de Sousa *et al. Journal of the Brazilian Computer Society* (2020) 26:1
https://doi.org/10.1186/s13173-019-0095-5

Journal of the Brazilian Computer Society

RESEARCH Open Access

Studying the evolution of exception handling anti-patterns in a long-lived large-scale project

Dêmora B C de Sousa[1], Paulo Henrique M. Maia[2], Lincoln S Rocha[1] and Windson Viana[1*]

Abstract

Exception handling is a well-known technique used to improve software robustness. However, recent studies report that developers typically neglect exception handling (mostly novice ones). We believe the quality of exception handling code in a software project is directly affected (i) by the absence, or lack of awareness, of an explicit exception handling policy and guidelines and (ii) by a silent rising of exception handling anti-patterns. In this paper, we investigate this phenomenon in a case study of a long-lived large-scale Java Web system in a Public Education Institution, trying to better understand the relationship between (i) and (ii), and the impact of developers' turnover, skills, and guidance in (ii). Our case study takes into account the technical and human aspects. As a first step, we surveyed 21 developers regarding their perception of exception handling in the system's institution. Next, we analysed the evolution of exception handling anti-patterns across 15 releases of the target system. We conducted a semi-structured interview with three senior software engineers, representatives of the development team, to present partial results of the case and raise possible causes for the found problems. The interviewed professionals and a second analysis of the code identified the high team turnover as the source of this phenomenon, since the public procurement process for hiring new developers has mostly attracted novice ones. These findings suggest that the absence of an explicit exception handling policy impacts negatively in the developers' perception and implementation of exception handling. Furthermore, the absence of such policy has been leading developers to replicate existing anti-patterns and spread them through new features added during system evolution. We also observed that most developers have low skills regarding exception handling in general and low knowledge regarding the design and implementation of exception handling in the system. The system maintainer now has a diagnosis of the major causes of the quality problems in the exception handling code and was able to lead the required measures to repair this technical debt.

Keywords: Case study, Exception handling, Exception handling anti-patterns

Introduction

Exception handling (EH) is a forward error recovery technique used to improve software robustness by providing means to structure fault-tolerance activities into the software source code [1, 2]. Exception model abnormal situations, detected at run time, which disrupt the normal control flow of a programme. When an exception is detected, the normal control flow is interrupted and the exceptional control flow begins. From this point forward, the EH mechanism takes control of a programme execution and starts a search for a proper handler to deal with such abnormal situation [3]. Widely adopted programming languages, such as Java, C#, and Python, provide constructs devoted to structure the exceptional control flow, specifying in the source code where and how exceptions can be raised, propagated, and handled [4].

However, despite its importance, EH is commonly neglected by developers. It is claimed as the least understood, documented, and tested part of a software system [5–8]. Recent studies have investigated the relationship between EH code and software maintainability [4], evolvability [9], architectural erosion [10], robustness [11], bug appearance [12], and defect-proneness [13]. They provide pieces of evidence that the quality of EH code may impact

*Correspondence: windson@virtual.ufc.br
[1]Federal University of Ceará, Av. Humberto Monte, 60440-593, Fortaleza, Brazil
Full list of author information is available at the end of the article

on the overall software quality. Moreover, researchers pointed out EH anti-patterns (e.g., *Catch and Do Nothing* and *Destructive Wrapping*) as a source of software failures, named exception handling bugs [12, 14, 15].

Based on previous studies [5, 9, 10, 15–17], we are convinced that the quality of EH code produced in a software project is directly affected by two aspects: an absence (or lack of awareness) of an explicit EH policy and guidelines, and a silent raising of EH anti-patterns. The former may lead developers to decide on their own, in an ad hoc way, how to design and implement the EH code. Therefore, this can cause, for instance, an architectural erosion problem, with respect to EH [10, 16]. Without a systematic code review process, the appearance of EH anti-patterns will only be avoided by the developers' skills, knowledge, and consciousness concerning the importance of EH code to the overall software quality [5].

Thus, to gather evidence that corroborates our belief, we conducted a case study on a long-lived large-scale Java Web system of a Public Education Institution (hereafter called institution) in a developing country, which has about 64,3471 lines of code, 4590 classes, and more than 46 people involved in its maintenance and evolution. We aimed at answering the following research questions:

- **RQ1**. *What perceptions does the development team of a large-scale system have about EH*?
- **RQ2**. *What and to what extent EH anti-patterns can be found in a large-scale system and how they evolve over time*?
- **RQ3**. *What factors contribute to the dissemination of EH anti-patterns in a large-scale system*?

In our previous study of this Java Web System [18], we examined the research questions RQ1 and RQ2. To answer RQ1, we surveyed the project team members responsible for designing and developing the exceptional flows of the analysed target system. In this online survey, we collected information concerning the team's skills, knowledge, and perception about the EH design, implementation, and importance in the institution and in the project itself. Next, to answer RQ2, we perform a source code analysis in 15 releases of the target system to identify the occurrence, types, and the number of EH anti-patterns added in each release.

In this paper, we went a step forward to answer the research question RQ3. The interview results revealed situations that could be the cause of XSA's exception handling problems. Hence, we decided to investigate them deeply by adding a third unit of analysis in our case study. The key points to be investigated were (i) the development team turnover, (ii) the inexperience of novice developers, and (iii) the developers' awareness of EH anti-pattern insertion. For instance, we dive into the understanding of the team turnover impact in the EH quality by re-analyzing the XSA code and applying a code statistic analysis. In addition, we analysed the programmers' knowledge concerning the EH anti-patterns further with the aim to investigate whether they were inserting or removing EH anti-patterns because they were unaware of them. For that, we applied a new online instrument to see if developers of the XSA system know the most recurring anti-patterns in the source code they implement. Finally, we improved the related work section and almost all the paper's figures and graphics.

To answer RQ3, we performed a set of actions. Firstly, we carried out a semi-structured interview with a project committee composed of the three most expert software architects. In the occasion, we presented the results of R1 and R2 to them. Secondly, we tracked down the changes made by the developers in the source code to analyse which group of professionals (experienced and novice ones) added and removed more anti-patterns in the system's source code. Finally, we applied an online quiz aiming at evaluating developers' capability to recognise EH anti-patterns from code snippets extracted from the studied system.

Our findings indicated that the majority of developers of the target institution believes in the importance of exception handling. However, they face obstacles during development activities, such as lack of documentation, lack of EH policy, and the absence of automated tools to detect bad EH practices. Several anti-patterns were detected in the system code. *Catch Generic*, *Generic Throws*, *Destructive Wrapping*, and *Catch and Do Nothing* EH anti-patterns are responsible for the largest number of violations. *Catch Generic* EH anti-pattern corresponds to more than 70% of all found violations, unlike the results presented in other studies [12, 15, 19]. The interviewed architects pointed the absence of a team dedicated to software quality inspection in the institution, the high team turnover, the inexperience of developers to work with large systems, and the replication of anti-patterns as the source of this phenomenon. The results from the code change analysis and the online quiz provide evidence supporting the claim of the developer's experience impacts in the inclusion and dissemination of EH anti-patterns in the target system. In short, we have observed that inexperienced developers are more prone to insert EH anti-patterns and less able to recognise them.

The remainder of this paper is organised as follows. In the "Background" section, we present the theoretical background. Next, in the "Research methodology" section, we describe the research methodology followed in this study. The "Unit of analysis 1: developer's perception about EH" section, "Unit of analysis 2: EH code analysis" section, and "Unit of analysis 3: cause and effect analysis" section details the unities of analysis investigated in the case study. The "Overall discussion" section

presents the overall discussions and the "Threats to validity" section the validity threats. The "Related work" section is devoted to related works, and the "Final remarks and future work" section concludes the paper.

Background

Java exception handling

In Java programming language, "an exception is an event, which occurs during the execution of a programme, which disrupts the normal flow of the programme's instructions" [20]. When an error occurs inside a method, an exception is raised. In Java, the raising of an exception is called *throwing*. Exceptions are represented as objects following a proper class hierarchy and can be divided into two categories: checked and unchecked. Checked exceptions are all exceptions that inherit, directly or indirectly from `Exception` class, except those ones that inherit, directly or indirectly, from `Error` or `RuntimeException` classes, named unchecked ones. Checked exceptions represent exceptional conditions that a robust software should recover from. Unchecked exceptions represent an internal (`RuntimeException`) or an external (`Error`) exceptional conditions that a software usually cannot anticipate or recover from. In Java, only handling of checked exceptions is mandatory, while the handling of unchecked exceptions is not.

When an exception is raised, the execution flow is interrupted and deviated to a specific point where the exceptional condition is handled. In Java, exceptions can be raised using the `throw` statement, signalled using the `throws` statement, and handled in the `try-catch-finally` blocks. The "`throw new E()`" statement is an example of *throwing* the exception `E`. The "`public void m() throws E,T`" shows how `throws` is used in the method declaration to indicate the signalling of exceptions `E` and `T`.

The `try` block is used to enclose the method calls that might throw an exception. If an exception occurs within the `try` block, that exception is handled by an exception handler associated with it. Handlers are associated to a `try` block by putting a `catch` block after it. A `try` block can be associated with multiples `catch` blocks. Each `catch` block catches a specific exception type and encloses the exception handler code. The `finally` block is optional, but whether declared always executes when the `try` block finishes, even if an exception occurs. The coding of cleanup actions within the `finally` block is recognised as a good practice.

Exception handling anti-patterns

Several studies have identified and characterised EH code anomalies, referring to them using their own nomenclature such as bug patterns [21], inappropriate coding patterns [22], bad smells [23, 24], and anti-patterns [14].

Barbosa et al. [19] identify software failures caused by such kind of code anomalies. They named them as exception faults or exception handling faults, since they are related to defining, throwing, propagating, and documenting exceptions. Ebert et al. [12] propose a taxonomy to classify this kind of faults, which they called exception handling bugs. They improve the definition of Barbosa et al., stating that EH bugs are those that occur when an exception is defined, thrown, propagated, handled, or documented in the cleaning action of a protected region where it is launched; when it should have been thrown or dealt with, but it has not.

Yuan et al. [21] found in their study that approximately 92% of catastrophic failures comes from non-fatal errors signalled by the software itself. In addition, 35% of such failures were caused by empty handlers, wrong actions in generic handlers, and handlers with comments suggesting the need for further implementation and correction (e.g., "`FIXME`" and "`TODO`").

In this study, we adopt the term EH anti-patterns to refer to the EH code anomalies. However, it is important to notice that there is no sufficient evidence on a causal relationship between the presence of such anti-patterns in the source code and software failure occurrence. In fact, the presence of anti-pattern can lead to maintainability problems, such as architectural degradation and technical debt. In that sense, Correa et al. [25] says that an anti-pattern describes a solution to a recurrent problem that generates negative consequences to a project. An anti-pattern can be a result of either not working a better solution or using a good solution in a wrong context.

Padua and Shan [15] identified the prevalence of EH anti-patterns in open source projects. They found that some of these problems have more occurrence than others; however, all were found in at least 3 of the 9 Java projects taken into account in the study. These anti-patterns were considered starting points for the inspection carried out in our study. Table 1 gives a brief description of some of these EH anti-patterns.

Existing static code analysis tools can detect these anti-patterns. Some of these tools are already available on the market and widely used. Researchers have also developed other tools with more specific purposes. For example, the PMD[1] tool is a source code analyser available for several languages. It detects possible bugs, unused code, unnecessary code duplication, and complexities. Also, it can identify the presence of 15 exception handling anti-patterns. In our case study, we analysed the presence of those 15 EH anti-patterns. We used three criteria to choose them: the risks that they bring to the system, the frequency that they usually occur, and the fact that they should be detectable by a tool. Moreover, even with

[1] https://pmd.github.io/

Table 1 Exception handling anti-patterns [15]

Anti-pattern	Meaning
Unhandled Exceptions	Exception handling does not capture all possible exceptions thrown in a try block.
Unreachable Handler	Exception handling code will never run.
Catch Generic	Exception handling captures many low-level exceptions by setting a high-level exception.
Destructive Wrapping	The developer propagates an exception as if it were another one. It causes loss of information regarding the original exception.
Catch and Do Nothing	Catch block code is empty.
Dummy Handler	The Exception handling code has no actions to recover errors.
Ignoring Interrupted Exception	Caught exception is ignored in the catch block.
Throw withing Finally	The code throws an exception within a finally block.
Generic Throw	The code throws a generic exception.

relatively simple anti-patterns, as we reported above, literature results showed that they might impact the system execution and code maintenance.

Research methodology

Main objectives

Our case study has the intention of evaluating the quality of the EH code of a long-lived large-scale Java Web system, which is maintained by a team of 46 professionals. The system owner institution provides insufficient documentation on the EH of its system. Based on previous studies, we know the absence of an EH policy can generate doubts during development activities and, consequently, interfere with the system's ability to recover from faults. The main goal of our case study is to investigate and confirm, in an exploratory research, whether or not this situation actually happens in practice. To do that, it is essential to comprehend the way Web developers understand their own EH code and to confront such code against standard EH code quality rules.

Case study organisation

The studies of Ebert et al. [12], Shah et al. [5], and Barbosa et al. [19] have served as a source of inspiration to design our case study. It followed the case study methodology described by Runeson et al. [26]. We split the study into three units of analysis according to the research questions presented in the "Introduction" section.

Figure 1 depicts these units, showing the origin of the collected data and listing some of the acquisition methods employed in the study. The first unit captures developers' perceptions concerning EH, how they deal with it in daily activities, and how EH is supported by the institution (e.g., proper checking tools and available documentation). Through an online survey, we interviewed professionals who deal directly with the planning and development of the exceptional flows of the target system.

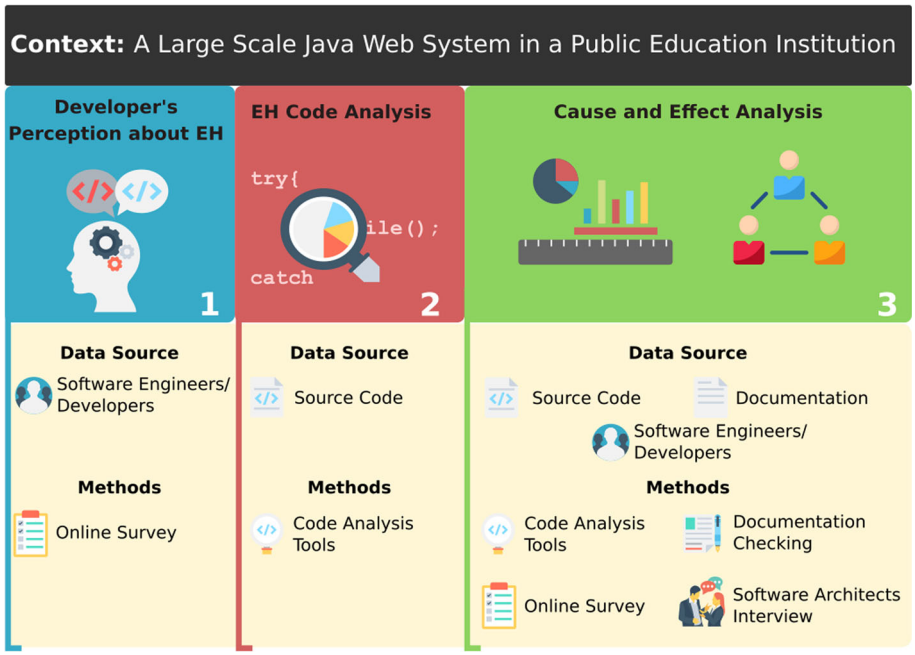

Fig. 1 Case study structure: exception handling in the XSA system

The second unit of analysis was designed to investigate how the EH is used in practice. Thus, we used a set of tools to inspect the code of the system under analysis. The third unit aims at answering the questions that arose with the analysis of the data collected in previous units. We tried to establish a cause-effect relationship from the triangulation of data from the code and the development team knowledge. These answers were helpful to better understand the case study context, making clearer the relationship between exception handling quality and development team knowledge, source code, system documentation, and administrative issues (e.g., developer team turnover).

Subject system

The target system is a Web-based system, which has more than 20,000 active users in a Public Education Institution. It is a management system implemented in JEE (Java Enterprise Edition) using a set of frameworks, such as JSF, Struts, and Hibernate. The institution acquired it in 2010. It was then adapted, expanded, and, nowadays, a division of the institution maintains it. They have at least 46 professionals involved in the process of system maintenance and evolution (e.g., coding and analysis). Several segments of the institution use the studied software, since it is vital to the institution activities. For security and confidentiality purposes, the system name will remain anonymous throughout the paper, being called only as the XSA system. The system encompasses a broad scope of the institution needs (e.g., resource management, people organisation, and academic data administration).

The XSA system has four layers: presentation, application, domain/business, and infrastructure/data access. This structure separates responsibilities and also groups modules with similar purposes. The software division splits the XSA system maintenance into 11 projects. The developers work on teams dedicated to maintain and develop modules on each project. Software analysts lead the development teams and mediate the communication with customers. The development process of each group is independent. Thus, they can use agile methodologies (most used), Waterfall model, or choose not to use any of them (ad hoc way).

The XSA system has brief documentation about EH. This documentation provides the following information:

- The hierarchy of custom system exceptions;
- How to use utility methods to notify the development team and the final user when exceptional events occur;
- How to handle exceptions at the business layer.

However, there is no guidance to the developer about where and how to handle Java language exceptions and other existing custom exceptions. Therefore, it is inferred that exceptions can be thrown and handled by any module and flow between layers without restrictions.

In the case study, we also analysed the evolution of anti-patterns throughout 15 releases of XSA system. The first release, which dates from 2010, represents the system in its initial stage, without changes made by the target institution of this study. In fact, the institution purchased this initial version from another company. From 2011, when the system customisation began, we consider semi-annual releases, until the end of 2017 (the most recent release at the time this paper was written). Table 2 shows data from some of these releases.

We decided to investigate the presence of EH anti-patterns in XSA due to frequent complaints from its users, which received messages from Unhandled Exceptions during system execution. Besides, some members of the XSA development team had already reported to us the desire to investigate the exception handling of the system. Monitoring the use of the system for 25 days, we noticed the presence in the system log of 2927 exceptions. From these exceptions, the XSA code did not catch 2634 exceptions, on average, 106 per day. The ten exceptions with the highest number of occurrences represented 95.22% of the total recorded. Eight of them were of the `RuntimeException` type. For the most part, these exceptions correspond to general exceptions, programming errors related to improper object access, database query errors, and connection problems. The most expressive in quantity was the `NullPointerException`. Only a tiny percentage of the exceptions (1%) corresponded to exceptions from the system hierarchy. So, our study could give the first insights into this problem.

We identified the presence of anti-patterns in each class, package, module, layer, and project of the 15 XSA releases. Also, we analysed the contribution of each developer in the insertion and removal of anti-patterns in two XSA releases (2015 and 2017). For privacy reasons, some data are not public and are not available in this paper. We chose

Table 2 Evolution of XSA system metrics

Data	2010	2011	2014	2017
Total lines of code	455819	476645	532685	650261
Packages	258	266	275	326
Source folders	28	29	30	35
Number of classes	3260	3368	3548	4625
Number of methods	47019	48711	53405	64307
Number of interfaces	66	70	77	115
Catch block lines of code	5874	6098	6943	8375
Number of handlings	3407	3545	4024	4835

to present only aggregated information or anonymized data in the paper, trying to preserve the comprehension of the studied phenomenon.

The XSA system is a customisation of a Web system used in 58 other public institutions in our country. In total, the customisations of the XSA system have a base of over 500 thousand active users. The case study presented in this paper focuses on specific customisation adopted at a particular institution. However, the results of this research may also motivate to investigate if a similar problem does not occur in the other 58 customisations. The analysis of the presence of EH anti-patterns in XSA may also interest developers and EH researchers working on Java Web systems with an analogous context of the XSA case study (e.g., Web teams that use JSF and Struts, public organisations with high turnover). Actually, some researchers have already pointed out the presence of EH anti-patterns and their relationship with bugs in Java Web systems, occurring to a lesser or greater extension [12, 15, 19]. Our case study goes further by investigating the quantitative and qualitative aspects of the EH anti-patterns' presence and their evolution in the XSA code, giving other perspectives of analysis.

Unit of analysis 1: developer's perception about EH

The first unit seeks to understand the human aspect of exception handling involved in XSA. We interviewed, from an online survey, the people who deal directly with the planning and development of the XSA's EH flows. This unit was intended to gather information about the developers/analysts perception regarding:

- The importance degree of EH in their point of view;
- Their impression concerning the EH code quality of XSA system;
- Their self-declared knowledge on EH coding; and
- Their satisfaction with their skills and knowledge of EH.

Subjects

The target audience for this unit of analysis was the software engineers of XSA system: both programmers and analysts of the institution. Those people have distinct roles in the system development cycle, such as system code developer, requirement analyst, database administrator, tester, and project manager. They perform one or more of those roles during maintenance cycles. The software engineers deal with the development of Java Web systems (i.e., the XSA system and other smaller systems). Java is the predominant technology in the institution. The project management tool Redmine[2] assists their development cycles.

Materials and methods

We elaborated an online survey based on Ebert et al. [12] and Shah et al. [5] works. Before applying it, we validated the online form with three experienced software architects. The online survey[3] contains:

- 2 questions that address the respondent's experience on software development;
- 26 Likert scale survey questions related to exception handling (e.g., quality of EH code in XSA, its documentation);
- 2 questions about general concepts of exception handling;
- 2 open questions regarding the use and perceptions of exception handling; and
- 1 statement for the instrument evaluation.

Procedure

XSA developers received an email with our online survey. In that email, we presented our research goals and guaranteed data anonymity and its exclusive use for academic purposes. For interpreting the results, we summarised the per cent of respondents who agreed or disagreed to the questionnaire's items, discarding the neutral answers.

Results

The online survey obtained 21 responses (53.84% of the professionals involved in system development). A significant part of them started working in the institution in the last 3 years (66.7%). Most respondents have been working with Web development for less than 6 years, about 61.9% of them. The rest of them is more expert (over 7 years of Web development experience). However, 61.9% of them consider themselves experts with respect to the software development in general. 52.3% of the professionals declared they do not feel confident with the tools and the programming languages used in the institution.

52.4% of the respondents reported they do not usually document code elements related to EH. By separating developers into two groups (novices and experts), we realise that none of the novices ($<$ 3 years of experience) claims to document the exceptional handling code. In contrast, 43.75% of experts reported documenting the EH code. Practically, all developers agree that the institution does not have clear EH guidelines or policies for its systems. 85.7% of them claim they have observed the impact of EH errors on end user activities. Also, 76.2% of them agree that users have already reported EH errors to the institution.

Only 14.3% of developers say that they use software quality checking tools (e.g., for checking the presence of code smells). None of the novices use these tools.

[2] https://www.redmine.org/

[3] The online survey instrument is available at http://twixar.me/qjGn.

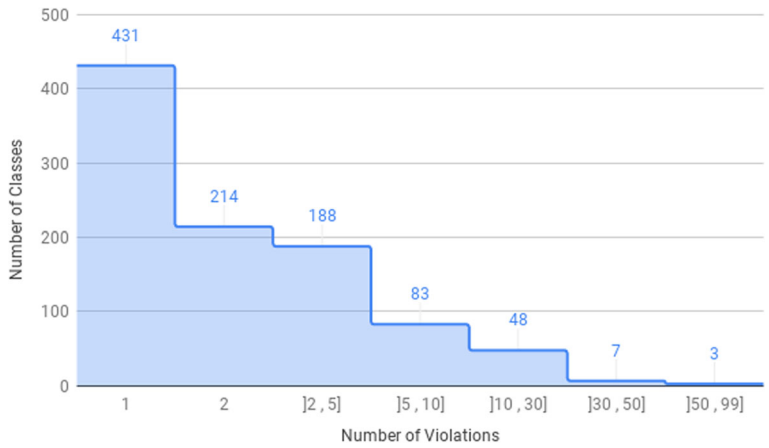

Fig. 2 Frequency of Anti-Patterns per classes in 2017.2 release

One interpretation is the reported errors may be a consequence of the low adherence of these checking tools throughout the development cycle. Only 28.57% of the respondents are satisfied with the way they deal with EH. Finally, everyone believes that EH in XSA must be improved and they pointed out the need to establish EH institutional policies.

Unit of analysis 2: EH code analysis

The second unit of analysis focuses on the technical aspect of EH in XSA. We used a set of tools (e.g., PMD and JavaParser) to extract information concerning the EH from XSA source code. In previous work, researchers evaluated EH by considering distinct aspects of software quality, highlighting those relevant to the evaluation of the EH in a system, such as anti-patterns, bugs, and general principles of EH. In this unit of analysis, our choice was to seek the presence of anti-patterns in 15 releases of the XSA system. We also analysed the evolution of code metrics related to XSA's EH implementation.

Materials and methods

For our analysis of anti-patterns evolution, we extracted metrics and searched for the presence of *anti-patterns*. For this, we developed a project in the Java language, named VbR (*Violations by Repository*). That application uses the RepoDriller[4] tool, which provides access to source code for versions through the XSA Git repository. Test codes were not considered in the analysis.

For anti-pattern identification, we adopted the PMD tool since it detects a higher amount of EH anti-patterns listed in Table 1. We used PMD with its default rulesets for EH, without any additional customisation. VbR executes PMD scripts to find the presence of them. Other metrics, such as the number of catch blocks and throwing, were extracted using the JavaParser[5] tool. VbR also generates output files in CSV (Comma-Separated Values) to facilitate the analysis and plotting of charts.

The term *violation* will be used henceforward to indicate the occurrence of an EH anti-pattern, since the tool detects them by using code rules that we defined.

Results

Current release

In the current version, we found 3423 occurrences of anti-patterns. These violations affect 974 classes of the system (21%). Figure 2 shows the distribution of these occurrences in classes files that have at least one anti-pattern. Sixty-seven per cent of the affected classes have a maximum of 2 violations. However, XSA has also some outliers, such as a class that has 99 anti-patterns. This class has over 4000 LoCs and 100 catch blocs.

Figure 3 shows the distribution of affected files with anti-patterns in the 2017 release. The view layer and the data access layer have the highest percentages of code files with violations, with quantities greater than 30% and 35%, respectively. The view layer has an essential role to avoid the presentation of internal problems of the system to the end users. Defects on the EH in the view layer may affect this role negatively.

Table 3 presents the results for each EH anti-pattern in the 2017 release. The anti-pattern related to catching generic exceptions (namely *Catch Generic*) is the one that has more violations in the XSA system. We considered only Java language-defined generic exceptions, such as `Exception`, `RuntimeException`, and `NullPointerException`. The results shows that more than 70% of violations for this anti-pattern and 50.51% of the EH blocks matches with this anti-pattern. It is important to mention that the occurrence of this anti-

[4] https://github.com/mauricioaniche/repodriller

[5] http://javaparser.org/

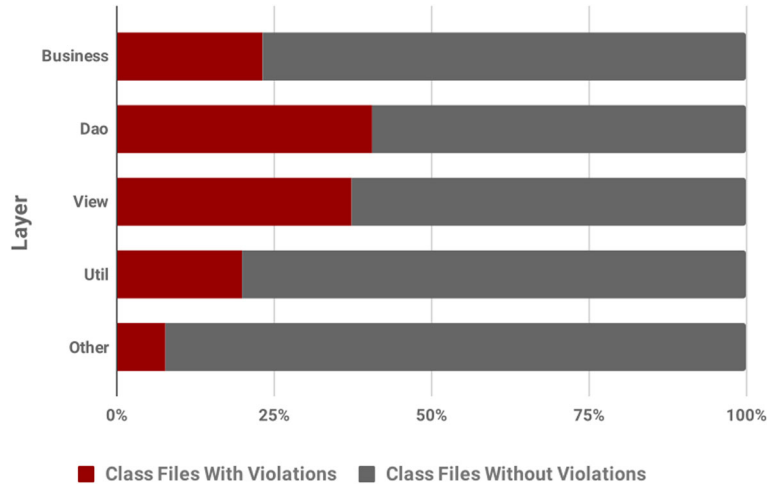

Fig. 3 Anti-patterns violations for each layer in 2017.2 release

pattern can lead to serious robustness problems such as the swallowing of relevant exceptions and the implementation of inefficient handling mechanisms [24].

Catch and do Nothing is the fourth anti-pattern with the highest number of violations. It is a recognised maintenance risk since it turns the code debugging difficult and causes loss of original error information [12]. However, in comparison with the most violated one, it is numerically much lower, since it affects only 2.38% of the EH implemented in XSA. The Java code snippets (Table 4) are examples of anti-patterns found in the XSA during the static code analysis.

Table 3 Anti-patterns find in the 2017.2 release

Anti-Pattern	Violations	Affected Handlers or Throwing
PMD - Anti-Patterns		
Catch Generic	2440	50,51%
Throws Generic	495	2,60%
Destructive Wrapping	215	4,45%
Catch and Do Nothing	115	2,38%
Throw within Finally	41	0,85%
Wrong Exception Thrown	17	0,35%
Relying on getCause()	5	0,10%
Dummy Handler	4	0,08%
Error in the Definition of Exception Class	0	0,00%
PMD - Other Types of Violations		
AvoidRethrowingException	43	0,89%
EmptyFinallyBlock	18	0,37%
AvoidThrowing NewInstanceOfSameException	14	0,29%
EmptyTryBlock	2	0,04%
ExceptionAsFlowControl	0	0,00%
AvoidCatchingThrowable	0	0,00%

Anti-pattern evolution

The XSA system had an increase in its number of classes and LoC (Lines of Code) between 2010 and 2017 (from 455,819 to 650,261). Table 2 shows that XSA is in

Table 4 Examples EH anti-patterns found

```
//Example of Destructive Wrapping
try {
if(con != null) con.close();
} catch (SQLException e) {
throw new DataException(e.getMessage());
}
//Example of Catch Generic
try {
cal = CalendarHelper.getCalendar(getCurrentUser());
} catch (Exception e) {
e.printStackTrace();
defaultHandle(e);
}
//Example of Throws Generic
public Object method(HttpServletRequest req)
throws Exception {
GenericDataAccess dao = getGeneric();
Object obj = getCommandClass().newInstance();
// . . . .
return obj;
}
```

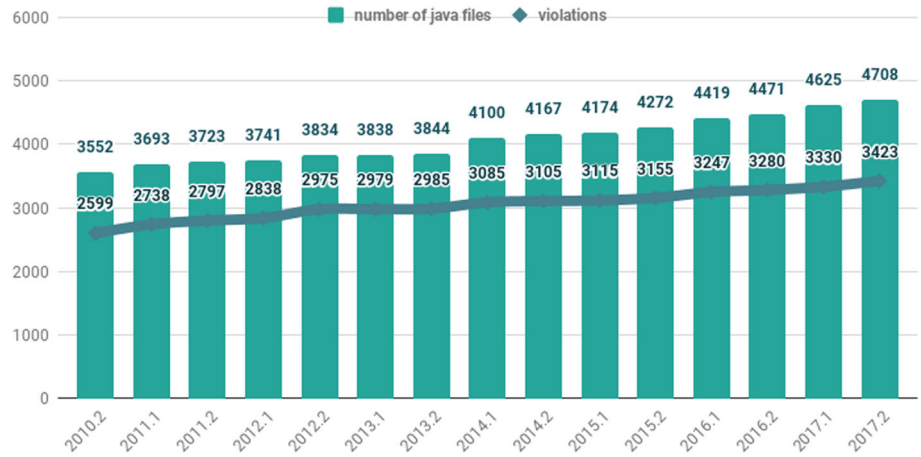

Fig. 4 Evolution of XSA system anti-patterns

constant maintenance and customisation, which explains its growth. Regarding EH, we observed the following growth percentages: 41.91% for implemented handling blocks and 38.64% for exception throwing. The source code expansion also included a proliferation of EH anti-patterns.

It is important to notice the number of violations can lead to misinterpretation of the phenomenon since the size of XSA has increased considerably. For instance, Fig. 4 shows the XSA evolution comparing the number of Java files and the violation occurrences. Their growth was quite similar, 31.70% for the number of violations and 32.54% for the number of Java files (i.e., `.java` files). In this way, we study some ratios between the number of rules violated and code metrics, see Eqs. (1) and (2), where ξ is an anti-pattern and Υ is the XSA source code.

$$\texttt{VtoH}(\Upsilon,\xi) = \frac{\#violationsIn(\xi,\Upsilon)}{\#handlersIn(\Upsilon)} \tag{1}$$

$$\texttt{VtoT}(\Upsilon,\xi) = \frac{\#violationsIn(\xi,\Upsilon)}{\#throwsIn(\Upsilon)} \tag{2}$$

`VtoH` (1) is the ratio of the number of violations of an anti-pattern per number of handlers implemented in XSA. Similarly, `VtoT` (2) ratio is calculated by dividing the number of violations per number of throws in XSA.

Figure 5 details the evolution of the four most found anti-patterns. *Catch and do Nothing* anti-pattern had 80 violations in the 2010 release and 115 in the 2017 counterpart, representing an increase of 43.75%. The *Destructive Wrapping* anti-pattern grew 115% between the 2010 and 2017 releases, having the number of violations jumped from 100 up to 215 in 2017.

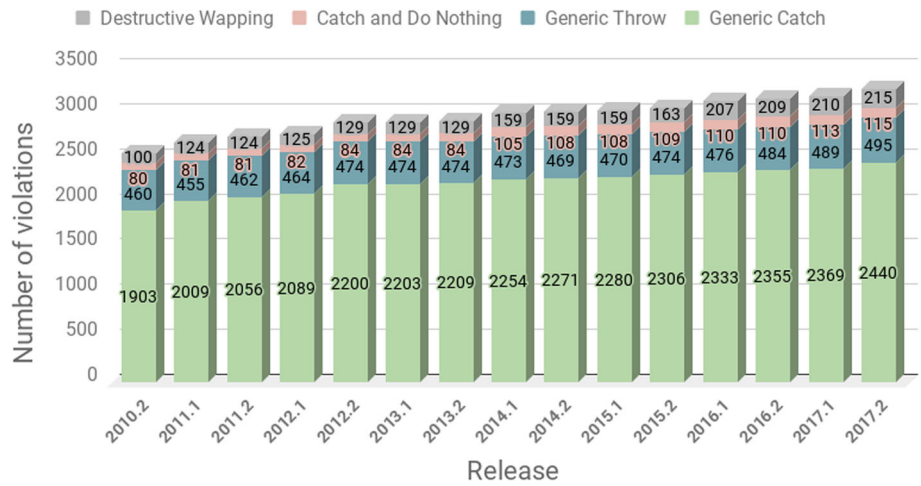

Fig. 5 Evolution of the four most found anti-patterns. **a** Signallings affected by Generic Throw. **b** Catch blocks affected by Generic Catch. **c** Catch blocks affected by Catch and Do Nothing. **d** Catch blocks affected by Destructive Wrapping

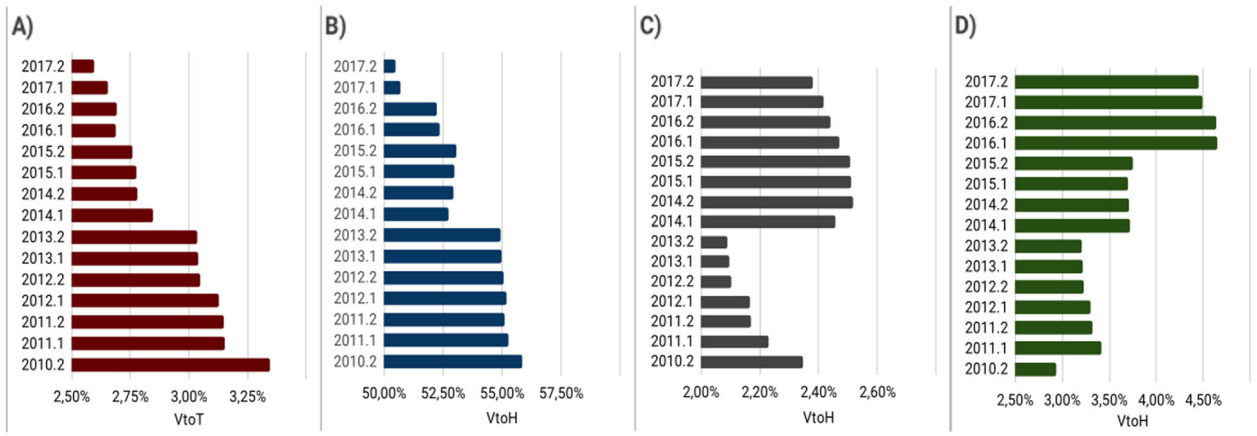

Fig. 6 Anti-patterns evolution in the XSA releases. **a** Signalings affected by generic throw. **b** Catch blocks affected by generic catch. **c** Catch blocks affected by catch and do nothing. **d** Catch blocks affected by destructive wapping

Figure 6 shows the evolution of four ratios. We observed an increase in the `VtoH` ratio regarding the anti-patterns *Catch and do Nothing* and *Destructive Wrapping*, specially between 2014 and 2017 and a decrease in the other two anti-patterns. The decrease of the most recurrent anti-patterns on the system (*Catch Generic* and *Throws Generic*) is a positive factor. A possible cause is the adoption of practices, such as implementation of specialised handlers and the signalling of specific exceptions, which contributed to an adequate development of error recovery activity. However, *Catch Generic* is still present in 50% of the handlers. Also, other anti-patterns continue to grow in XSA (even its ratios).

Figure 7b exhibits an absolute increase in the number of anti-patterns detected over the years for each XSA layer. This information reveals that developers have not refactored the EH code sufficiently, which indicates the existence of a technical debt. The view layer remains the one with more violations, followed by the data access, and business layers. Grouping all the violations in the 15 versions, we obtain that the generic exception handling and the launching of generic exceptions correspond to, respectively, 61.86% (12,279) and 32.42% (6437) of the violations in the view layer. In the data access layer, the generic exception handling accounts for 93.50% (15,929) of the violations.

Figure 7a shows the ratio of the number of violations to the number of Java files for each layer. There is no steady growth in these rates for all layers. In fact, the data layer demonstrates a decrease of 12.61% in this ratio (from 2.95 to 2.58 in the last release). We noticed a different behaviour for the other layers, since there was an increase in the rate: view (3.02%), business (35.11%), and util (67%).

Unit of analysis 3: cause and effect analysis

In the third unit, we started trying to attest a cause-effect relationship using data triangulation from the code analysis and the development team knowledge. Firstly, we interviewed a project committee composed of three experienced software engineers. From the insights and comments of this interview, we improved our code analysis by seeking pieces of evidence of the causes pointed out by the project committee (e.g., team turnover). We present this process and its results in this section.

Semi-structured interview

The purpose of this interview was to provide a better understanding of the case study results. Also, we aim at

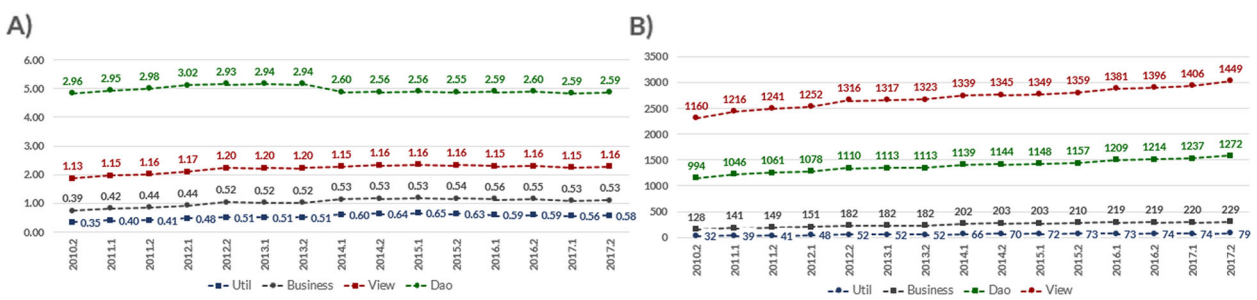

Fig. 7 Evolution of anti-pattern violations per layer. **a** Evolution of #violations per #Java file (PMD) and **b** #violations in each layer (PMD)

identifying motivation causes and the context that produced these results in the XSA system.

Subjects

We conducted a semi-structured interview with a project committee composed of the three most experienced software architects of XSA. The interviewees have taken several roles since the XSA system was deployed in the institution, such as software analyst, project manager, and development team leader.

Materials and methods

The interview was organised and conducted by three researchers, all co-authors of this paper. The interview followed a semi-structured format (i.e., an interview guide[6]). We analysed the interview results following six steps proposed by Creswell [27]. We transcribed the interview, structuring it according to the interview guide. We re-read the transcript and marked the text with codes that refer to the content of the sentences or paragraphs (e.g., exception handling policy, developer experience, and development cycle). Then, we analysed the created codes and grouped them, summing up a total of 27 codes that were gathered in 6 categories, which were previously proposed by Creswell [27].

Procedure

The interview was a unique 2-h meeting with the three subjects and the researchers. We recorded the interview and transcribed it for further analysis. The interview had four phases:

- Phase 1: Presentation about the research, its objectives, and structure;
- Phase 2: Presentation and discussion of results obtained from the online questionnaire (unit of analysis 1);
- Phase 3: Presentation and discussion of metrics collection results and presence of anti-patterns in the system (unit of analysis 2);
- Phase 4: Suggestions and evaluation of interview's format and content.

One researcher conducted the semi-structured interview. She showed the study data and presented questions about this data to the project committee. These questions were about particular practices and experiences of that development environment, such as the staff routine, the materials used, and the developers experience. The other researchers took notes, clarified doubts, and asked unplanned questions to the interviewees.

[6] The interview guide is available at http://twixar.me/KqGn

Results

The interview played a fundamental role in the understanding of the motivating context of the case study findings. Among them, participants cited the lack of documentation as one of the major problems faced by developers. They state this issue is also causing the presence of EH anti-patterns in the system. Without it, a developer does not have an understanding of the architecture and the custom exceptions that are part of the XSA exceptional flow. Developers underuse these exceptions. Respondents believe that the team has no clear view on the use of this type of exception and in which situations they should apply them. Interviewees concluded that this lack of knowledge may be leading developers to neglect the exception handling code.

There are no policies for EH in the institution. Respondents pointed out the process of developing exceptional flows is seen as an informal and empirical one. This reactive behaviour is found when EH is classified as a low-priority activity [5]. Respondents recalled that other code policies and software processes have already been debated and adopted by the institution. They sought to improve the overall quality of the XSA system code, but EH has never been seen as a problem.

According to the interviewees, the *lack of experience of the novice developers* has a direct impact on the quality of the EH code. Frequently, newcomers have just finished their undergraduate and are in their first job. Therefore, they have no experience in the development of large-scale systems or with the EH for Java Web. The interviewees listed some behaviours adopted by this profile of developers, such as the following:

- Developers print the exception stack-trace on the system screen to show the occurrence of an exceptional situation. After, they do not complete the catch code since the main flow is correct running;
- Novices find more important displaying information about the exception than to seek appropriate handling;
- Developers replicate existing practices in code, even those that are not suitable, such as the Empty Catch Blocks or Catch Generic;
- These developers do not know what actions to implement in the catch block. They choose to throw general exceptions;
- Their implementation practices make it difficult to understand their code with nested treatments and, sometimes, without specific handling actions;
- Generally, they are more concerned with finishing the code to be executed. Therefore, they implement the mandatory handlers required by the Java language without thinking about the ideal recovery actions.

After presenting the EH anti-pattern concept, interviewees quickly identified their occurrence in the XSA system. Spontaneously, respondents listed the following issues: *Lack of Documentation*, *Destructive Wrapping*, *Catch Generic*, *Generic Exception Throwing*, *Dummy Handler*, *Empty Catch Blocks*, and *Unhandled Exceptions*. However, the number of violations found during our study surprised the interviewees. They expected stabilisation of these problems or modest growth. Respondents did not expect high number of these issues. Nevertheless, after a few minutes, one of the interviewees stated that the high value of violations should be expected. Since developers did not fixed them, it was reasonable that they would continue to be reproduced over the years.

Regarding the possible effects of the EH anti-patterns' presence, the interviewees mentioned the prevalence of *Destructive Wrapping* makes it difficult finding the original cause of the error. Therefore, the effort spent to find and fix some kinds of bugs increases. Useful information remains between the various rows of the log files. Developers need more effort to fetch the original error stacktrace, which impacts in solving the problem that causes it. This situation is stressed when developers write incomplete and useless information in the log files. Besides, the system does not have a clear policy about the Unhandled Exceptions, and therefore, the developers do not know when and what strategy should be employed. Sometimes, exceptions end up being passed on to the end user.

The interviewees connected the evolution of EH anti-patterns to the replication of bad practices already existing in the original code. Once the XSA documentation is incomplete and not up-to-date, developers are guided by the existing source code itself. They become, therefore, replicators of existing behaviours. Another possible cause pointed out is the process that drives the code importing during the implementation of new modules. They think this process is a possible conveyor of bad practices since it consists in the adaptation of features coming from an external environment.

The interview results reveal situations that could be the cause of XSA system exception handling problems. So, we decide to investigate them deeply for confirming their relationship with the results presented in unit 02. The key points to be investigated were as follows:

- Development team turnover;
- The inexperience of novice developers; and
- Developers' awareness of EH anti-pattern insertion.

The following subsections detail the study of these assumptions.

Turnover

Hypothesis: *The high turnover of developers makes it difficult for them to become proficient in the XSA code and, consequently, mastering EH code.*

Our goal was to gather information on team turnover and try to quantify its impact on XSA development.

Materials and methods

The hypothesis has the team turnover as a central issue in the EH anti-patterns' presence. Therefore, it was necessary to obtain data on hiring and leaving members of the development team. The system that manages the XSA human resources, holder of the turnover data, made it possible to understand how this phenomenon occurs. From there, the list of employees, with the date of admission and withdrawal, has been obtained since 2010.

Results

Figures 8 and 9 show some data concerning the team turnover confirming its strong presence in our case study. In 2011, the institution started the process of XSA purchase and adaptation. At that time, the team had 15 active professionals. The number of developers grew 73.33%

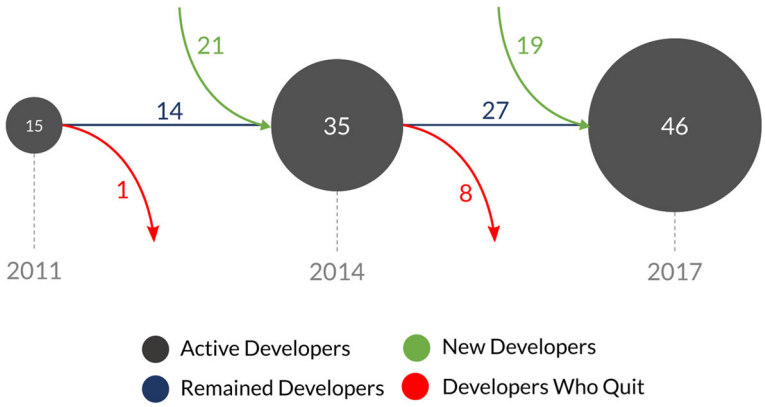

Fig. 8 Development team turnover per release

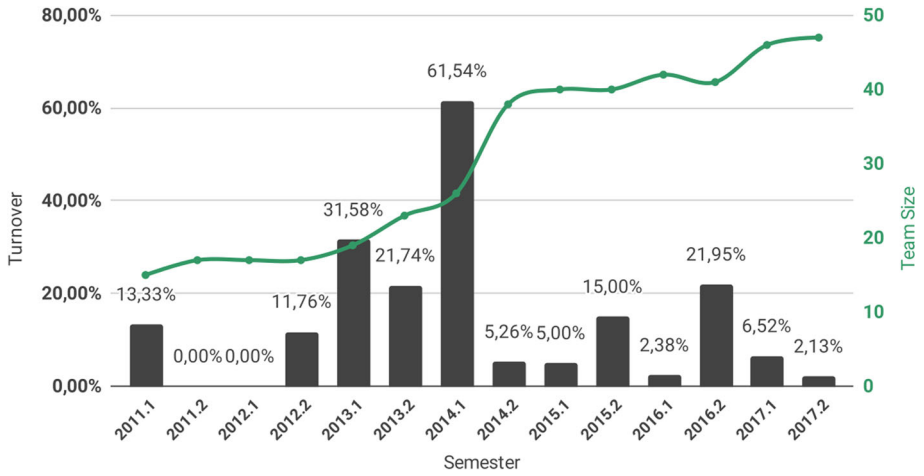

Fig. 9 Development team turnover per semester

before the seventh release (2014.1), and 76.92% before the 2017 current version. Throughout the period analysed in our case study, the institution hired 43 professionals, and 12 others changed their jobs. The average turnover per semester was 14.16% with a median of 9.14% and a standard deviation of 16.59%.

According to Chatzipetrou et al. [28] "the turnover rate for a period of time is calculated by dividing the number of the employees who left during that period, by the average number of employees in that period". Using this definition, we computed the turnover rate of XSA development team. By doing so, we took the studied period from 2011 to 2017 into account, resulting in a turnover rate of approximately 30%.

We judged that the target institution presents a high team turnover during the period observed in our study based on (i) the historical turnover rate of the institution itself and (ii) the turnover rates reported in previous work. On the one hand, regarding (i), before 2011, the institution team turnover rate was very low, close to 0% in several years. Thus, if we compare this historical rate with the one we assess during the period we had studied (30%), we can see that the turnover rate got considerably higher in the XSA team. This phenomenon was due both to the replacement of people who left the institution and due to the expansion of the size of the team. On the other hand, previous studies [28–30] that had considered the team turnover rate as an indicator to evaluate issues concerning to costs, risk, and developer satisfaction suggests that a turnover around 30% is considered high.

The institution that maintains XSA hires professionals through public procurement process. These jobs are attractive for beginners. However, they end up leaving for other companies after a short time. These novice developers have a proper level of foundation skills, which is required to succeed in the public tender process. However, they have a lack of development experience with both the Java language and the XSA system itself. The institution has purchased the XSA system and evolved it to meet their particular needs. However, most of the professionals engaged in the XSA evolution were newly hired. 66.7% of them are in the institution for a maximum of 3 years, which is a period after the XSA acquisition. Therefore, knowledge about EH has not been properly documented and passed on to new developers.

The inexperience of novice developers

Hypothesis: *Novice developers are responsible for inserting most of the EH anti-patterns.*

Previous research has pointed out that novices do not see exception handling as a priority activity. To confirm or refute the impact of novices on the insertion of anti-patterns, we analysed the 2 years that contained the most substantial number of professionals newly hired by the institution, 2015 and 2017.

Materials and methods

We considered two groups of developers, novices, and experts in that analysis. We, then, computed all the changes included in the system and calculated the quantitative of anti-patterns added by each professional. To do this, we developed a programme in Java and used the RepoDriller and PMD tools, which respectively gave us access to the system revisions and the anti-patterns added in each one. We consider novices the developers with less than 2 years of experience in the institution; the experts are those who have more than 3 years of work in the institution.

Procedure

We run RepoDriller and performed data compilation and aggregation. For statistical significance tests comparing

Table 5 Comparison novices vs experts

Data	Number	Violation exclusions	Violation insertions	Java changes	Insertions/ Java changes	Exclusion/ Java changes	Exclusion mean	Insertion mean
Experts - 2015	7	31	20	277	7.22%	11.19%	4.43	2.86
Novices - 2015	7	11	46	559	8.23%	1.97%	1.57	6.57
Experts - 2017	14	56	140	1794	7.80%	3.12%	4.00	10.00
Novices - 2017	6	12	177	920	19.24%	1.30%	2.00	29.50

the two groups (i.e., experts and novices), we used the Wilcoxon-Mann-Whitney test due to the small size of the samples.

Results

After the analysis, we confirmed that the novice programmers inserted more EH anti-patterns than the experienced ones. Also, experts remove more anti-patterns than the novice ones. Table 5 shows a comparison of these absolute values, and the rate of inserted and removed violations to changed Java files in 2015 and 2017.

In 2015, 14 people changed Java system files. The group of experts (7 people) added 20 violations, while the group of novices (7 people) added 46 violations. Novices modified more Java files, meaning they were more prone to errors. Also, some of the novices added most of the violations. In fact, two of them did not add any violations (see Fig. 10). The average rate of inserted violations to changed Java files is very close between the two groups (7.22% and 8.23%, respectively). When we performed significance tests, we observed these differences are not statically significant (Mann-Whitney $U = 22$, $n_1 = 7$, $n_2 = 7$, $p < 0.05$ two-tailed).

Regarding the exclusion of anti-patterns, in 2015, the group of experts removed 31 violations. The group of novices that year deleted only 11 violations even having modified more Java files. Also, one of the experts did almost 50% of violation exclusions. The average rate of removed violations to changed Java files is very different between the two groups (11.19% and 1.97%, respectively). However, when we performed significance tests, we observed these differences are not statically significant (Mann-Whitney $U = 20$, $n_1 = 7$, $n_2 = 7$, $p < 0.05$ two-tailed).

In 2017, 20 people changed the system's Java files. The group of experts, now composed of 14 people, has added 140 violations. The group of novices, consisting of 6 people, included 177 violations. A very high value given the differences in size between the groups. Another interesting fact is that the experts removed 56 violations and the novices, only 12 (see Fig. 11). Three of the novices implemented at least 1 EH anti-pattern for each four Java file changes. The average rate of inserted violation to Java file changes is higher in the novice group (19.84%) than in the expert group (7.8%). When we performed significance tests, we observed these differences are statically significant (Mann-Whitney $U = 12$, $n_1 = 20$, $n_2 = 6$, $p < 0.05$ two-tailed).

These results corroborates the suspicion of project committee members, which indicates a significant impact of team turnover and the experience of developers in the insertion of violations. This novice developers' behaviour is similar to that presented in the work of Shah et al. [5]. The authors identified that beginners replicate EH already structured in the code, throw general exceptions, and use the exception handling without recovery actions. However, as our study revealed, experts also continue to contribute negatively to this scenario. In addition to improving the skills of novices, other measures need to be performed with all team members.

Developers' awareness of EH anti-pattern insertion

Hypothesis: *Developers are not aware they are inserting EH anti-patterns in XSA since they do not know most of these EH anomalies*

Our goal was to analyse programmers' knowledge concerning the EH anti-patterns further. We wanted to investigate if they are inserting or not removing such violations because they are unaware of them.

Materials and methods

To collect data, we created a second questionnaire[7]. The online instrument was intended to see if developers of the XSA system know the most recurring anti-patterns in the source code they implement. The study of Palomba et al. [31] served as the basis for the questionnaire construction. The Palomba et al. also aimed to investigate developers' perceptions about the relationship between bad smells and low-quality planning or implementation.

The instrument elaboration had as an essential step the selection of source code that represented the use of the anti-patterns. We included the most recurring EH anti-patterns in the XSA system. Besides, we have also included anti-patterns related to the use of *Generic Exceptions*, *Catch Generic*, *Destructive Wrapping*, *Throws Generic*, and *Wrong Exception Thrown*. After selecting codes, we have prepared the first version of the form. Questions to identify the respondents profile were also included, such

[7] EH knowledge questionnaire is available at http://twixar.me/nrGn

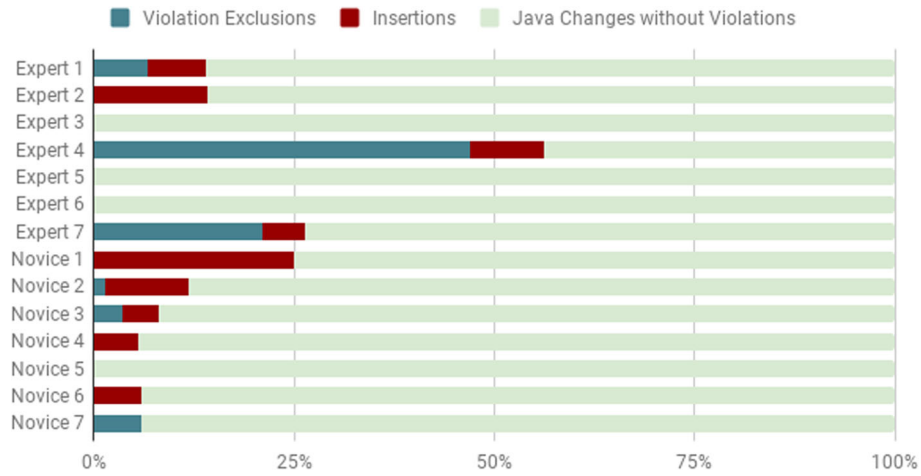

Fig. 10 Comparison novices vs experts—2015

as their experience and their questionnaire evaluation itself.

To fill out the questionnaire, the respondent should review the code snippet of the issue and report any implementation problem or exception design. If he encountered any problems, the form displayed a page where he should describe the anomaly faced. If the developer did not find an error, another code appeared to be evaluated by him. The instrument has five code snippets, only four of them had an EH anti-pattern.

Procedure

Before applying it, two researchers from the Software Engineering area evaluated the instrument. They answered the questions and provided feedback on the form for its clarity and format. From their feedback, we created a new version of the instrument. Finally, before being sent to developers, we conducted a pilot test with a developer of the institution to find out if the questionnaire was suitable for the target audience. From the pilot study, we also established the time required to answer it.

Results

The online form received 14 responses (30.43% of the professionals involved in system development). During the analysis, we separated the respondents who were able to perceive the existence of exception handling problems (the first stage of the question) and the developers who were able to classify the anti-pattern correctly. These results are seen in Fig. 12. The EH anti-pattern least perceived by the developers was the *Destructive Wrapping*. Only 42.86% of the respondents perceived it, and

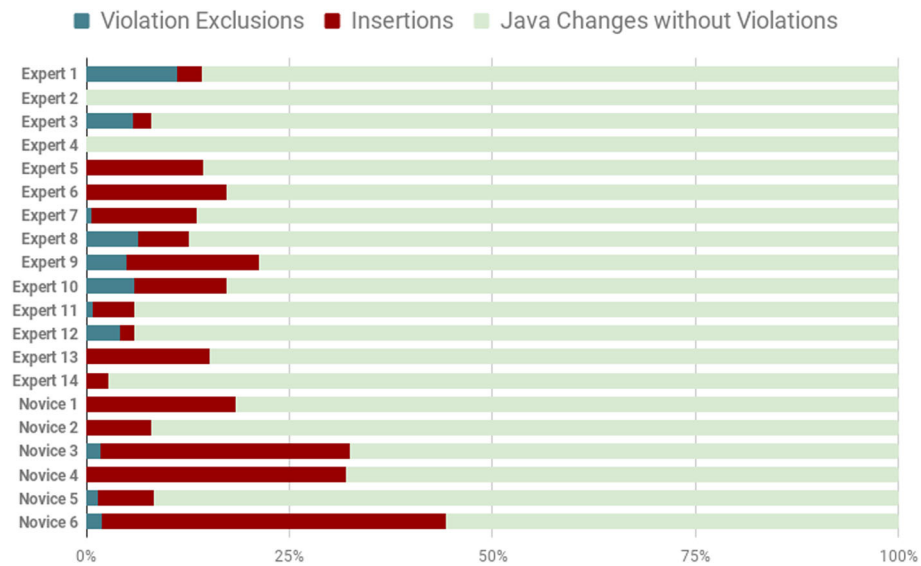

Fig. 11 Comparison novices vs experts—2017

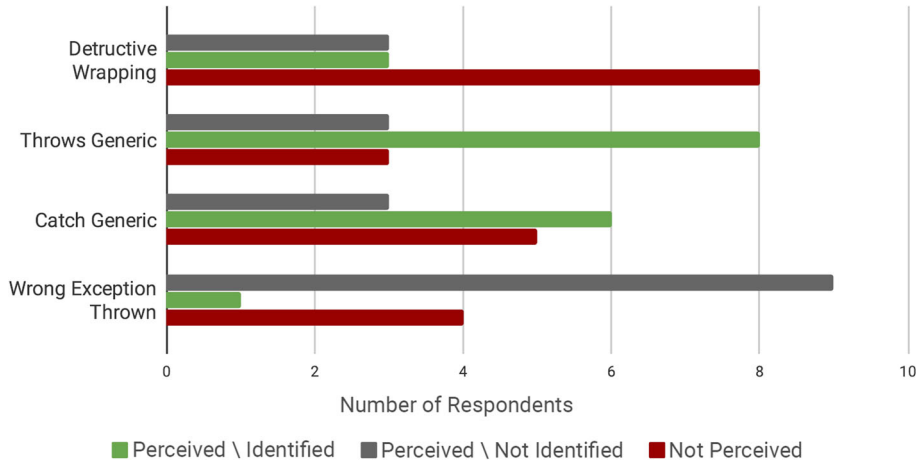

Fig. 12 *Anti-pattern* perception

only 21.43% could classify it correctly. The least identified anti-pattern was the *Wrong Exception Thrown*. Only one respondent was able to recognise it. Almost half of the respondents classified *Catch Generic* and *Throws Generic*, about 42.86% and 57.14%, respectively.

To analyse the answers, we separated the developers again in novices (less than 2 years) and experts (3 or more years) according to the time of experience in the institution. The purpose of this classification was to verify whether the time in the institution could have some influence on the recognition of anti-patterns. The code used in the questionnaire was similar to that found in the XSA system, so, this analysis could give us more insights. On average, 42.86% of the expert answers identified anti-patterns against 21.43% of the beginner responses. Figure 13 shows the differences between the two groups.

Replication of bad practices from the original system

Hypothesis: *The XSA system, likewise other systems with the same origin, has several anti-patterns because they already existed in the original system.*

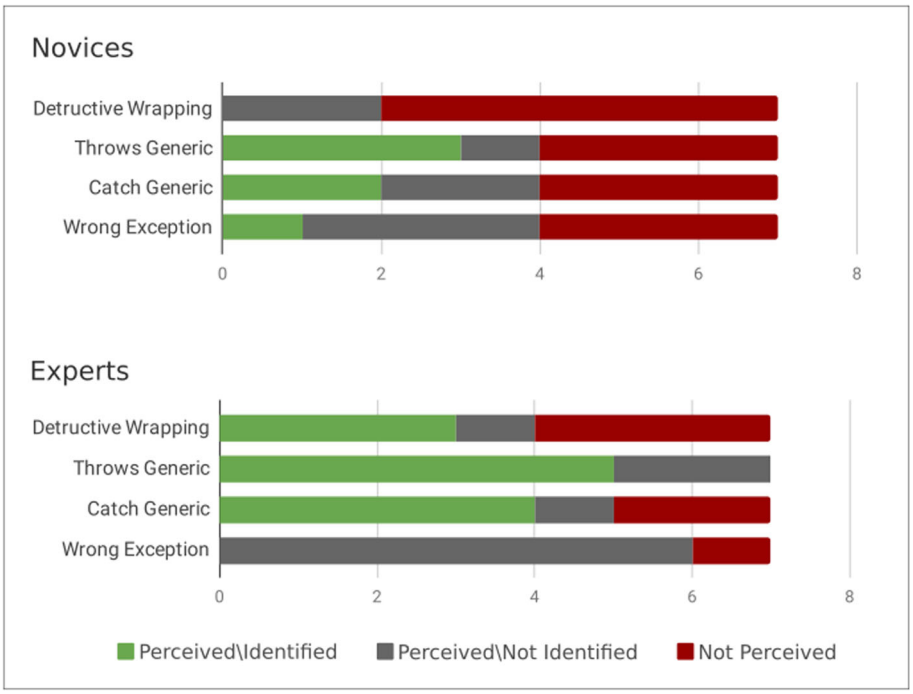

Fig. 13 *Anti-pattern* perception, experts vs novices

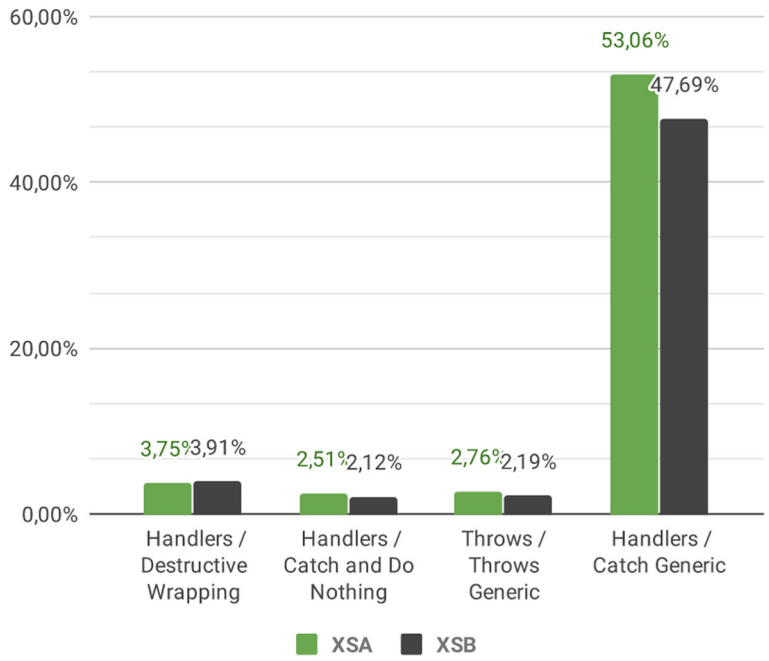

Fig. 14 Comparison between anti-patterns from similar systems

The XSA system has been acquired from another institution. As presented in the unity of analysis 2, the system already had several anti-patterns originally. Those bad practices could be a target of replications, as pointed out by the participants of the semi-structured interview. Therefore, we aimed at investigating whether environments originated or customised from the same system contained similar problems related to exception handling.

Procedure

The approach used in this analysis consisted of comparing the percentage of anti-patterns existing in XSA with the percentage presented in other versions derived from the same initial system. The maintaining institution of XSA had access to the source code of a similar system, but only until the version generated in late 2015. Therefore, the comparison took place with the 2015.2 version of XSA.

Results

With the generated data, we obtained the results presented in Figs. 14 and 15. The system used in the comparison was named XSB in order to ease the comprehension. Some percentages were generated to enable comparisons, such as the following:

- The number of treatments per the amount of Destructive Wrapping, Catch and Do Nothing, and Catch Generic violations; and

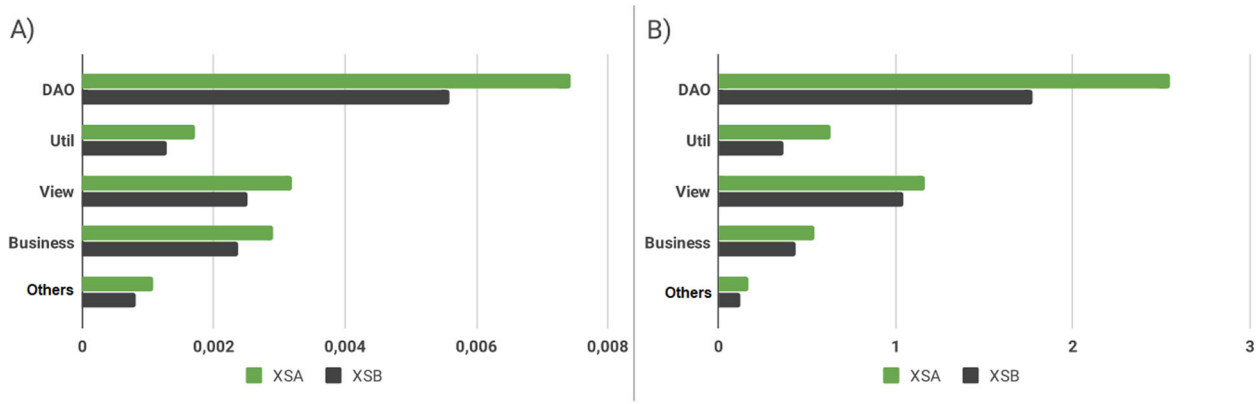

Fig. 15 Comparison of similar system anti-patterns with regard to LoC and number of files. **a** #Violations per#LoC and **b** #violations per #Java file

- The amount of flags per the number of Throws Generic violations.

The results show that the XSA and XSB systems have close percentages. In addition, the same types of violations are more recurrent in both systems. Quantitatively, the XSB system has 693 additional violations. This divergence occurs because the code growth in the XSB system is greater than the increase in violations. Hence, we can conclude that by including new exception handling, developers responsible for the XSB system insert fewer violations than the team responsible for XSA. Despite this, we observed that there was no exception handling prioritisation, since the amount of anti-patterns is greater than in the original version of 2010, shared by both systems.

However, it should be emphasised that the quantitative comparison presented in this subsection is not sufficient to contrast how different teams deal with the exception handling. Thus, it is possible for a team to be more active in changing exceptional flows than another, for example, by modifying EH of system modules, refactoring some components, increasing the amount of anti-pattern in other components, or replacing instances of the exceptions used, even if this does not affect the final amount observed in Figs. 14 and 15. Consequently, there is no way to see if the anti-patterns existing in the initial version affected the development of the systems or even if they were replaced by other instances of the same anti-patterns. A historical analysis of the systems would be required to detect such practices. Unfortunately, no access to the XSB system modification history has been obtained. Therefore, Assumption 05 was only partially verified.

Overall discussion

After performing the case study, we can conclude that, answering RQ1, XSA professionals consider the Exception Handling an essential and necessary feature. Even novice developers expressed this opinion. Despite these answers in the online survey, the code analysis revealed many anti-patterns in the system's source code. In this section, we point out possible interpretations of this phenomenon. We also contrast our findings with previous work results.

Case study peculiarities

Regarding RQ2, the *Generic Catch* anti-pattern is the most expressive among those detected in the XSA code. Its significant number of occurrences is a peculiarity not evidenced in other works. For instance, Padua and Shan [15] identified an average of 31.9% of handlers affected by this anti-pattern. Some studies showed the number of bugs caused by this bad smell does not have a significant value [12, 19].

In the study of Padua and Shang [15], the anti-pattern *Generic Throw* is also not among the most recurrent. However, we found several occurrences of this anti-pattern in XSA code. They represent 14.42% of the violations in the and PMD tool. This anti-pattern also contributes to the proliferation of *Generic Catches* in the code.

Regarding the evolution of the system, Nogueira et al. [32] show a decrease in anti-pattern violations when comparing the first and last software releases. They affirmed refactoring tends to exchange empty handlers by specialised handlers. However, the XSA system is not yet following this trend. The anti-pattern occurrences have grown again since 2014 (see Fig. 5).

Lack of documentation issues

The organisation that maintains the XSA has an incomplete and not up-to-date architecture documentation. A set of policies for implementing EH are not available in the institution. The developers mentioned these problems both in the online survey and in the semi-structured interview. This phenomenon is not a peculiarity of XSA. Similarly, 80% of the respondents from the Ebert et al. [12] survey claim EH policies are lacking in the organisations in which they work. It is notorious these guidelines have an essential role to play in influencing the choice of appropriate EH implementation [5]. However, its neglect is still latent.

From a general perspective, XSA system developers report they are not adept at documenting EH design and code. Inadequate documentation of EH is already a known issue [12]. It impacts the developers' understanding of the consequences of not offering adequate EH [21]. Also, it has implications in their comprehension of the type of exceptions thrown by a method [14].

EH policy, turnover, and skill issues

By analysing all the data extracted from the three units, we can confirm some assumptions concerning RQ3. We believe the absence of EH policies and guidelines are the leading causes to EH anti-pattern dissemination in XSA. The lack of static analysis tools to monitor the EH anti-patterns insertion plays a crucial role too. However, not less important, the high team turnover and, consequently, developers' inexperience with XSA, strongly contribute to the growth and dissemination of EH anti-patterns in XSA.

Sometimes, novice developers have also poor skills in Web development with Java. This high team turnover is also a problem in other similar public institutions. Thus, diagnosing the EH status and proposing possible actions to minimise them may help other organisations to improve the quality of their systems.

Threats to validity

Internal validity. The unit of analysis 1 considers only the information provided in the online questionnaire. This

method of collection may have bias depending on the interest of the respondents. However, the results of the second and third units of analysis validated some collected by the unit of analysis 1 (e.g., the occurrence of anti-patterns in the view layer, the absence of explicit EH policies, and the developers' knowledge about EH bad practices).

Another threat to validity concerns the percentage of online survey and quiz respondents, about 53% and 30.43%, respectively. To deal with this threat, we use the interview to perform data triangulation with the answers given by survey respondents and data from the source code change analysis. Additionally, this percentage of responses is quite similar to the achieved in similar studies in the literature.

This work does not analyse all aspects relevant to EH (e.g., the situation in which developers add anti-patterns to the code, the XSA tests, and the training received by the developers). We also did not inspect all existing anti-patterns. The analysis tools we use in the study limit the breadth of the case study. However, these limitations did not prevent the achievement of the objectives. We have been able to understand the state of implementing the EH in a real system, presenting its vulnerabilities and its possible causes from the insights of the professionals engaged in its development.

External validity. As a case study, this research fits into a particular context. We did not select the participants randomly. They work in the same development environment and share experiences (training, documentation, code, and work difficulties). This context makes it difficult to generalise the results obtained for other public institutions and companies that develop software. For example, organisations with more code regulation and inspection, or those that use other programming languages for Web application development, may produce distinct results.

Finally, it is important to mention that the lack of guidelines and a global policy for exception handling design and implementation might have a different impact on systems with different architectures and implemented using one or multiple programming languages.

Construction validity. The online forms, answered without our help, may not have been well understood by responders. To mitigate this threat, we performed several trials with other researchers and software developers to validate the comprehensibility of each question, the organisation, and time needed to fill the questionnaire.

Another threat concerns the VbR, the tool we developed to gather information about XSA evolution and team turnover. To mitigate this threat, we developed such a tool on top of well-tested libraries, such as JavaParser and RepoDriller, giving us some guarantees of reliability. Besides, looking at reducing the chances of bias, VbR uses a well-tested and widely adopted open source tool, PMD, for the EH anti-pattern detection.

Related work

In this section, we present the main related work to this research. Those researches have an emphasis on the exception handling analysis. They use as data sources both the software developers' opinion and artefacts that either belongs to the system development cycle, such as source code, identified bug reports, and documentation.

We selected the related studies using the Snowballing literature review [33], applying both backward and forward analyses. The inclusion criteria were (i) studies should analyse the exception handling with respect to real software development cycle elements and (ii) studies should have been published between 2010 and 2018. The exclusion criterion was that studies whose main contribution is the proposition of tools, techniques, and methods. There were selected papers whose title satisfies the following search string:

exception AND ("*handling*" OR "*handler*") AND ("*case study*" OR "*empirical study*" OR "*exploratory study*")

Figure 16 shows the snowballing results. The initial selection of studies resulted in 11 papers. After analysing the title and applying the inclusion and exclusion criteria, we removed five papers, thus resulting in six papers in the initial set. Those remaining six pages were input to the first iteration of the Snowballing. The backward analysis found 10 new papers, while 12 new studies were identified in the forward counterpart. From the new set of 22 papers, only 12 papers satisfied the inclusion and exclusion criteria. With those 12 papers, we started the second Snowballing iteration, in which four papers were found in the backward analysis and eight studies were selected in the forward analysis. After applying the inclusion and exclusion criteria to those 12 papers identified, only three papers were included. At the end of the Snowballing process, our result set consisted of 21 studies.

After that, we separated the studies according to the source of the data collected by the authors and classified them in the following categories: developer-centric studies (human aspect), with two studies, software-centric studies (technical aspect), containing 14 studies , and hybrid studies (includes both human and technical aspects), which included five papers. In the next subsections, we detail the most cited work in each of the three categories in which they were classified.

Developer-centric studies

The work in [34] addresses the perception of developers and organisations about bugs in the exception handling. It was used as an online questionnaire, answered by 154

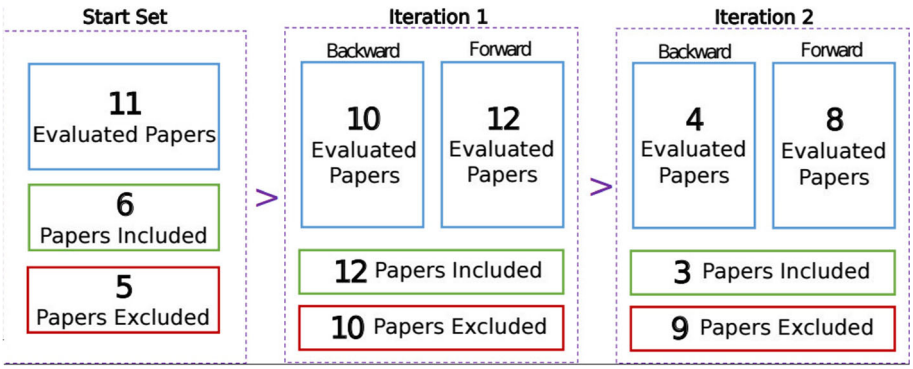

Fig. 16 Snowballing results

programmers and researchers with experience in Java, to collect the information. The authors' main conclusions are (i) the exception handling code are rarely tested and documented, (ii) experienced professionals tend to be more critical with respect to the quality of the exceptional behaviour code, and (iii) developers use the exception handling to implement fault tolerance strategies and to improve the code quality.

Two studies to capture both novices and experienced developers' view about the exception handling are described in [5]. In total, 15 Java developers participated in a semi-structured interview. According to the results, less experienced developers try to avoid exception handling or just to imitate existing practices in the code, thus not devoting time to handle exceptions properly. On the other hand, experienced developers see exception handling as an integral and inseparable part of the software development process and also use it to provide better feedback about the runtime errors, to prevent data corruption, and to control the system execution flow.

The studies in [5] and [34] aimed at understanding the developers' perception of exception handling. They provided a better understanding of the human side regarding the exception handling and inspired new studies, such as this one.

Software-centric studies

The authors of [14] conducted a study to understand the approaches used in exception handling in Java libraries by analysing their exception flows. In addition, posts in bug reporting systems regarding exception handling were also inspected and linked to the analysed flows. The authors found out that most of the analysed libraries do not document *RuntimeExceptions*. Exception handling anti-patterns were detected in 25% of the analysed code and more than 20% of the reported problems of the most popular libraries were related to exception handling.

To verify whether the exception handling can be considered risky is the main objective of [35]. For this, it analysed the defect density in classes of the Eclipse IDE and exception handling metrics extracted from its source code. The results reveal a decline in the amount of defects during the software evolution. However, the defects associated with the exception handling continued to grow over the 6 years considered in the research. The opposite happens with defects not related to exception handling.

The authors in [4] conducted a study to understand the relationship between programme evolution and its robustness. The research focuses on the evolution of exception handling in systems developed in Java and C#. The analysis of the exceptional flows is carried out with the use of metrics that indicate the changes made between releases of the same system and its impact. Among these metrics, we can cite the number of Unhandled Exceptions along the execution flow, which provides an indication of system robustness. The authors conclude that the exception handling in Java systems undergoes more modifications over time. Nonetheless, they have fewer scenarios that negatively impact their robustness. Conversely, the exception handling in C# systems seems to be more fragile, indicating a greater amount of Unhandled Exceptions in the verified scenarios.

The work in [6] intended to show which methods with undocumented exceptions are responsible for application failures. The authors analysed several *stack traces* related to failures in Android applications to identify the methods responsible for them. They also looked at the Android API source code and catalogued methods that have documented exceptions. From the collected data, they found that 69% of the methods involved in failures did not have their exceptions documented. Moreover, only 18% of public or protected methods had such documentation and 24% of the methods found in the *stack traces* launched generic exceptions, such as *RuntimeException* and *NullPointerException*, which were not documented on the interfaces.

The authors in [19] tried to understand which types of either exceptional failures or failures due to misuse or

lack of exception handling occur in systems. The authors identified releases of Hadoop and Tomcat Apache systems related to lack of exception handling. Almost 41% of the faults were classified in the category "Information Swallowed", which contains faults caused by the lack of adequate information of an exception, also encompassing the anti-pattern known as Destructive Wrapping.

The papers discussed in this subsection are studies that use several elements such as source code, system logs, and reported bugs to understand how developers use exception handling in practice. In addition, they expose the exception handling deficiencies and its impact in the involved systems. Since they focus on software, they do not present the developers' perceptions and the difficulties found in development environments that interfere in the exception handling quality. These aspects, however, are investigated in our work.

Hybrid studies

The authors in [36] investigated the use of exception handling mechanisms in C++ in order to understand how developers use them for error recovery in the midst of other activities required in the software development. They investigated the practices implemented in the source code of 65 open-source projects by means of static analyses provided by a proprietary tool. They found out that, on average, only 0.03% of the code is intended for recovery actions, that the most thrown exception is of type *RuntimeException* and that 16.71% of the handlers are empty. In addition, they conducted an online search to capture the understanding and perception of C++ developers about exception handling. Most respondents agreed that developers often avoid handling exceptions due to, among other reasons, performance issues and lack of knowledge about how to use these elements.

An analysis of reported errors of two systems, Tomcat and Eclipse, with the objective of better understanding the causes, severity, frequency, and difficulty of solving these bugs, is shown in [12]. Additionally, through an online questionnaire, they tried to understand the perceptions that the developers of the analysed systems and other institutions have about exception handling and its related bugs. At the end, the authors concluded that organisations often do not institutionalise policies for exception handling, since they rarely have specific tests or documentation for that purpose. They also found that exception handling bugs are rarely reported, and the main causes of such bugs are the lack of exception handling, exceptions that should be (but are not) thrown, and programming errors in catch blocks.

Finally, the authors in [37] investigated when reported *stack traces* may express circumstances in which there are a greater likelihood of bugs in exception handling in Android applications. Among the findings, they highlight that the majority cause of the stack trace problems were related to programming bugs (e.g., *NullPointerException*, *IllegalArgumentException*, *RuntimeException*), 65% of application failures were related to runtime exceptions, and only one was documented with the tag *@throws*.

Discussion

The studies presented in this section propose approaches or distinct analysis methods to help the comprehension of exception handling usage in both systems and developers' view. Their main contributions focus on four topics: (i) exception handling anti-patterns [4, 8, 14, 15, 32, 38], exception handling defects [19, 35, 39], developers' perception [5, 34, 40], and exception handling evolution [41, 42]. Some of those studies [12, 36, 38] also have interception with other topics since they cross-check the information collected from different sources.

The developers' perception was assessed through online questionnaires [34, 36–38] and semi-structured interviews [5, 40]. In general, the researches aimed at understanding how developers use the exception handling, what approaches are used, the differences between novices and experienced developers, what they understand about exception handling bugs and how they use exception handling in Android and Swift applications.

There have been found anti-patterns in different domains, such as both desktop and server applications, and programming libraries [8, 15], as well as in different programming languages [4, 15]. There were used manually source code inspection, scripts to automate the detection of anti-patterns, and several releases of the system in order to analyse the evolution of anti-patterns [4, 8, 32].

Similarly, the reported defects were associated to the inadequate use of exception handling. The data were collected through bug reports and the system releases. The authors of those researches categorised bugs [12, 19] and identified whether classes that handle exceptions are more defect prone [39, 43]. They also investigated when using exception handling is risky by calculating defect density metrics [35] and found out situations with more likelihood to have bugs [35]. Furthermore, the lack of exception documentation has also been analysed as a possible cause of application failures.

The evolution of the exception handling was perceived by observing its constructs (*try*, *catch*, *throws*), the use of customised exceptions, and anti-patterns throughout the analysed system releases. Those information were necessary to understand the development of the portion of code destined for exceptional flows [36] and which changes impact the exception handling [8]. In addition, there was investigated how the exceptional interfaces evolve [42], whether the developers use more customised exceptions as they acquire more knowledge about the project [41], and how normal and exceptional code changes are related

to exception handling failures in Java, C# e Android applications [4, 44].

We noticed that the hybrid researches, which investigate both human and technical aspects regarding EH, produce a better comprehension of the investigated phenomena and bring results that are more powerful and better justified than those that adopt only one strategy. Also, few studies had access simultaneously to the code and the professionals involved in the development of the same system. Hence, we chose that approach since a study carried out closer to the real environment that originated the investigated phenomenon can elucidate its causes better and produce positive effects in this environment (e.g., improving the quality of the future releases of the system).

Finally, a few papers focused on the presence and evolution of anti-patterns in the development of Web systems, which is the main topic of this work. Another critical difference of our work is the study and analysis of the impact that the team turnover has in the production and correction of exception handling anti-patterns.

Final remarks and future work

The exception handling is typically associated with the software capability to recover itself from abnormal occasions. However, despite its importance, EH is commonly neglected by developers, causing several undesired situations in software usage (e.g., unfriendly error messages, system fail). Previous studies show that EH problems come from human or technical aspects. In this realm, this work aimed at comprehending how software engineers of a public institution perceive the EH. Also, we wanted to discover the technical situations faced in that real environment, which affect the quality of the EH in a particular large-scale system.

We conducted a case study with a large-scale Web system. The online survey showed developers are aware of EH importance even without an explicit EH policy in the institution. However, as the code analysis revealed, many EH anti-patterns were found, and they are not a novelty in the system. We can conclude that recognising the EH importance was not enough to avoid the insertion and proliferation of anti-patterns in the system code.

Our findings and the beliefs of the project committee indicate team turnover has an essential impact on the insertion of EH anti-patterns. Data from the code analysis during the years 2015 and 2017 showed that novices inserted more EH anti-patterns. They also remove less EH violations than experts. In the case of 2017, we also found that there is a significant difference between the groups concerning the ratio of violations to Java files changes. However, as our research revealed, experts also continue to contribute negatively to this scenario. In addition to improve the skills of novices, the adoption of practices and tools that support a better development and maintenance process is required (e.g., EH policies, conformance checking of these policies by a quality team).

Our case study offers insights and confirms the impact of team turnover and EH skills in EH anti-pattern dissemination in XSA. Nonetheless, other studies should confirm if this phenomenon occurs systematically or it is a particularity of XSA.

As future research work, we intend to address the creation of guidelines for EH documentation and policies for Java Web systems. This aims at improving the developers' comprehension of how the EH should be used in a large-scale system. Furthermore, we intend to analyse the impact of EH training and code analysis tool adoption in the evolution of XSA's EH anti-patterns, because the absence of those activities may also be contributing to the presence of EH anti-patterns.

Finally, another interesting future work could be to study a way to support the refactoring of the Catch Generic anti-pattern (the most prevalent in XSA), pointing out the benefits and possible drawbacks to conduction such global system refactoring.

Acknowledgements
We acknowledge the XSA's maintenance institution for giving us access to their code. We also are glad to the software engineers that helped us in this case study investigation.

Authors' contributions
DBCde S is the main author of this research, which presents the result of her Computer Science Master dissertation. WV and LSR were their Master Degree supervisors, which also analysed the research data and written most of the sections of the paper. PHMM contributed on identifying insights from the data triangulation and on writing the manuscript. All authors read and approved the final manuscript.

Funding
The research has not received any external funding.

Availability of data and materials
The datasets generated and analysed during the current study are not publicly available due to the fact XSA is a third-party system with a critical role in its institution. However, the data analysis is available from the corresponding author on a reasonable request.

Competing interests
The authors declare that they have no competing interests

Author details
[1] Federal University of Ceará, Av. Humberto Monte, 60440-593, Fortaleza, Brazil.
[2] State University of Ceará, Av. Dr. Silas Munguba, 60741-000, Fortaleza, Brazil.

Received: 12 February 2019 Accepted: 29 December 2019
Published online: 27 January 2020

References

1. Garcia AF, Rubira CM, Romanovsky A, Xu J (2001) A comparative study of exception handling mechanisms for building dependable object-oriented software. J Syst Softw 59(2):197–222
2. Shahrokni A, Feldt R (2013) A systematic review of software robustness. Inf Softw Technol 55(1):1–17
3. Buhr PA, Mok WYR (2000) Advanced exception handling mechanisms. IEEE Trans Softw Eng 26:820–836
4. Cacho N, Barbosa EA, Araujo J, Pranto F, Garcia A, Cesar T, Soares E, Cassio A, Filipe T, Garcia I (2014) How does exception handling behavior evolve?

An exploratory study in Java and C# applications. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE. https://doi.org/10.1109/icsme.2014.25
5. Shah H, Gorg C, Harrold MJ (2010) Understanding exception handling: viewpoints of novices and experts. IEEE Trans Softw Eng 36(2):150–161
6. Kechagia M, Spinellis D (2014) Undocumented and unchecked: exceptions that spell trouble. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014. ACM, New York. pp 312–315
7. Chang B-M, Choi K (2016) A review on exception analysis. Inf Softw Technol 77(C):1–16
8. Oliveira J, Borges D, Silva T, Cacho N, Castor F (2018) Do android developers neglect error handling? A maintenance-centric study on the relationship between android abstractions and uncaught exceptions. J Syst Softw 136(Supplement C):1–18
9. Osman H, Chiş A, Corrodi C, Ghafari M, Nierstrasz O (2017) Exception evolution in long-lived Java systems. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE. https://doi.org/10.1109/msr.2017.21
10. Filho JLM, Rocha L, Andrade R, Britto R (2017) Preventing erosion in exception handling design using static-architecture conformance checking (Lopes A, de Lemos R, eds.). Springer, Cham
11. Cacho N, César T, Filipe T, Soares E, Cassio A, Souza R, Garcia I, Barbosa EA, Garcia A (2014) Trading robustness for maintainability: an empirical study of evolving C# programs. In: Proceedings of the 36th International Conference on Software Engineering – ICSE 2014. ACM Press. https://doi.org/10.1145/2568225.2568308
12. Ebert F, Castor F, Serebrenik A (2015) An exploratory study on exception handling bugs in Java programs. J Syst Softw 106:82–101
13. Sawadpong P, Allen EB (2016) Software defect prediction using exception handling call graphs: a case study. In: 2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE). IEEE. https://doi.org/10.1109/hase.2016.13
14. Sena D, Coelho R, Kulesza U, Bonifácio R (2016) Understanding the exception handling strategies of Java libraries: an empirical study. In: Proceedings of the 13th International Workshop on Mining Software Repositories – MSR '16. IEEE. https://doi.org/10.1145/2901739.2901757
15. de Pádua GB, Shang W (2017) Studying the prevalence of exception handling anti-patterns. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE Press. https://doi.org/10.1109/icpc.2017.1
16. Barbosa EA, Garcia A, Robillard MP, Jakobus B (2016) Enforcing exception handling policies with a domain-specific language. IEEE Trans Softw Eng 42(6):559–584
17. Barbosa EA, Garcia A (2017) Global-aware recommendations for repairing violations in exception handling. IEEE Trans Softw Eng PP(99):1–1. https://doi.org/10.1109/TSE.2017.2716925
18. de Sousa DBC, Maia PH, Rocha LS, Viana W (2018) Analysing the evolution of exception handling anti-patterns in large-scale projects: a case study. In: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS '18. ACM, New York. pp 73–82. https://doi.org/10.1145/3267183.3267191
19. Barbosa EA, Garcia A, Barbosa SDJ (2014) Categorizing faults in exception handling: a study of open source projects. In: 2014 Brazilian Symposium on Software Engineering. IEEE. https://doi.org/10.1109/sbes.2014.19
20. Gallardo R, Hommel S, Kannan S, Gordon J, Zakhour S. B. (2014) The Java tutorial: a short course on the basics, 6th edn. Java Series. Addison-Wesley Professional, Boston
21. Yuan D, Luo Y, Zhuang X, Rodrigues GR, Zhao X, Zhang Y, Jain P, Stumm M (2014) Simple testing can prevent most critical failures: An analysis of production failures in distributed data-inte nsive systems. In: 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). pp 249–265
22. Sinha S, Orso A, Harrold MJ (2004) Automated support for development, maintenance, and testing in the presence of implicit control flow. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society. pp 336–345
23. Chen C-T, Cheng YC, Hsieh C-Y, Wu I-L (2009) Exception handling refactorings: directed by goals and driven by bug fixing. J Syst Softw 82(2):333–345
24. Coelho R, Rocha J, Melo H (2018) A catalogue of Java exception handling bad smells and refactorings. In: Pattern Languages of Programs (PLoP), 2018 25th International Conference On. The Hillside Group. http://www.hillside.net/plop/2018/papers/proceedings/
25. Correa AL, Werner CM, Zaverucha G (2000) Object oriented design expertise reuse: an approach based on heuristics, design patterns and anti-patterns. In: Software Reuse: Advances in Software Reusability. Springer. pp 336–352. https://doi.org/10.1007/978-3-540-44995-9_20
26. Runeson P, Host M, Rainer A, Regnell B (2012) Case study research in software engineering: guidelines and examples, 1st edn.. Wiley Publishing, Hoboken
27. Creswell JW (2014) Research design: qualitative, quantitative, and mixed methods approaches. Sage publications, California
28. Chatzipetrou P, Šmite D, van Solingen R (2018) When and who leaves matters: emerging results from an empirical study of employee turnover. In: Proceedings of the 12th ACM/IEEE International symposium on empirical software engineering and measurement, ESEM '18. ACM, New York. pp 53–1534. https://doi.org/10.1145/3239235.3267431
29. Haines VY, Jalette P, Larose K (2010) The influence of human resource management practices on employee voluntary turnover rates in the canadian non governmental sector. ILR Review 63(2):228–246. https://doi.org/10.1177/001979391006300203
30. Smite D, van Solingen R (2016) What's the true hourly cost of offshoring? IEEE Softw 33(5):60–70. https://doi.org/10.1109/MS.2015.82
31. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014) Do they really smell bad? A study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE. pp 101–110. https://doi.org/10.1109/icsme.2014.32
32. Nogueira AF, Ribeiro JC, Zenha-Rela MA (2017) Trends on empty exception handlers for Java open source libraries. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE. pp 412–416. https://doi.org/10.1109/saner.2017.7884644
33. Wohlin C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14. ACM Press. https://doi.org/10.1145/2601248.2601268
34. Ebert F, Castor F (2013) A study on developers' perceptions about exception handling bugs. In: 2013 IEEE International Conference on Software Maintenance. IEEE. https://doi.org/10.1109/icsm.2013.69
35. Sawadpong P, Allen EB, Williams BJ (2012) Exception handling defects: an empirical study. In: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering. IEEE. pp 90–97. https://doi.org/10.1109/hase.2012.24
36. Bonifácio R, Carvalho F, Ramos GN, Kulesza U, Coelho R (2015) The use of C++ exception handling constructs: a comprehensive study. In: 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE. pp 21–30. https://doi.org/10.1109/scam.2015.7335398
37. Coelho R, Almeida L, Gousios G, Van Deursen A, Treude C (2017) Exception handling bug hazards in Android. Empir Softw Eng 22(3):1264–1304
38. Queiroz FD, Coelho R (2016) Characterizing the exception handling code of android apps. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). IEEE. pp 131–140. https://doi.org/10.1109/sbcars.2016.25
39. Marinescu C (2011) Are the classes that use exceptions defect prone? In: Proceedings of the 12th international workshop and the 7th annual ERCIM workshop on Principles on software evolution and software evolution - IWPSE-EVOL '11. ACM. https://doi.org/10.1145/2024445.2024456
40. Cassee N, Pinto G, Castor F, Serebrenik A (2018) How swift developers handle errors. In: Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18. https://doi.org/10.1145/3196398.3196428
41. Osman H, Chiş A, Schaerer J, Ghafari M, Nierstrasz O (2017) On the evolution of exception usage in java projects. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE. pp 422–426. https://doi.org/10.1109/saner.2017.7884646
42. Barbosa EA, Garcia A (2011) Analyzing exceptional interfaces on evolving frameworks. In: 2011 Fifth Latin-American Symposium on Dependable

Computing Workshops. IEEE. pp 17–20. https://doi.org/10.1109/ladcw.2011.19
43. Marinescu C (2013) Should we beware the exceptions? An empirical study on the eclipse project. In: 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE. https://doi.org/10.1109/synasc.2013.40
44. Oliveira J, Cacho N, Borges D, Silva T, Castor F (2016) An exploratory study of exception handling behavior in evolving android and java applications. In: Proceedings of the 30th Brazilian Symposium on Software Engineering - SBES '16. ACM. https://doi.org/10.1145/2973839.2973843

Publisher's Note