CrashAwareDev: Supporting Software Development based on Crash Report Mining and Analysis

Leandro Beserra[*†], Roberta Coelho[†]
[*]Informatics Superintendence
[†]Department of Informatics and Applied Mathematics
Federal University of Rio Grande do Norte
Natal, Brazil
ldbeserra@info.ufrn.br, roberta@dimap.ufrn.br

***Abstract*— Exception handling mechanisms are a common feature of mainstream programming languages to deal with exceptional conditions. If on the one hand they allow the programmer to prepare the code to deal with exceptional conditions, on the other hand they can become a source of bugs that threatens the application robustness – studies have shown that uncaught exceptions are the main cause of application crashes. To keep track of crashes and enable an easier fault localization, several applications, nowadays, use crash reporting tools. In this work, we propose a tool named CrashAwareDev, integrated with the Eclipse IDE, which mines the information available in crash reporting tools to support programmers in their day-to-day activities. The proposed tool alerts developers about the classes related to recent crashes and warns about source code characteristics (bug patterns) that can be related to future crashes (similar to the ones that have happened). Doing so the tool aims at bringing the development environment closer to crash reporting tools, that are usually only used for bug fixing or for extracting robustness metrics. A case study was conducted and showed that the tool can support software development by displaying bug pattern alerts directly in the source code, signaling the classes involved in recent faults, and speeding up the crash's fault localization within the development environment itself.**

***Index Terms*— Exception handling, crash report, Eclipse plug-in, uncaught exceptions**

I. Introduction

Exception handling mechanisms [1] are a common feature of modern programming languages. Exception handling structures are used to deal with unexpected events that occur during the execution of a program [2], allowing exceptions to be thrown, captured, and handled at different points in the system. However, the exception handling code designed to make a system more robust often works the other way around and become a burden programmers has to cope with, leading to bugs such as the uncaught exceptions. The uncaught exceptions are the main cause of crashes in Java software systems [3]. A crash is an abnormal behavior of a system that leads to the interruption of its execution.

After a crash occurs, systems typically store information related to the crash on crash report systems. Such information usually contains the uncaught exception that caused the crash and its stack trace. The exception stack trace is a representation of the method call stack and contains information about classes and methods by which an exception was propagated. Since the exception stack trace is a source of information widely used by programmers while debugging they are often added on crash reports [4]. Moreover, these reports may contain other information such as the operating system used at the time of the crash, the user login, the browser/system version, the user's IP address, and other request parameters. In addition to facilitating the fault localization, the information available on crash report tools can assist in prioritizing bug corrections (depending on the number of users affected, for example) or understanding the impact of system crashes [5]. The utility of crash report systems may go beyond recording crash data and supporting debugging. Some studies have shown

DOI reference number: 10.18293/SEKE2019-002

that such information can be used for various purposes such as fault classification [6][7] and fault localization [8][9][10]. None of the existing works, however, extract data from crash reports to support programmers during system coding.

In this paper, we propose a way to mine information stored in crash reports and provide useful information to the programmer within his/her programming environment. We present a tool called CrashAwareDev, an Eclipse[1] IDE plug-in, which supports the developer in coding time by (i) alerting him/her about the classes related to recent crashes; (ii) warning about source code characteristics (bug patterns) that can be related to future crashes (similar to the ones that have happened); and (iii) providing direct access to crash reports within the IDE. A case study was conducted on an industrial web-based software system comprising of 1300 KLOC of Java source code. Along the evaluation period, a group of 5 developers used the tool on a daily basis (during 4 days). Overall the tool presented 95 warnings (i.e., 17 class alerts and 78 bug pattern warnings).

II. Background

The Java programming language provides an exception handling mechanism to support error handling [12]. When an error occurs during execution of code in a try block, the error is caught and handled by an exception handler in one of the subsequent catch blocks associated with it. If no catch block can handle the error, the method is terminated abnormally and the Java virtual machine (JVM) searches backward through the call stack to find an exception handler that can handle the error [15].

In Java, exceptions are represented according to a class hierarchy, on which every exception is an instance of the Throwable class, and can be of three kinds: the checked exceptions (extends Exception), the runtime exceptions (extends RuntimeException) and errors (extends Error) [12].

Checked exceptions represent conditions that, although exceptional, can reasonably be expected to occur, and if they do occur must be dealt with in some way. Unchecked runtime exceptions represent conditions that, generally speaking, reflect errors in your program's logic and cannot be reasonably recovered from at run time [13]. By convention, instances of Error represent unrecoverable conditions which usually result from failures detected by the Java Virtual Machine due to resource limitations, such as OutOfMemoryError. Normally these cannot be handled inside the application [13].

[1]https://www.eclipse.org/

III. Analyzing Crash Reports

A. Methodology

Before implementing the CrashAwareDev tool, we conducted a study whose goal was to identify the characteristics of most frequent crashes of an industrial Web-based system, and based on such characteristics, check wether or not existing static analysis tools could alert about them. Figure 1 illustrates the main steps taken in this study. The target system used in this study was an industrial Web-based system, named SIPAC, comprising of 1300 KLOC of Java source code ans designed to automate business processes for universities focusing on different and complementary aspects, such as administration, planning and management – SIPAC is used in approximatelly 55 Brazilian universities.

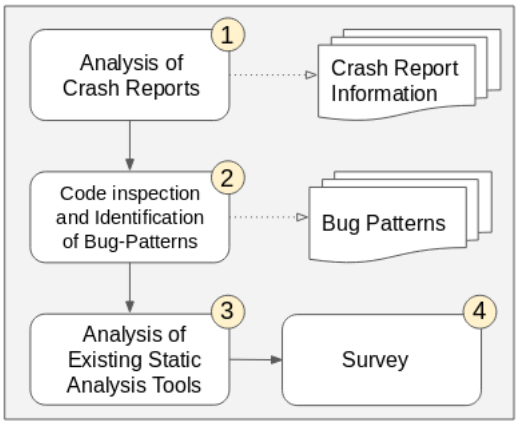

Fig. 1. Methodology overview.

Step 1: Analysis of Crash Reports. We analyzed the most frequent exceptions that caused crashes over a period of one month.

Step 2: Code Inspection and Identification of Bug Patterns. We manually inspected the code related to the most frequent kinds system crashes to identify common characteristics among them.

Step 3: Analysis of Existing Static Analysis Tools. We analyzed whether or not existing static analysis tools could be able to detect and therefore prevent such causes of crashes identified on the previous step.

Step 4: Survey. Finally, we conducted a study on which we surveyed a group of developers (working on the target system) to evaluate how they used.

B. Crash Analysis

In order to investigate the main causes of crashes reported for the target system, we analyzed the crash reports of one month (June 2018). In this period (2018-06-01 to 2018-06-30), approximately 1933617 accesses were made to the system, from this set 680 accesses were faulty (0.0035% of the accesses) - on which the user faced one or more crashes. We observed that approximately 1966 crashes were recorded in the period, affecting 347 distinct users.

We extracted the types of most frequent uncaught exceptions (leading to crashes). The top 5 types of uncaught exceptions raised shown in Table I. We can observe that approximately 63.9% (1241 errors of 1966) of the crashes of the analyzed period was caused by five types of exceptions.

Root Cause	Ocurr.	%	Analyzed
NullPointerException	749	38,09	10
LazyInitializationException	171	8,69	2
JspException	166	8,44	21
IllegalArgumentException	102	5,18	1
PSQLException	53	2,69	5
Total	**1241**	**63,9**	**39**

TABLE I
THE TOP 5 TYPES OF UNCAUGHT EXCEPTIONS RAISED IN JUNE/2018.

We then manually inspected the source code related to a subset of such crashes (we randomly selected a subset of each of the top 5 uncaught exceptions – as shown in the last column of Table I). The purpose of this analysis was to identify common characteristics related to the causes of the most frequent uncaught exceptions.

Overall we inspected the source code related to 39 crashes. We could observe common characteristics among some of them, which led to the definition of four specific bug patterns listed in Table II.

Tag	Description	Ocurr.
BP01	Unchecked database query return	3
BP02	Unchecked classes methods parameters	2
BP03	Unchecked request parameters	2
BP04	Controllers do not implement get/set methods	4

TABLE II
BUG PATTERNS DETECTED DURING ANALYSIS.

Each bug pattern found in this study is described as follows:

- **BP01:** When querying any data in a database, it is recommended to check if the value is not null, to prevent a possible null pointer exception.
- **BP02:** Static methods of utility classes should check if their arguments are not null.
- **BP03:** Like BP01, objects retrieved from the HTTP session must be checked.
- **BP04:** Private attributes of view controllers must most often have the get and set methods implemented. Otherwise, a JspException may be thrown while rendering the Web page.

C. Analysis of Existing Static Analysis Tools

We then investigated whether some of the existing static analysis tools could alert about some of the 39 crash causes identified during the manual inspection. We used SonarLint[2] and SpotBugs[3], both in its Eclipse's plug-in version and observed that none of the 39 defects were detected by the tools.

D. Survey

We applied a survey the developers working on the target system to investigate how they use the crash report information during development. The survey was sent to 25 developers, and we obtained 14 responses, from which: 5 respondents mentioned that never used the crash report; 8 respondents mentioned that only used for debugging purposes; and only one mentioned that used the crash report on a daily basis for monitoring of errors. Moreover,

[2]https://www.sonarlint.org/
[3]https://spotbugs.github.io

7 respondents mentioned that they had difficulties in using the features of the crash report system used.

IV. The CrashAwareDev Tool

Based on the steps described previously we implemented a tool – called CrashAwareDev. CrashAwareDev's main purpose is to present information mined from crash reports and identified bug patterns on the developer's IDE, at coding time. One of the motivations of this research was the perception that, in the context of target system development (SIPAC), crash report information is basically used for debugging failures and was not used to alert the developer of potential crash causes or classes frequently related to crashes. Figure 2 represents an overview of the main features of CrashAwareDev and described next.

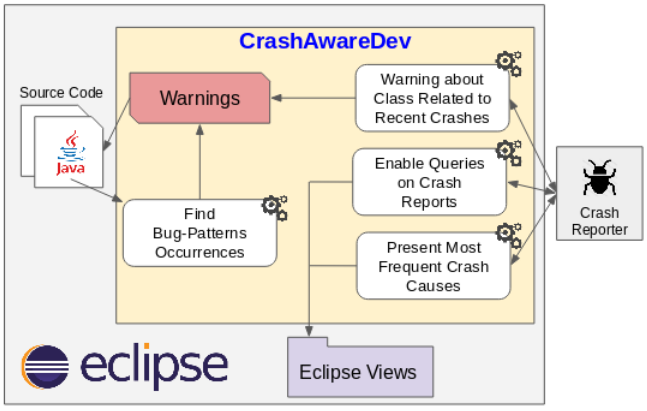

Fig. 2. CrashAwareDev's Components diagram.

Warning about Classes related to Recent Crashes. The tool alerts whether the class being changed was associated with at least one recent system crash (i.e., the period considered is configurable). We have adopted the following heuristic: if the class appears in the exception stack trace of one crash, then it will be associated with the crash, as this means that the exception at some point was propagated within a method of this class. However, this is not to say that the defect is located in that class, but can information help to identify the fault's origin. The tool generates warnings that are displayed in source code during development as illustrated in Figure 3.

Find Bug-Patterns Ocurrences. On the study described previously, we identified source code

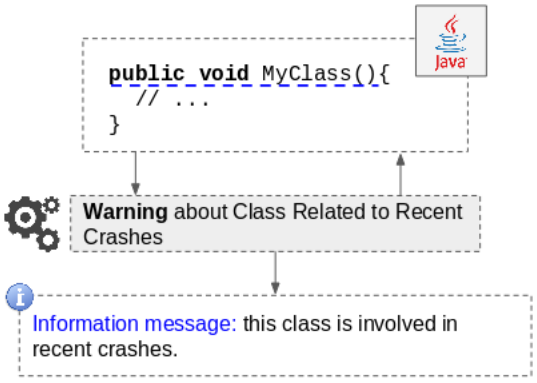

Fig. 3. Warning about classes related to recent crashes.

characteristics (bug patterns) that lead to crashes in SIPAC. Hence, at each compilation, the tool statically analyzes of the changed artifacts looking for bug patterns (described in Table II) and warnings that are displayed directly in IDE as illustrated in Figure 4.

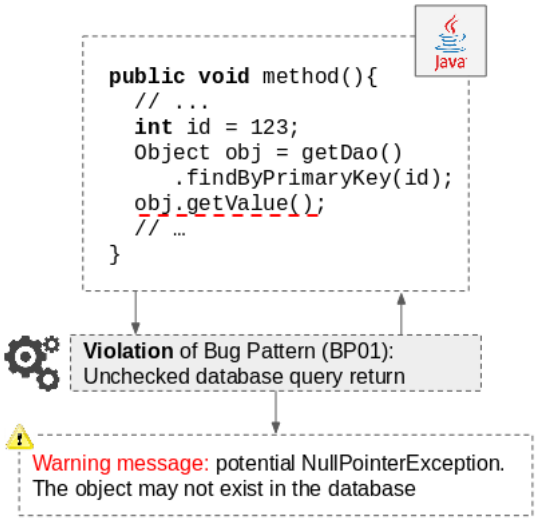

Fig. 4. Bug-Pattern Violation Alert.

Enable Queries on Crash Reports. This feature aims to bring the programmer closer to the crash report during coding. The goal is to enable the developer to query for recent system crashes, filtering them by class name. Summarized crash information is displayed in an Eclipse view with a link to display the complete information directly in the crash reporter (in an external browser).

Present Most Frequent Crash Causes. This feature basically displays the most common types of uncaught exceptions that have occurred in a

given period of time (e.g., one month). The data displayed are similar to those shown in Table I.

V. Evaluation

To evaluate CrashAwareDev tool we performed a case study on which a group of developers used the tool for a given period of time and quantitative and qualitative data regarding the tool usage was collected. We invited SIPAC developers to participate in the study and a group of 5 developers volunteered to participate. They received brief training on the features of CrashAwareDev, and used the tool for 4 consecutive days, for approximately 8 hours daily. Table III presents the metrics collected in this study.

Metric	Count
M1: Number of classes changed	105
M2: Number of classes with some alert	61
M3: Number of alerts per bug pattern	78
BP01 - Check values when querying in a database	47
BP02 - Check auxiliary methods arguments	15
BP03 - Check values carried by request	16
BP04 - Implement private attributes' get/set	N/A
M4: Class alerts associated with a crash	17

TABLE III
METRICS COLLECTED DURING EVALUATION

During the execution period, we collected in the logs that 105 different classes were changed (M1). In 61 of these, some problem was detected by the tool and at least one warning was shown to the developer (M2). A total of 78 bug pattern alerts were displayed. They were distributed among the patterns listed in Table III (M3).

The BP04 pattern was removed during the execution of the study because the participants reported that most warnings would be false positives, as not every private method of controllers should necessarily be referenced in JSP pages. Therefore, we disabled the checking of this pattern during the study. As described earlier, the *Class Checker* feature analyzes whether the changed class is present in some recent crash stack trace. We collected 17 warnings on this check (M4).

Moreover, we also counted how many times each CrashAwareDev feature was used during the study (Q5). The *Query Crashes from Crash Reporter* function was used four times by the participants. We observed that this query was performed after an alert was displayed in the class (during the *Class Checker* execution) and the developer was interested in checking the crashes in which the class was involved. Of these four queries, in two times the participant used the link to see the complete information of the crash. The *Query Top Root Causes* feature was not used during this study. This information did not prove useful to programmers during the study, but we believe that it is of greater interest to leader developers (who did not participate in the study).

We also questioned participants about the usefulness of the tool. All of them mentioned that the tool warnings were useful. One of the participants also mentioned that more bug-checking rules could have been implemented as they could indeed help in preventing crashes in the long run. Two other developers also suggested that in some cases the tool might also support the fault localization in the production environment in a future version.

VI. Related Work

Information mining on crash reporters and/or stack traces. Some studies were carried out with the objective of extracting data from crash reports and using them for various purposes. Among them are those who used the data to find bug hazards in exception handling code [14], to classify the types of failures [6][7] and to facilitate their location [8][9][10]. In this article, we use a grouping of crashes per type of exception. We sought to identify error patterns associated with these types through manual analysis of crashes. Our research did not propose any mechanism for locating defects, but we tried to approach the programmer's development environment to the crash reporter, which may help at coding time the identification of classes associated to recent faults.

Bug Detection Tools. Several static analysis tools have been proposed to detect bug patterns in the source code from predefined rules. These patterns are categorized into various types such as correctness, code smells, vulnerability, security, performance, etc. PMD [17] is an open-source application and is based on bug patterns to find errors in the Java source code. It allows new patterns to be written in Java or XPath. Another tool proposed was the FindBugs [18] which also

examines the source code and bytecode of Java programs. In this paper a subset of the FindBugs rules was described and compared to the PMD with respect to the number of alerts generated, showing that the number of FindBugs alerts is lower in all experiments done. In 2016 FindBugs was discontinued and succeeded by SpotBugs, which we use in this work. Another existing static analysis tool is SonarQube [19] which presented a proposal for analysis along with continuous integration systems. We used in this work your Eclipse version, called SonarLint. CrashAwareDev has a static analysis feature, as well as the tools mentioned. However, we seek to define rules based on real crashes, in order to reduce the known false positives in the existing tools.

VII. Concluding Remarks

In this work, we presented a way to use data from crash reports to support programmers during software development. To promote this support, we have developed a plug-in tool for the Eclipse IDE, CrashAwareDev. Before proposing the tool, we performed a detailed study of crashes stored in a real crash reporter in order to identify the main causes of crashes. A set of common causes of crashes were defined as bug patterns which could be identified in the static analysis performed by CrashAwareDev. Moreover, the tool also alerts the developer about classes frequently involved in crashes and enabled them to access information of crash reports within the IDE. A case study in a real development context was performed and the results revealed that the tool could indeed alert the developers about several application-specific bug patterns.

Acknowledgments. This research has been supported by CAPES-BRAZIL.

References

[1] Goodenough J. B. Exception handling: Issues and a proposed notation.Commun.ACM, ACM, New York, NY, USA, v. 18, n. 12, p. 683–696, dez. 1975. ISSN 0001-0782.

[2] Sawadpong P.; Allen E. B. Software defect prediction using exception handling call graphs: A case study. In:2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE). [S.l.: s.n.], 2016. p. 55–62. ISSN 1530-2059.

[3] Jo J.-W. et al. An uncaught exception analysis for java. Journal of Systems and Software, v. 72, n. 1, p. 59 – 69, 2004. ISSN 0164-1212.

[4] Schroter A. et al. Do stack traces help developers fix bugs? In:2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). [S.l.: s.n.], 2010. p.118–121. ISSN 2160-1852.

[5] An L.; Khomh F. Challenges and issues of mining crash reports. In:2015 IEEE 1st International Workshop on Software Analytics (SWAN). [S.l.: s.n.], 2015. p. 5–8.

[6] Kim S.; Zimmermann T.; Nagappan N. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In:Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks. Washington, DC, USA: IEEE Computer Society, 2011. (DSN '11), p. 486–493. ISBN 978-1-4244-9232-9.

[7] Dhaliwal T.; Khomh F.; Zou Y. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In:Proceedings of the 2011 27th IEEE International Conference on Software Maintenance. Washington, DC, USA: IEEE Computer Society, 2011. (ICSM '11), p. 333–342. ISBN 978-1-4577-0663-9

[8] Sinha S. et al. Fault localization and repair for java runtime exceptions. In:Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. New York,NY, USA: ACM, 2009. (ISSTA '09), p. 153–164. ISBN 978-1-60558-338-9.

[9] Wang S.; Khomh F.; Zou Y. Improving bug localization using correlations in crashreports. In:2013 10th Working Conference on Mining Software Repositories (MSR). [S.l.:s.n.], 2013. p. 247–256. ISSN 2160-1852.

[10] Wu R. et al. Crashlocator: Locating crashing faults based on crash stacks. In:Proceedings of the 2014 International Symposium on Software Testing and Analysis. New York, NY,USA: ACM, 2014. (ISSTA 2014), p. 204–214. ISBN 978-1-4503-2645-2.

[11] Cabral B., Marques P. (2007) Exception Handling: A Field Study in Java and .NET. In: Ernst E. (eds) ECOOP 2007 – Object-Oriented Programming. ECOOP 2007. Lecture Notes in Computer Science, vol 4609. Springer, Berlin, Heidelberg

[12] Gosling J, Joy B, Steele G. The Java Language Specification (The Java Series). Addison-Wesley: Reading, MA, 1997.

[13] Arnold K., Gosling J., Holmes D., The Java Programming Language, Fourth Edition, Addison-Wesley Professional, 2005.

[14] Coelho R. et al. Unveiling exception handling bug hazards in android based on github and google code issues. In:Proceedings of the 12th Working Conference on Mining Software Repositories. Piscataway, NJ, USA: IEEE Press, 2015. (MSR '15), p. 134–145.

[15] Lee, S. , Yang, B. and Moon, S. (2004), Efficient Java exception handling in just-in-time compilation. Softw: Pract. Exper., 34: 1463-1480. doi:10.1002/spe.622

[16] Basili V. R.; Caldiera G.; Rombach D. The goal question metric approach. Encyclopedia of Software Engineering, v. 1, 01 1994.

[17] PMD: An extensible cross-language static code analyzer. 2018. https://pmd.github.io/ [June 2018].

[18] Hovemeyer D.; Pugh W. Finding bugs is easy. SIGPLAN Not., ACM, NewYork, NY, USA, v. 39, n. 12, p. 92–106, dez. 2004. ISSN 0362-1340.

[19] Sonarsource. SonarLint. 2019. https://www.sonarlint.org/ [January 2019].