____

If No, please explain your answer:

_____
_____
_____
_____
_____
_____

3) Confidence in the defect patterns reported.

- ▢ **Not confident.**
- ▢ **Little confident.**
- ▢ **Confident.**
- ▢ **Largely confident.**
- ▢ **Completely confident.**

4) How easy was it to perform the task.

- ▢ **Very hard.**
- ▢ **Hard.**
- ▢ **Normal.**
- ▢ **Easy.**
- ▢ **Very easy.**

5) What were the difficulties found during the task?

_____
_____
_____
_____

Appendix D – Forms of Task1 – Failure Analysis and Defect Pattern Identification – Groups A and B Version

Task 1 Description

Together with this task description you received the following documents:

- A JEE error log (stack traces) of a system
- A system source code that generated the error log.

You are asked to inspect the software source code, searching for defect patterns, e.g., defects that occurred because of the same cause.

While doing so, please consider:

- This is an individual work and discussions with colleagues are not allowed.

Thank you very much for your participation!

Defect Pattern Reporting Form

Name: _____

Start time: _____

End time: _____

Please remember also registering the start and end time of any breaks you might have taken during the exercise.

Failure List

ID	File Name	Line	Exception Type	Error Message

Defect Patterns List

Defect Name	
Description	
Exception Type and Failure Message	
Defect Pattern	
Fixed Code Pattern	

Defect Name	
Description	
Exception Type and Failure Message	
Defect Pattern	
Fixed Code Pattern	

Defect Name	
Description	
Exception Type and Failure Message	
Defect Pattern	
Fixed Code Pattern	

Follow-up Questionnaire Task1

Name:_____

1) Briefly describe your strategy for detecting defect patterns:

_____
_____
_____
_____
_____
_____

2) Did you consider the time sufficient to conclude your task (Yes/No): _____

If No, please explain your answer:

_____
_____
_____
_____
_____
_____

3) Confidence in the defect patterns reported.

- ▢ **Not confident.**
- ▢ **Little confident.**
- ▢ **Confident.**
- ▢ **Largely confident.**
- ▢ **Completely confident.**

4) How easy was it to perform the task.

- ▢ **Very hard.**
- ▢ **Hard.**
- ▢ **Normal.**
- ▢ **Easy.**
- ▢ **Very easy.**

5) What were the difficulties found during the task?

_____
_____
_____
_____
_____

Appendix E – Forms of Task 2 – Static Analysis Rule Programming

Task 2 Description

Together with this task description you received the following documents:

- A defect pattern documentation
- A system source code that contains instances of that defect pattern.

You are asked to program a static analysis rule that locate the instances of the defect pattern.

While doing so, please consider:

- This is an individual work and discussions with colleagues are not allowed.

After finishing this task please upload this document using the following link.
https://goo.gl/forms/jY3vKLy4RpouQ6Bi2

Thank you very much for your participation!

Static Analysis Rule Programming Reporting Form

Name: _____

Inspection start time: _____

Inspection end time: _____

Please remember also registering the start and end time of any breaks you might have taken during the exercise.

Your source code enters here:

Follow-up Questionnaire Task2

Name:_____

1) Briefly describe your strategy for programming a static analysis rule:

_____
_____
_____
_____
_____
_____

2) Did you consider the time sufficient to conclude your task (Yes/No): _____

If No, please explain your answer:

_____
_____
_____
_____
_____
_____

3) Confidence in the static analysis rule programmed.

- ☐ **Not confident.**
- ☐ **Little confident.**
- ☐ **Confident.**
- ☐ **Largely confident.**
- ☐ **Completely confident.**

4) How easy was it to perform the task.

- ☐ **Very hard.**
- ☐ **Hard.**
- ☐ **Normal.**
- ☐ **Easy.**
- ☐ **Very easy.**

5) What were the difficulties found during the task?

_____
_____
_____
_____
_____

Appendix F – Forms of Task 3 – Rule Evaluation and Context Analysis – Pilot Version

Task 3 Description

Together with this task description you received the following documents:

- A defect pattern documentation
- A list of defect candidates of this defect pattern.
- A system source code that contains the defect candidates.

You are asked to inspect the defect candidates in source code to confirm if they are defects, identify the contexts of false positives if needed, and calculate the precision and recall of the rule and its contexts.

While doing so, please consider:

- This is an individual work and discussions with colleagues are not allowed.

After finishing this task please upload this document using the following link.
https://goo.gl/forms/umoSposMgAqFo2uq1

Thank you very much for your participation!

Defect Pattern Reporting Form

Name: _____

Start time: _____

End time: _____

Please remember also registering the start and end time of any breaks you might have taken during the exercise.

Defect Candidate List

				Alerted			
Timestamp	File	Line	True Status	Defect Pattern Rule	Ctx1	Ctx2	Ctx3
	PrincipalTermo.java	378					
	BuscaTermoAditivoServlet.java	49					
	BuscaTermoAditivoServlet.java	54					
	FormTermoAditivoServlet.java	115					
	FormTermoAditivoServlet.java	225					
	FormTermoAditivoServlet.java	265					
	VerTermoAditivoServlet.java	48					
	VisualizarTermoEAditivo.java	43					
	VisualizarTermoEAditivo.java	45					
	FormTermoEstagioServlet.java	214					
	FormTermoEstagioServlet.java	600					
	FormTermoEstagioServlet.java	646					
	FormTermoEstagioServlet.java	749					
	FormTermoRescisaoServlet.java	81					
	ValidaUtils.java	232					
	ValidaUtils.java	233					

Precision and Recall of each Rule

Rule	Precision	Recall

Context documentation template

Defect Pattern Name	
Context Name	
Non-Exception Context Description	
Cause	
How to Identify the Context	
Context Code Example	

Defect Pattern Name	
Context Name	
Non-Exception Context Description	
Cause	
How to Identify the Context	
Context Code Example	

Defect Pattern Name	
Context Name	
Non-Exception Context Description	
Cause	
How to Identify the Context	
Context Code Example	

Follow-up Questionnaire Task3

Name:_____

1) Briefly describe your strategy for inspecting the defect candidates:

_____
_____
_____
_____
_____
_____

2) Did you consider the time sufficient to conclude your task (Yes/No): _____

If No, please explain your answer:

_____
_____
_____
_____
_____
_____

3) Confidence in about defect candidate classification (defect / no defect).

- ☐ **Not confident.**
- ☐ **Little confident.**
- ☐ **Confident.**
- ☐ **Largely confident.**
- ☐ **Completely confident.**

4) How ease was to perform the task.

- ☐ **Very hard.**
- ☐ **Hard.**
- ☐ **Normal.**
- ☐ **Easy.**
- ☐ **Very easy.**

5) What were the difficulties found during the task?

_____
_____
_____
_____
_____

Appendix G – Forms of Task 3 – Rule Evaluation and Context Analysis – Group A and B Version

Task 3 Description

Together with this task description you received the following documents:

- A defect pattern documentation
- A list of defect candidates of this defect pattern.
- A system source code that contains the defect candidates.

You are asked to inspect the defect candidates in source code to confirm if they are defects, identify the contexts of false positives if needed, and calculate the precision and recall of the rule and its contexts.

While doing so, please consider:

- This is an individual work and discussions with colleagues are not allowed.

Thank you very much for your participation!

Defect Pattern Reporting Form

Name: _____

Start time: _____

End time: _____

Please remember also registering the start and end time of any breaks you might have taken during the exercise.

Defect Candidate List

File	Line	Is Defect?	Which Context?
PrincipalTermo.java	378		
BuscaTermoAditivoServlet.java	49		
BuscaTermoAditivoServlet.java	54		
FormTermoAditivoServlet.java	115		
FormTermoAditivoServlet.java	225		
FormTermoAditivoServlet.java	265		
VerTermoAditivoServlet.java	48		
VisualizarTermoEAditivo.java	43		
VisualizarTermoEAditivo.java	45		
FormTermoEstagioServlet.java	214		
FormTermoEstagioServlet.java	600		
FormTermoEstagioServlet.java	646		
FormTermoEstagioServlet.java	749		
FormTermoRescisaoServlet.java	81		
ValidaUtils.java	232		
ValidaUtils.java	233		

Context documentation template

Defect Pattern Name	
Context Name	
Context Description	
Context Pattern	

Defect Pattern Name	
Context Name	
Context Description	
Context Pattern	

Defect Pattern Name	
Context Name	
Context Description	
Context Pattern	

Follow-up Questionnaire Task3

Name:_____

1) Briefly describe your strategy for inspecting the defect candidates:

_____
_____
_____
_____
_____
_____

2) Did you consider the time sufficient to conclude your task (Yes/No): _____

If No, please explain your answer:

_____
_____
_____
_____
_____
_____

3) Confidence in about defect candidate classification (defect / no defect).

- ☐ **Not confident.**
- ☐ **Little confident.**
- ☐ **Confident.**
- ☐ **Largely confident.**
- ☐ **Completely confident.**

4) How ease was to perform the task.

- ☐ **Very hard.**
- ☐ **Hard.**
- ☐ **Normal.**
- ☐ **Easy.**
- ☐ **Very easy.**

5) What were the difficulties found during the task?

_____
_____
_____
_____
_____

Appendix H – TAM questionnaire used in the study

Name:

_____

After finishing this task please upload this document using the following link.

https://goo.gl/forms/RPsXJrYX5rt1Ttm03

PDM = Pattern-Driven Maintenance

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Using *PDM* would improve my performance in preventing latent unhandled exceptions (ie., prevent faster)					
Using *PDM* would improve my productivity in preventing latent unhandled exceptions (ie., prevent more and faster)					
Using *PDM* would enhance my effectiveness in preventing latent unhandled exceptions (i.e., prevent more)					
I would find *PDM* useful in preventing latent unhandled exceptions					
Learning to operate *PDM* would be easy for me					
I would find it easy to get *PDM* to prevent unhandled exception					
It would be easy for me to become skillful in the use of *PDM*					
I would find *PDM* easy to use					
I intend to *PDM* regularly at work					