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

Diogo S. Mendonça [1,2], Arndt von Staa [2], Marcos Kalinowski [2]

[1] Escola de Informática e Computação – Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ) - Rio de Janeiro – RJ – Brasil

[2] Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) - Rio de Janeiro – RJ – Brasil

diogo.mendonca@cefet-rj.br, {arndt, kalinowski}@inf.puc-rio.br

Abstract. Background: *Unhandled exceptions affect the reliability, usability, and security of web applications. 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 designed a method called Pattern-Driven Maintenance (PDM), which relies on identifying defect patterns from failures and producing custom static analysis rules that can be used for prevention. We applied PDM to two industrial web applications measuring the reliability improvement, evaluated reuse of static analysis rules produced on within- and cross-company software, and studied the effectiveness, challenges faced, and acceptance of novice maintainers on applying PDM.* ***Results****: In both industry cases, we eliminated pattern-related failures improving the application reliability. Some of the static analysis rules produced by applying PDM were reused on within- and cross-company software. We identified knowledge and experiences that influence effectively applying the steps of the PDM method.* ***Conclusions****: PDM can help maintainers to improve the reliability of existing 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.*

Resumo. Contexto*: Exceções não tratadas afetam a confiabilidade, usabilidade e segurança em aplicações web. 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 projetamos o método chamado Manutenção Orientada a Padrões (Pattern-Driven Maintenance - PDM), que consiste em identificar padrões de defeitos se baseando em falhas para produzir regras de análise estática que podem ser utilizadas para a prevenção de defeitos. Nós aplicamos PDM em duas aplicações web na indústria medindo a melhoria na confiabilidade das aplicações. Nós também avaliamos o reuso das regras de análise estática*

produzidas na mesma empresa e em outras empresas. Finalmente, nós estudamos a eficácia, os desafios encontrados e a aceitação de mantenedores novatos aplicando o método PDM. ***Resultados****: Nos dois casos industriais, nós eliminamos completamente as falhas relacionadas a exceções não tratadas latentes 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.* ***Conclusões****: O PDM pode ajudar os mantenedores a melhorar a confiabilidade de aplicações existentes. Nós disponibilizamos orientações sobre como utilizar o método, reutilizar as regras de análise estática produzidas, e quais conhecimentos e experiências são necessários para aplicar o PDM com eficácia.*

1. Introduction

Maintenance is the most costly phase in the software lifecycle (Bourque, Fairley, and others 2014). Defect prevention and correction activities consume part of these resources. Additionally, the impact of failures in software in use can range from a slight inconvenience to severe damage, including economic ones (Jones and 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 (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 thrown exceptions yet. We refer to this type of source code problem as *unhandled latent exceptions*.

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

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

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

analysis (Heckman and Williams 2011; Muske and Serebrenik 2016) do not focus on unhandled latent exceptions, thus they are inadequate to treat this problem. Application-specific approaches (Ersoy and 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.

To fail and to learn from failure are essential parts of the engineering discipline (Petroski and Baratta 1988). 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

The remainder of this extended summary of Mendonça (2019) thesis is organized as follows: Section 2 presents our research method. Section 3 presents the PDM method. Section 4 presents our results. We conclude and present our contributions in Section 5.

2. Methodology

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

[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.

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 or inspection, revealing their latent defects. The verification 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 (Mendonça et al., 2018). 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 (Mendonça & Kalinowski, 2020). 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 (Mendonça & Kalinowski, 2020b). 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).

3. Pattern-Driven Maintenance

In this section, we briefly explain the proposed Pattern-Driven Maintenance (PDM) method. Figure 1 shows the activities collapsed into steps along with 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).

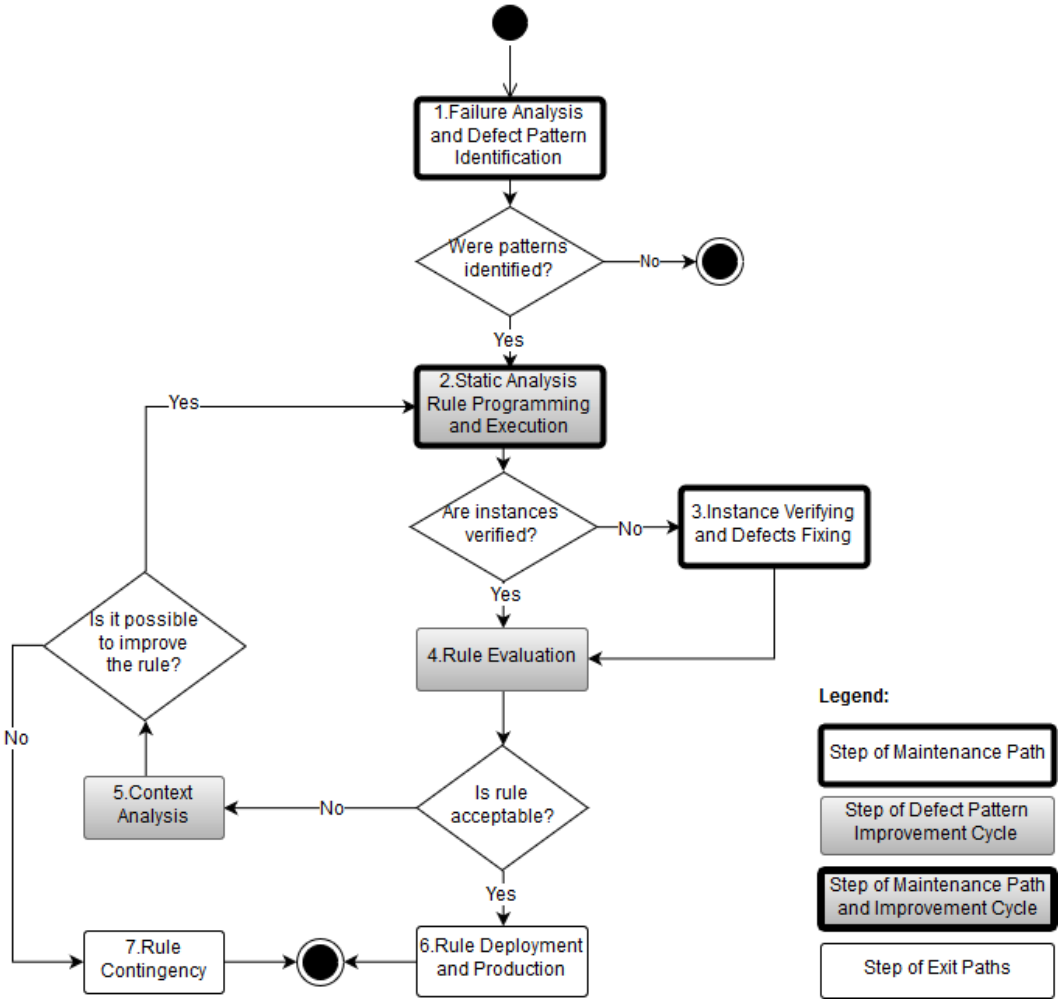

Figure 1: The PDM method control flow

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 inMendonça et al. (2018)..

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 in Mendonça (2019).

4. Results and Discussion

In this section, we answer our design science knowledge questions about PDM.

We applied the PDM method to two industrial web applications from different companies and using different technologies (-Mendonça et al. 2018). 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. 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 (RQ1). We also evaluated the static analysis rules produced by the PDM method (RQ2). 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 (-Mendonça et al. 2018).

As noticed in our lessons learned (RQ3), 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 (-Mendonça et al. 2018).

We found that rules produced by applying PDM might be reused in within- or cross-company environments (RQ4), and not only for software in the maintenance phase, but also recently developed ones (Mendonça & Kalinowski, 2020a). 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 (RQ6), 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 (RQ5) (Mendonça & Kalinowski, 2020a). We also observed that previous successful experience with PDM influences rule reuse adoption (RQ6) (Mendonça & Kalinowski, 2020a).

Based on our experience, we recommend some practices for the evaluation and implementation of the reuse of rules produced by PDM (Mendonça & Kalinowski, 2020a). 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.

We have evaluated PDM regarding maintainers' effectiveness in applying it and their acceptance of the method (RQ7) (Mendonça & Kalinowski, 2020b). 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 the 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. We also found that task 2 requires experience working with abstract syntax trees and static analysis rule programming.

Finally, the maintainers answered a TAM questionnaire about PDM acceptance (RQ8) (Mendonça & Kalinowski, 2020b). Most of them found PDM useful but not easy to apply. However, the perceived ease of use of PDM could be hindered by the conditions of the limited time of an observational study.

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

5. Conclusion

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 significant 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 main 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 (Mendonça & Kalinowski, 2020a). 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).

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.*

After checking the reuse of defect patterns produced by PDM, we still had doubts if maintainers would effectively apply (RQ7) and accept (RQ8) PDM

(Mendonça & Kalinowski, 2020b). This doubt was justified because only the maintainer that created the method (the author of this thesis) had applied PDM and he has 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 properly select or train maintainers for applying the method. In this way, we state our third contribution:

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

The papers produced that are directly related to this thesis can be found in Table 1. 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.

We also intend to develop tools to facilitate PDM's application. The difficulties collected during PDM's acceptance study showed that novice maintainers have difficulties in identifying and documenting defect patterns. They also have problems with implementing static analysis rules. Tool support for these activities could help novice maintainers to effectively apply PDM.

Table 1. Papers produced throughout 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	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.; KALINOWSKI, Marcos. Towards Practical Reuse of Custom Static Analysis Rules for Defect Localization. In: **Proceedings of Simpósio Brasileiro de Qualidade de Software (SBQS'20).** ACM, 2020, 10 pages.	3, 5	Published
MENDONÇA, Diogo S.; KALINOWSKI, Marcos. An Empirical Investigation on the Challenges of Creating Custom Static Analysis Rules for Defect Localization. **Information and Software Technology (IST)**. Elsevier, 2021. E-print available at ArXiv.	6	Written, under submission

Referências

Bourque, Pierre, Richard E Fairley, and others. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

Dogan, Serdar, Aysu Betin-Can, and Vahid Garousi. 2014. “Web Application Testing: A Systematic Literature Review.” *Journal of Systems and Software* 91: 174–201.

Ersoy, Ersin, and Hasan Sözer. 2016. “Extending Static Code Analysis with Application-Specific Rules by Analyzing Runtime Execution Traces.” In *International Symposium on Computer and Information Sciences*, 30–38.

Garousi, Vahid, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. 2013. “A Systematic Mapping Study of Web Application Testing.” *Information and Software Technology* 55 (8): 1374–96.

Heckman, Sarah, and Laurie Williams. 2011. “A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis.” *Information and Software Technology* 53 (4): 363–87.

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

Jones, Capers, and Olivier Bonsignour. 2011. *The Economics of Software Quality*. Addison-Wesley Professional.

Li, Yuan-Fang, Paramjit K Das, and David L Dowe. 2014. “Two Decades of Web Application Testing-A Survey of Recent Advances.” *Information Systems* 43: 20–54.

Mendonça, D.S., T.G. Da Silva, D.F. De Oliveira, J.S. Brando, H. Lopes, S.D.J. Barbosa, M. Kalinowski, and A. Von Staa. 2018. “Applying Pattern-Driven Maintenance: A Method to Prevent Latent Unhandled Exceptions in Web Applications.” In *International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1145/3239235.3268924.

Mendonça, Diogo, and Marcos Kalinowski. 2020a. “Towards Practical Reuse of Custom Static Analysis Rules for Defect Localization.” In *Proceedings of Simpósio Brasileiro de Qualidade de Software (SBQS’20)*, 10. Natal: ACM.

Mendonça, Diogo Silveira. 2019. “Pattern-Driven Maintenance: A Method to Prevent Unhandled Latent Exceptions in Web Applications.” PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO - PUC-RIO. https://doi.org/10.17771/pucrio.acad.45455.

Mendonça, Diogo Silveira, and Marcos Kalinowski. 2020b. “An Empirical Investigation on the Challenges of Creating Custom Static Analysis Rules for Defect Localization.” *ArXiv*, November. http://arxiv.org/abs/2011.12886.

Muske, Tukaram, and Alexander Serebrenik. 2016. “Survey of Approaches for Handling Static Analysis Alarms.” In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference On*, 157–66.

Petroski, Henry, and Anthony J. Baratta. 1988. “To Engineer Is Humam—The Role of Failure in Successful Design.” *The Physics Teacher*.

SonarSource. 2008. “SonarQube.” 2008. https://www.sonarqube.org/.

Wieringa, Roel. 2014. *Design Science Methodology for Information Systems and Software Engineering*. *Springer Berlin Heidelberg*. https://doi.org/10.1145/1810295.1810446.