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

Diogo S. Mendonça[1,2], Tarcila G. da Silva[1], Daniel Ferreira de Oliveira[1], Julliany Sales Brandão[1],
[1] Centro Federal de Educação Tecnológica Celso Suckow da Fonseca
Brazil
{diogo.mendonca, tarcila.silva, daniel.oliveira, julliany.brandao}@cefet-rj.br

Helio Lopes[2], Simone D.J. Barbosa[2], Marcos Kalinowski[2], Arndt von Staa[2]
[2] Pontifícia Universidade Católica do Rio de Janeiro
Brazil
{lopes, simone, kalinowski, arndt}@inf.puc-rio.br

ABSTRACT

Background: Unhandled exceptions affect the reliability 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 latent unhandled exceptions automatically 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. **Results:** In both cases, our approach allowed identifying defect patterns and finding latent unhandled exceptions to be fixed in the source code, enabling to completely eliminate the pattern-related failures and improving the application reliability. The static analysis rules produced by PDM achieved a precision of 59-68% in the first application and 89-100% in the second, where lessons learnt from the first evaluation were addressed. **Conclusions:** The results strengthen our confidence that PDM can help maintainers to improve the reliability for unhandled exceptions in other existing web applications.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**

• **Software and its engineering → Software reliability**

KEYWORDS

Maintenance of Web Applications, Reliability, Method, Patterns, Unhandled Exceptions

ACM Reference format:

D.S. Mendonça, T.G. da Silva, D.F. Oliveira, J.S. Brandão, H. Lopes, S.D.J. Barbosa, M. Kalinowski and A.V. Staa. 2018. International Symposium of Empirical Software Engineering and Measurement (ESEM 2018). In *Proceedings of 12th International Symposium on Empirical Software Engineering and Measurement, Oulu, Finland, October 2018 (ESEM'18)*, 10 pages.

1 INTRODUCTION

Maintenance is the most costly phase in the software lifecycle [4]. 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 [20]. Among those failures are the ones generated by exceptions that are not handled by the application, *i.e.*, *unhandled exceptions*, which affect its reliability.

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 [13, 22], showing recurrent occurrence of unhandled exceptions. However, the logs record only the unhandled exceptions thrown during software use. It is possible that the web application has a source code that lacks exception handling, but that has not thrown exceptions yet, thus no log refers to these exceptions. Throughout this paper we name this type of source code problem as *latent unhandled exceptions*.

Identifying latent unhandled 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 with handling exceptions. Additionally, the use of software libraries that may throw exceptions that are unknown to the developer increases the chance of latent unhandled exceptions. These application specificities influence how exception handling should be introduced in the application, thus influencing the identification of unhandled exceptions.

Consequently, it is difficult to detect latent unhandled exceptions automatically. Automated approaches for testing web applications [7, 12, 23] and locating defects using static analysis [17, 24] do not focus on latent unhandled exceptions, thus being inadequate to treat this problem. Application-specific approaches [9] 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.

To fail and to learn from failure are essential parts of the engineering discipline [25] . In this paper we apply this principle to unhandled exceptions. Therefore, inspired by the design science engineering cycle [29], we designed a method called Pattern-Driven Maintenance (PDM) to perform corrective and preventive maintenance of web applications against latent unhandled 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 in the failures and in the application source code to identify source code patterns that trigger unhandled exception, i.e., a defect pattern. Once such 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 with improving the precision of the static analysis rules, working as a learning cycle.

We applied the PDM method to two industrial web applications from different companies and using different technologies. In both applications, applying the method enabled identifying three defect patterns and locating their latent instances statically (using SonarQube [27]). 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 realiability. 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.

This paper is organized as follows: Section 2 presents the research design. Section 3 analyzes the problem and discusses related work. Section 4 presents the PDM method. Section 5 describes the two application cases. Section 6 presents the threats to the validity of the study. Section 7 discusses the results. Finally, Section 8 concludes the paper and suggests future work.

2 RESEARCH DESIGN

Our research design is inspired in the design science engineering cycle [29]. The design problem can be defined using the design science template as:

- **Improve** the reliability of web information systems that present failures caused by unhandled exceptions
- **by** designing a method to automate the localization of the activable unhandled 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.

The design science approach starts with idealized assumptions to produce an artifact that solves a practical problem. Afterwards, engineering cycles are performed with controlled conditions, gathering experience to improve the artifact. Each engineering cycle relaxes gradually the conditions of experimentation by approximating them to practical conditions. Those cycles are performed until the artifact is ready to be used in practice.

In our case, we started designing PDM with 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 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.

After formulating these assumptions, in the first cycle, an initial version of PDM was designed and applied to a small size industrial web application, in controlled conditions (one of the authors 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, our two evaluations aimed to answer the following design science knowledge questions about PDM:

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

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

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

Table 1: Metrics used in PDM evaluation.

Precision	$\frac{Defects\ alerted}{Defects\ alerted + False\ positives\ alerted}$
Relative Recall	$\frac{Defects\ alerted}{Defects\ matching\ the\ intended\ pattern}$
Reliability	$1 - \left(\frac{Failures}{Accesses}\right)$

We use the metrics presented in Table 1 to answer the knowledge questions. It is noteworthy that the recall is relative to the defects matching the intended pattern. Those 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 might include false positives. The purpose of the PDM learning cycle is to improve the precision without harming the relative recall.

The reliability is measured using the number of failures produced by unhandled exceptions during a period, in comparison with the number of accesses on the application in the same period. Sensitivity is discussed based on the experience of applying the method to two independent and different industrial applications.

3 BACKGROUND AND RELATED WORK

In order to provide the background for understanding the rationale used in our solution concept, this section provides an overview on research related to unhandled exceptions in web applications and automated approaches for maintenance of web applications. We discuss the applicability of these approaches to deal with the problem of unhandled 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 [28], low quality [5], or being outdated [10], thus typically not being a trustworthy resource during maintenance [6].

Kallepalli and Tian [22] propose the use of web server access logs to analyze the reliability and perform the statistical testing of web applications. 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.

Goševa-Popstojanova et al. [13] 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. However, the unique errors were used to empirically assess the number of defects and they did not consider identifying defect patterns.

Many primary and some secondary studies addressing automated testing of web applications have been conducted. Among the secondary studies, Li et al. [23] explained the main techniques found in the literature for testing web applications, whereas Garousi et al. [12] and Doğan et al. [7] 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 [8] 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 [2] 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 [16]. 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 [14, 15, 26], also known as web APIs, may be followed by testing as a preventive maintenance approach for web applications. Some techniques use static analysis [15] or symbolic execution [14] of the source code to recover the web API. Sohan et al. [26] 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 [14, 15] and good examples [26] 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 low probability of occurrence [23]. 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 [17, 24]. The software suites used to search for defects in source code using static analysis are called Linters [1]. Open source tools, such as SonarQube [27], 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 to the practical adoption of Linters [19]. Furthermore, fewer than 5% of the static analysis rules used in open source projects are custom rules [3], i.e., application-specific rules. This low usage may indicate difficulty to work with custom rules to detect application-specific defects.

Ersoy and Sözer [9] presented an automated approach to detect 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 paper, differently from that of Ersoy and Sözer [9], 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 on the corrective and preventive maintenance of web applications. In contrast to other approaches to the same purpose and application domain [2, 8, 14, 15, 26], it focuses on latent unhandled exceptions in web applications, including specialized activities to detect and fix them.

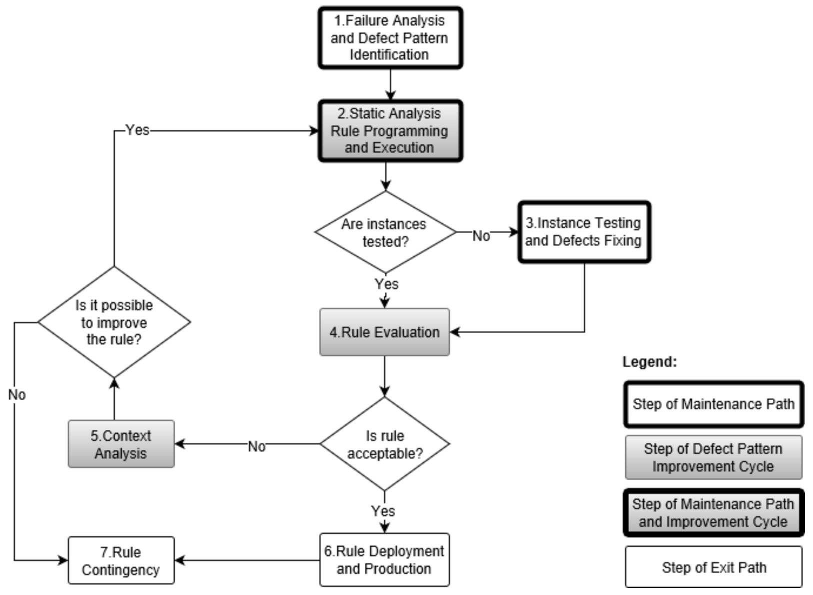
1.Failure Analysis and Defect Pattern Identification
2.Static Analysis Rule Programming and Execution
Are instances tested?
No
Yes
3.Instance Testing and Defects Fixing
4.Rule Evaluation
Is rule acceptable?
No
Yes
5.Context Analysis
Is it possible to improve the rule?
Yes
No
6.Rule Deployment and Production
7.Rule Contingency
Legend:
Step of Maintenance Path
Step of Defect Pattern Improvement Cycle
Step of Maintenance Path and Improvement Cycle
Step of Exit Path

Figure 1: The PDM method

4 SOLUTION CONCEPT

In this section, we explain the proposed Pattern-Driven Maintenance (PDM) method. 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). 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.

The maintenance path includes activities to process the server logs and identify defect patterns (step 1), develop static analysis rules to detect the latent defects (step 2), and to test the detected instances and correct the defects (step 3).

The defect pattern improvement cycle is performed when the evaluation 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. When they 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.

4.1 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 error message and the source code file and line where the unhandled exception occurred. The log processing activity includes the automated 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 [21]. Hence, defect pattern identification consists in 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 the PDM method tests and fixes only the defects identified by the pattern. After finding a pattern, the maintainer must document it.

4.2 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 [27], 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 and continue to the next PDM method activity.

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.

4.3 Instance Testing and Defect Fixing

In step 3, the maintainer tests each instance of the pattern located and corrects the defects found. System level tests should be conducted for each alert instance with the intention to verify if the target exception is thrown. Thus, a test case needs 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 might be difficult.

4.4 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 relative recall of the static analysis rule. Calculating these metrics requires testing all defect instances, which might require significant effort, thus, organizations may opt to conduct more informal rule evaluations.

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 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 [19]. Low precision affects the confidence of developers in defect detection, thus causing them to eventually abandon this feature [19]. 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.

4.5 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 test cases that do not reveal defects, i.e., false positive alerts, provide information related to the context in which the static analysis rule fails to detect a defect (e.g., source code control flow that will not allow throwing the exception). The maintainer inspects the source code exercised by those test looking for the reason why the alert is a false positive. The identified contexts are added to the defect pattern documentation.

After performing context documentation, the maintainer decides whether it is feasible to implement the contexts making modifications on the static analysis rules. If there are feasible contexts, the maintainer performs the steps of programming and executing (2) and evaluating (4) the rules improved by those contexts. Finally, the 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, a new 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).

4.6 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 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.

4.7 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 presents information and examples about the defect patterns. The programming guidelines consolidate the defect patterns with explicit instructions on how to prevent them. The maintainers and developers should receive training on the defect patterns and on how to use the programming guideline.

5 INDUSTRIAL EVALUATIONS

Following the design science methodology [29], 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.

5.1 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 [18]. Although this software has 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 first author of this paper 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 the Django Error log. Table 2 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 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 type of exception, source file, and line, thus resulting in seven complete entries for defect pattern identification.

As shown in Table 2, 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 [21], we investigated the source code where these exceptions occurred for defect pattern identification and identified three defect patterns, presented in Table 6. 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 instances identified by the patter, the rule could be evaluated. Table 3 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. Section 2), the levels of rule precision were not acceptable for the company for defect alerting within the developers' IDE. Hence we started a defect pattern improvement cycle.

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, being from the request, database or a constant. We programmed the rules to identify these contexts in SonarQube. Table 4 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, not detecting 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 2: Number of failures and defects according to 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**

Table 3: 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 4: Evaluation of the rules enhanced with the context.

Rule	**Contexts from Alerts**	**Precision**	**Relative Recall**
Django ORM get	Parameter is not constant (A)	52%	100%
	Parameter is not from the database (B)	63%	100%
	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 5 presents the results of this measurement. We can observe that the number of failures caused by the identified defect patterns had a significant reduction after defect fixing deployment, reaching zero failures at the second months 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 3, *Float Conversion*) to 68% (Table 4, *Django ORM get A and B*), thus the rules were not accepted by the company for alerting developers.

Table 5: 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 6: 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`

Regarding the lessons learnt from applying PDM in this first industrial evaluation, they are twofold. First, concerning the precision, the main reason for not being able to further improve it was that the abstractions needed to represent the source code context of the defect patterns were not present in the static analysis tool selected, namely, SonarQube with 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 testing false positive instances, calling for a faster way to evolve the rules to eliminate false positives as soon as possible.

5.2 Second Evaluation

The second software selected to apply PDM was a small sized administrative web application of an educational institution, written in PHP. The second and third author of the paper are the developers responsible for the software and work for this institution, thus having 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 learnt 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 to achieve higher levels of precision.

We also changed the way we executed the PDM method. 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 3 and Table 4), which required 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 intended 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 8 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 7 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 7, 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 this paper was written, 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 learnt, this PDM application allowed identifying two. 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 [11]. 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 expressive 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.

6 Threats to validity

In this paper we reported on applying an application-specific method in two different industrial contexts to identify, treat and prevent latent unhandled exceptions and improve application reliability. Thus, given the application-specific nature, no further theoretical generalizations or claims were made within the paper 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 was not stopped while PDM was applied. Thus, the maintenance and evolution activiy 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 5 and Table 7), isolating this confounding factor.

Table 7: 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

Table 8: 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_var);`	`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_var);`	`if (!isset($id_var) ||`
			`     empty($id_var) ||`
			`     !is_numeric($id_var)){`
			`     …`
			`}`
			`$res = $someDao->someMethod($id_var);`

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.

External validity. As is common with empirical studies conducted in industry, the method application results are specific to the software applications and their characteristics, not being generalizable. Finally, a single person was responsible for applying the PDM method steps (while discussing them and adjusting decisions jointly with the team of authors). While he was familiar to 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.

7 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, we were able to eliminate failures caused by identified patterns of unhandled exceptions, helping to improve the overall application reliability. The reliability concerning those unhandled exceptions improved in 0.031% (99.969% to 100%) for the first and 0.22% (99.78% to 100%) for the second application. Considering the monthly access profile of the applications, the expectation is to reduce the number of failures per month in 37 and 10, respectively.

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 superior to 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 how PDM steps are performed impacts the application effort. Indeed, the PDM variation applied in the second evaluation, considering the context of false positives as soon as possible, showed to be positive 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 might 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.

8 CONCLUSION

Failures generated by unhandled exceptions affect the reliability of web applications. In this paper, we proposed PDM, a method that iteratively uses static and dynamic analysis to find, correct, and prevents unhandled exceptions in web applications.

We successfully applied the PDM method to two industrial web applications. The method was used to produce application-specific static analysis rules that alert unhandled exception patterns directly in the source code. Moreover, the method allowed iteratively improving the precision of the rules in revealing defects. In our first evaluation, we achieved a precision of rules of 59-68%, which was not accepted for defect alerting during development. Based on lessons learned this result could be improved in our second evaluation, achieving a precision of rules of 89-100%, which was accepted for defect prevention purposes during development. In both cases, software maintenance using PDM allowed eliminating the failures caused by the identified defect patterns, thus helping to improve the applications' reliability.

These results strengthen our confidence that PDM can help maintainers to deal with the problem of unhandled exceptions in web applications. As future work, we intend to perform other PDM evaluations, including independent replications by less experienced professionals, and to investigate the extent to which rules produced by PDM in one application could be reused in another one.

ACKNOWLEDGMENTS

This work was partially supported by the CNPq grant 141345/2015-2.

REFERENCES

[1] Ayewah, N. et al. 2008. Using Static Analysis to Find Bugs. *IEEE Software*. 25, 5 (2008), 22–29. DOI:https://doi.org/10.1109/MS.2008.130.

[2] Bau, J. et al. 2010. State of the art: Automated black-box web application vulnerability testing. *Proceedings - IEEE Symposium on Security and Privacy* (2010), 332–345.

[3] Beller, M. et al. 2016. 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)* (2016), 470–481.

[4] Bourque, P. et al. 2014. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

[5] Briand, L.C. 2003. Software documentation: how much is enough? *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on* (2003), 13–15.

[6] Das, S. et al. 2007. Understanding Documentation Value in Software Maintenance. *Proceedings of the 2007 Symposium on Computer Human Interaction for the Management of Information Technology* (2007).

[7] Dogan, S. et al. 2014. Web application testing: A systematic literature review. *Journal of Systems and Software*. 91, (2014), 174–201.

[8] Elbaum, S. et al. 2003. Improving web application testing with user session data. *Proceedings of 25th International Conference on Software Engineering*. (2003), 49–59.

[9] Ersoy, E. and Sözer, H. 2016. Extending static code analysis with application-specific rules by analyzing runtime execution traces. *International Symposium on Computer and Information Sciences* (2016), 30–38.

[10] Forward, A. and Lethbridge, T.C. 2002. The relevance of software documentation, tools and technologies: a survey. *Proceedings of the 2002 ACM symposium on Document engineering* (2002), 26–33.

[11] Gamma, E. et al. 2002. *Design Patterns – Elements of Reusable Object-Oriented Software*.

[12] Garousi, V. et al. 2013. A systematic mapping study of web application testing. *Information and Software Technology*. 55, 8 (2013), 1374–1396.

[13] Goševa-Popstojanova, K. et al. 2006. Empirical Characterization of Session--Based Workload and Reliability for Web Servers. *Empirical Software Engineering*. 11, 1 (Mar. 2006), 71–117. DOI:https://doi.org/10.1007/s10664-006-5966-7.

[14] Halfond, W.G.J. et al. 2009. Precise interface identification to improve testing and analysis of web applications. *Proceedings of the eighteenth international symposium on Software testing and analysis* (2009), 285–296.

[15] Halfond, W.G.J. and Orso, A. 2007. 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* (2007), 145–154.

[16] Hartley, D. 2012. Chapter 1 - What Is SQL Injection? *SQL Injection Attacks and Defense (Second Edition)*. J. Clarke, ed. Syngress. 1–25.

[17] Heckman, S. and Williams, L. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*. 53, 4 (2011), 363–387.

[18] inFinance: 2010. *http://www.infinance.com.br/*. Accessed: 2018-07-20.

[19] Johnson, B. et al. 2013. Why don't software developers use static analysis tools to find bugs? *2013 35th International Conference on Software Engineering (ICSE)* (May 2013), 672–681.

[20] Jones, C. and Bonsignour, O. 2011. *The economics of software quality*. Addison-Wesley Professional.

[21] Kalinowski, M. et al. 2012. Evidence-Based Guidelines to Defect Causal Analysis. *IEEE Software*. 29, 4 (Jul. 2012), 16–18. DOI:https://doi.org/10.1109/MS.2012.72.

[22] Kallepalli, C. and Tian, J. 2001. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*. 27, 11 (2001), 1023–1036.

[23] Li, Y.-F. et al. 2014. Two decades of Web application testing-A survey of recent advances. *Information Systems*. 43, (2014), 20–54.

[24] Muske, T. and Serebrenik, A. 2016. Survey of approaches for handling static analysis alarms. *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on* (2016), 157–166.

[25] Petroski, H. and Baratta, A.J. 1988. To Engineer is Humam—The Role of Failure in Successful Design. *The Physics Teacher*. (1988).

[26] Sohan, S.M. et al. 2015. SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server (N). *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov. 2015), 271–276.

[27] SonarQube: 2008. *https://www.sonarqube.org/*. Accessed: 2018-07-20.

[28] Souza, S.C.B. de et al. 2006. Which documentation for software maintenance? *Journal of the Brazilian Computer Society*. 12, 3 (2006), 31–44.

[29] Wieringa, R. 2014. *Design Science Methodology for Information Systems and Software Engineering*.