Improving Developers Awareness of the Exception Handling Policy

Taiza Montenegro*†, Hugo Melo†, Roberta Coelho†, Eiji Adachi‡
*Informatics Superintendence
†Department of Informatics and Applied Mathematics
‡Digital Metropolis Institute
Federal University of Rio Grande do Norte
Natal, Brazil
taiza@ppgsc.ufrn.br, hugofm@ppgsc.ufrn.br, roberta@dimap.ufrn.br, eiji@imd.ufrn.br

***Abstract*—The exception handling policy of a system comprises the set of design rules that specify its exception handling behavior (how exceptions should be handled and thrown in a system). Such policy is usually undocumented and implicitly defined by the system architect. Developers are usually unaware of such rules and may think that by just sprinkling the code with catch-blocks they can adequately deal with the exceptional conditions of a system. As a consequence, the exception handling code once designed to make the program more reliable may become a source of faults (e.g., the uncaught exceptions are one of the main causes of crashes in current Java applications). To mitigate such problem, we propose Exception Policy Expert (EPE), a tool embedded in Eclipse IDE that warns developers about policy violations related to the code being edited. A case study performed in a real development context showed that the tool could indeed make the exception handling policy explicit to the developers during development.**

***Index Terms*—exception handling, exception handling policy, Eclipse plug-in**

I. Introduction

Exception handling (EH) mechanisms have been embedded in several mainstream programming languages (e.g., Java, C++, C#) as a way to promote the development of robust systems [1]. However, recent studies have shown that recurrent failures in software systems are caused by poor implementation and maintenance of the exception handling code [2], [3], [4], [5]. In other words, despite being aimed at improving software robustness, exception handling mechanisms are actually hindering it.

In practice, developers implement and maintain exception handling code without being aware of the exception handling policy, i.e., the set of design rules that specify how exception handling code should be implemented in a program [6]. In fact, exception handling is still an afterthought in many software projects [2] and exception handling policies are still undocumented and implicitly defined by software architects [7]. For this reason, programmers often treat exception handling as an ad hoc process, sprinkling catch-blocks carelessly throughout the code in all places [8] without being warned of potential faults being introduced in the source code. Such behavior may ease the introduction of exception handling-related faults in software projects.

Recently, domain-specific languages have been proposed to specify and verify exception handling policies of Java systems [9], [6]. The works that propose such languages, besides allowing the definition an exception handling policy, provide ways of checking the EH policy for a system in operation (at runtime) [9], or statically at build time [6]. Since such checking steps are not performed during implementation and maintenance phases, developers are not aware of which parts of their code violate the specified policy, thus making them still unaware of the design rules that govern exception handling in their systems.

In this work, we propose a tool called *Exception Policy Expert* (EPE) to make the exception handling policy explicit to developers on their development environment. This tool loads a set of exception handling rules (previously defined by the software architects) and performs intra-method analyses of the code being edited (on the IDE). The results of such analyses may be of two kinds: (i) suggestions of places where exceptions should be handled (according to a pre-defined policy); or (ii) compile time warnings when any piece of the edited code breaks one of the rules that composes the policy. By continuously checking the exception handling policy during development time, such tool can make developers aware of it. By doing so, such tool may prevent known exception handling bugs caused by violations of the exception handling policy (e.g., unintended handler action [10] - an exception handling bug very difficult to detect which occurs when an exception is mistakenly handled by catch-block that was not intended to handle it).

This paper also reports a case study that quantitatively assesses the EH policy violations detected in a real development context of a medium-sized (i.e., 40 developers) software development company. The case study was conducted on an industrial web-based software system comprising of 593 KLOC of Java source code of which around 4.5 KLOC are dedicated to exception handling. Along the evaluation period, a group of 6 developers used the tool on a daily basis (3 to 8 days). Overall, the tool was used in 30 working days. The results have shown that the proposed approach could detect several EH policy violations during development. Overall the tool could detect 214 distinct EH violations (i.e., 202 *improper*

handling violations and 12 *improper throwing* violations). Moreover, the proposed approach provided 174 suggestions related to the points in the code where the exception being thrown should be handled. This paper makes the following contributions:

- We propose EPE, a tool for detecting exception handling rules violations in Java applications during development.
- We present the intra-method analyses performed by the tool and integrated to the IDE.
- We evaluate EPE in a real development environment. The tool revealed approximately 7 distinct EH violations per person per day and approximately 6 distinct EH tips per person per day.

The contributions of this work allow for: (i) developers of Java applications to make more informed decisions in the presence of exceptions (i.e., whether signaling or handling), and (ii) designers Java applications to make the exception handling policy explicit during development to the development team - instead of being implicitly defined or defined in a single artifact that contains all the rules that compose the policy (as proposed by the existing approaches mentioned previously [9], [6]). Although the tool presented here works for Java applications, its supporting ideas can also be useful to the development of systems developed in other languages that also have an embedded exception handling mechanism. The remainder of this paper is organized as follows. Section II presents background about exception handling mechanisms. Section III discusses the study motivation. Section IV presents and evaluates the EPE tool. Section V provides discussions and lessons learned. Section VI presents related works and Section VII concludes this work.

II. Exception Handling Mechanism

In order to support the reasoning about the exception handling policy of a system, we present the main concepts of an exception-handling mechanism. Since our work focus on Java language, we related such concepts to their representation in Java. An exception handling mechanism is comprised of four main concepts described next: the exception representation, the exception signaler, the exception handler, and the exception model.

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

Exception Signaling - in Java an exception can be explicitly signaled using the throw statement or implicitly signaled by the runtime environment (e.g., *NullPointerException*, *OutOfMemoryError*). In this work, we focus on the exceptions that are explicitly signaled.

Exception Handling and Handler Binding - in Java, once an exception is signaled, the runtime environment looks for the nearest enclosing exception handler (Javas try-catch block), This search for the handler on the invocation stack aims at increasing software reusability, since the invoker of an operation can handle the exception in a wider context.

Exception Model - defines how signalers and handlers are bound. In Java, once an exception is signaled, the runtime environment looks for the closest exception handler and unwinds the execution stack if necessary.

III. Motivation

As mentioned before, some works have already mentioned that the exception handling policy is usually implicitly defined by the architect [7], [9], [6] and are not known by most developers. However, no empirical study has been conducted to investigate this issue.

To better understand how developers deal with the exception handling policy during development, we conduct a series of semi-structured interviews. The study was conducted in the context of a professional software development team of SINFO/UFRN. SINFO/UFRN is the Informatics Superintendence of Federal University of Rio Grande do Norte (UFRN) in Brazil. The SINFO development team is responsible for developing SIG (i.e., Sistemas Integrados de Gestao), a family of large-scale web-based information systems which perform the full automation of university management activities and that is been deployed in around 20 Brazilian federal institutions.

Overall the team was composed of 40 developers working on the development and maintenance of a Java Web-based information system. For the semi-structured interview, we interviewed 8 developers, in a total of 10 contacted. They have the average of 5,98 years (median: 6,5 years) of professional experience in software development. Each interview lasted on average 39,75 minutes (median: 37,5 minutes). We used open coding technique [11] to iterate through the interview transcripts. The interviews revealed a lack of awareness about the exception handling policy. Next, we provide a selection of quotes to illustrate each finding. To ensure participants anonymity, they are identified with the convention P#.

Lack of Awareness. Participants mentioned that they do not have any guideline to assist them in the implementation of exception handling code. They said that there is a lack of documentation about exception handling policy, or they are unaware of it. For this reason, the EH code is usually implemented without the supporting documentation/guideline. In this company, the EH rules were implicitly defined by the architects and not specified anywhere.

> *"Now I realized that we should have a documentation explaining the exception handling policies."* [P8]

Lack of training. Most participants said that they learned exception handling strategies by inspecting the existing code.

> *"When I need to catch an exception, I rely on the code of others. I observe how they raise and handle try to do the same way."* [P5]
>
> *"We didn't have any training. We learn in small steps."* [P3]

Neglect. Moreover, during the interview we could observe the same "ignore-for-now" behavior found on the previous

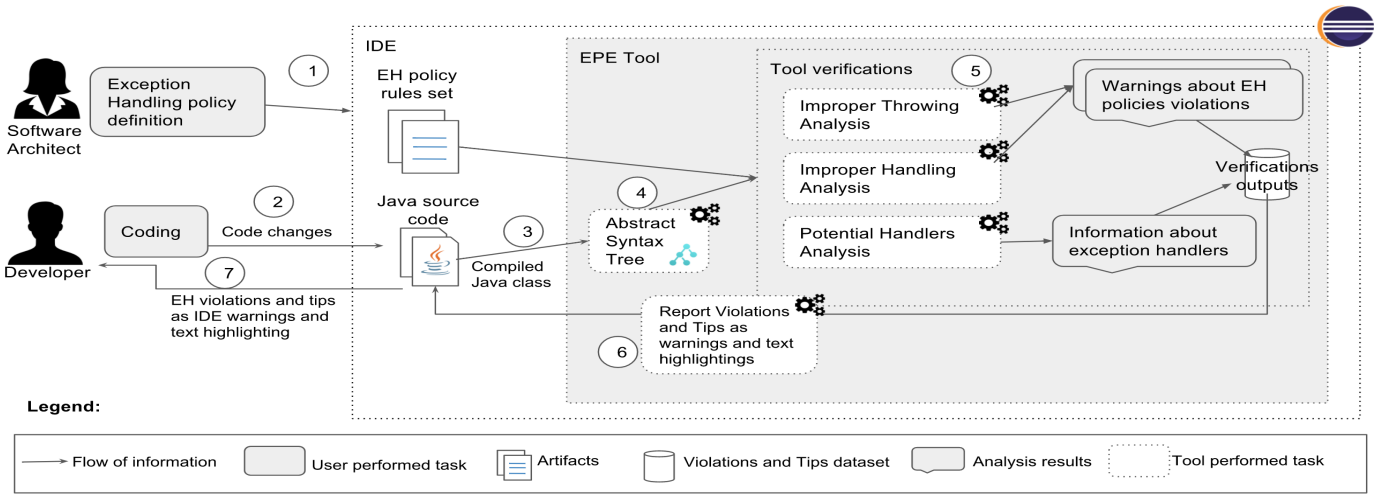

Fig. 1. The overview of the approach supported by EPE.

studies [12], [2] - the developers avoid the EH code until an error occur or when the IDE forces the developer to deal with the exception (either re-throwing or handling in the case of checked Java exceptions)

> *"We care about delivering the product and only think about the happy-path. If there is a problem, we take a note of it and think how to deal with it afterward."* [P7]

> *"I implement the method call and if it is necessary to deal with some exception, I re-throw it."* [P2]

IV. The Exception Policy Expert Tool

In order to tackle the problems presented in the previous section (i.e., developers are usually unaware of the exception handling rules that compose the EH policy of a system), we propose EPE - a tool that makes the exception handling policy explicit to the developers while they are programming and warns them about policy violations related to the code being implemented. The tool is integrated with the IDE (i.e., Eclipse plug-in) and runs at compilation time so that the developers do not need to explicitly invoke it. The EPE tool is an open source project and is available online[1]. The tool works with two input data sources: a file specifying the EH policy and one or more changed Java files. The EPE tool performs a set of intra-method exception handling analysis on the Abstract Syntax Tree (AST) of the changed Java files looking for pieces of EH code (i.e, catch blocks and throw statements) that violate any of the pre-defined EH rules that compose the policy. In case of a violation, the EPE tool (i) presents a warning message detailing why a given piece of code violates a rule; and (ii) highlights the line of code containing the violation. Moreover, the tool also provides exception handling information about the potential handlers of each exception being thrown on the analyzed methods that are related to one or more exception handling policy. The Fig. 1 gives an overview of the proposed approach. Further details on the EH policy definition and the intra-method analyses performed by the tool are detailed next.

A. Policy Definition

The first step of the proposed approach is the definition of the exception handling rules that compose the EH policy of a system. Some DSLs have been proposed to support the definition of the EH policy [9], [6]. Such languages provide elements to define the set of design rules that compose the exception handling policy and have syntactic differences between them.

We chose one of such languages, called Exception Contract Language (ECL) [9], to use in our approach. Each design rule in ECL specifies the handling and signaling capabilities of a method or set of methods. The main elements of ECL are the (i) signaler, that represents a method, class or package that throws exceptions, (ii) exception, that identifies the exception type thrown and (iii) handler, that defines a method, class or package that should be responsible for handling the exception. The following snippet illustrates an example of an EH design rule defined using ECL. For instance, it exemplifies a binding rule that defines that when a *CoreException* is signaled by methods in the *StudentValidation* class, this exception should be handled by methods in the *ViewFilter* class. In our approach, therefore, an exception handling policy is comprised of a set of ECL rules. For the sake of simplicity, in this paper, we will omit some ECL syntax details when presenting examples of ECL rules.

There are three kinds of EH rules: (i) the EH binding rule - specifies what are the elements responsible for handling a list of exceptions thrown by a signaler; (ii) the canonly-signal rule that defines all exceptions that a signaler can throw; and (iii) the cannot-handle rule, that defines a method, class or package that should not handle a given exception.

[1] *https://github.com/lets-ufrn/ExceptionPolicyExpert*

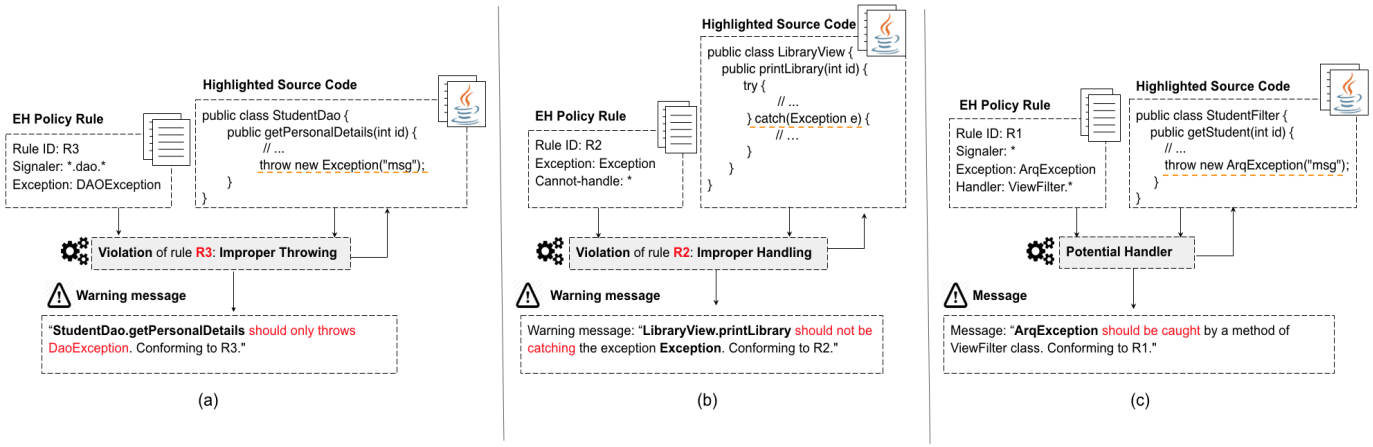

Fig. 2. The intra-method analyses performed by EPE: analyses inputs and provided outputs.

```
ehrule {
    id = <id_rule>
    signaler = StudentValidation.*
    exceptionAndHandlers = {
        exception = CoreException
        handler = [ ViewFilter.* ]
    }
}
```

B. The Source Code Analysis

In order to give an instant feedback to the developer, the tool observes the class compilation event generated by the Eclipse IDE and runs intra-method analyses for each of the modified classes. To do so, the EPE tool works with the Abstract Syntax Tree (AST) representation of changed Java files. After performing the intra-method analyses of the class under edition, the developer will receive visual alerts as warning messages and code highlighting containing the analysis results.

The goal of such the intra-method analyses is to verify the source code (of the changed Java files) looking for EH rule violations or providing tips concerning the exception handling code. Three intra-method analyses are performed: (i) Improper throwing - that looks for methods that raise exceptions in a way that violates one or more rules; (ii) Improper handling - that verifies if some method is handling exceptions in a way that violates one or more rules; and (iii) Potential handlers - that verifies if the policy rules contain information about possible handlers for a thrown exception. Such verifications, that can express a rule violation or just some tip related to the policy, are summarized in Table I and detailed next.

TABLE I
VERIFICATIONS IMPLEMENTED BY EPE TOOL

Verification	Type	Applied to
Improper throwing	Violation	Signalers
Improper handling	Violation	Handlers
Potential handlers	Information/Tips	Signalers

1) Improper Throwing: The goal of this intra-method verification is to alert developers if the piece of code they are implementing is throwing an exception that is not allowed by the EH policy. To do so, the tool checks if the throw statement is part of a method that is referenced by an *onlysignal* ECL rule. Since this rule defines all the exception that the method may throw, if the current code is throwing another exception, a violation is detected. The Fig. 2(a) illustrates such scenario and the Algorithm 1 has the pseudocode performed by the auxiliary methods intra-method analysis.

Algorithm 1 Improper throwing

Require: Source Code of class under edition (SC)
1: $S \leftarrow \emptyset$
2: **for** method M : Program Methods from SC **do**
3: **for** statement S_i : Statements from M_i **do**
4: **if** S_i is a $throw$ statement **then**
5: $E \leftarrow$ GETTHROWNEXCEPTION(S_i)
6: $FR \leftarrow$ GETCANONLYSIGNALRULES(M_i)
7: **for** rule FR_i : FR **do**
8: **if** $E \notin$ GETEXCEPTIONSFROMRULE(FR_i) **then**
9: $S \leftarrow S \cup \{S_i\}$
10: **end if**
11: **end for**
12: **end if**
13: **end for**
14: **end for**
15: return S

The auxiliary methods in the Algorithm 1 are all named with *Get* prefix. Such methods either access the AST or the EH rules (which are stored in a data structure in memory) to get information such as: (i) the exception thrown in a given throw statement (*GetThrownException*); (ii) the list of canonly signal rules related to a method(*GetCanOnlySignalRules*); and (iii) the list of exceptions related to a given rule (*GetExceptionsFromRule*).

2) Improper handling: The goal of this analysis is to warn developers if an implemented catch block is handling an exception that should not be handled in this given method according to the policy. This restriction can be specified in the EH policy using a cannot-handle rule. If this violation is detected, the EPE tool underlines the code causing the violation and presents a message, as illustrated in Fig. 2(b) and detailed in Algorithm 2.

Algorithm 2 Improper handling

Require: Source Code of class under edition (SC)
1: $S \leftarrow \emptyset$
2: **for** method M : Program Methods from SC **do**
3: **for** statement S_i : Statements from M_i **do**
4: **if** S_i is a *catch* statement **then**
5: $E \leftarrow$ GETCAUGHTEXCEPTION(S_i)
6: $FR \leftarrow$ GETRULESFROMSIGNALER(M_i)
7: **for** rule FR_i : FR **do**
8: **for** exception CH : GETCANNOTHANDLEEXCEPTIONSFROMRULE(FR_i) **do**
9: **if** E matches CH **then**
10: $S \leftarrow S \cup \{S_i\}$
11: **end if**
12: **end for**
13: **end for**
14: **end if**
15: **end for**
16: **end for**
17: return S

All auxiliary methods in the Algorithm 2 are also named with *Get* prefix. Such methods also access the AST or the EH rules. The information provided by such methods are as follows: (i) the exception caught in a given catch statement (*GetCaughtException*); (ii) the list of all EH rules where a given method is defined as signaler (*GetRulesFromSignaler*); and (iii) the list of exceptions that are defined on a *cannnot EH rule* (*GetCannotHandleExceptionsFromRule*).

3) Potential handlers: This analysis aims to provide information to the developers about the potential handlers of an exception. Usually, when developers throw an exception, it is not clear what are the points in the code where this exception should be handled (a limitation inherent to the implicit control flow of exceptions which impairs program understandability [4], [12], [2]). Hence, in our approach, if the compiled code throws an exception, the tool looks for rules matching the signaler and the exceptions and informs to the developer what should be its handlers according to the policy. The analysis pseudocode is detailed in Algorithm 3. The tool underlines the piece of code related to the information and presents a message in the IDE, as exemplified in the Fig. 2(c). This information can be useful to developers that do not know where the exception being thrown should be handled.

The auxiliary methods in the Algorithm 3 are also named with *Get* prefix, and access the AST or the EH rules. The information provided by such methods are: (i) the exception

Algorithm 3 Potential handlers

Require: Source Code of class under edition (SC)
1: $S \leftarrow \emptyset$
2: **for** method M : Program Methods from SC **do**
3: **for** statement S_i : Statements from M_i **do**
4: **if** S_i is a *throw* statement **then**
5: $E \leftarrow$ GETTHROWNEXCEPTION(S_i)
6: $FR \leftarrow$ GETRULESFROMSIGNALER(M_i)
7: **for** rule FR_i : FR **do**
8: $H \leftarrow$ GETHANDLERSFROMRULE(FR_i)
9: **if** $E \in$ GETEXCEPTIONSFROMRULE(FR_i) **then**
10: $S \leftarrow S \cup \{H\}$
11: **end if**
12: **end for**
13: **end if**
14: **end for**
15: **end for**
16: return S

thrown in a given throw statement (*GetThrownException*); (ii) the list of all EH rules where a given method is defined as signaler (*GetRulesFromSignaler*); (iii) the list of methods that are defined as handlers on a given rule (*GetHandlersFromRule*); and (iv) the list of exceptions related to a given rule (*GetExceptionsFromRule*).

V. EVALUATION

In order to evaluate the EPE tool in a real development environment, we contacted a software development team of the Informatics Superintendence of Federal University of Rio Grande do Norte (SINFO/UFRN) in Brazil. The SINFO development team is responsible for the development of SIG (i.e., Sistemas Integrados de Gestao), a family of large-scale web-based information systems which perform the full automation of university management activities and that is been deployed in around 20 Brazilian federal institutions. Currently, this development team is composed by 40 developers.

All SIG systems follow the same Model-View-Controller architectural pattern. Table II summarizes some source-code metrics characterizing the size of one of them, which was chosen as our target system.

To conduct our evaluation study, first, we needed to specify the exception handling policy of the target system. Since there was no documentation describing it, we talked to the system architects in order to specify the policy. Together with the

TABLE II
SOURCE CODE METRICS OF THE TARGET SYSTEM

Metrics	Values
LOC	593,276
# of classes	3,841
# of methods	51,408
# of catch blocks	2,912
# of throw statements	1,775

architects, the source code was inspected to identify a set of potential rules. The architects defined the rules in natural language and one of the authors of this work translated such rules to ECL rules - the architects were not requested to define the rules in ECL because it was out of the scope of this study to evaluate ECL usability. The meeting on which the EH policy was defined took around two hours. Table III summarizes such rules.

TABLE III
DEFINED EH POLICY RULES

Rule	Description
R1	signaler = * (i.e., any method) exception = ArqException CoreException DaoException SecurityException RuntimeCoreException handler = ViewFilter.*
R2	handler = * cannot-handle exception = Exception
R3	signaler = *.dao.* (i.e., any method of dao package) canonly-signal exception = DaoException
R4	signaler = *.jsf.* (i.e. ,any method of View layer) canonly-signal exception = ArqException CoreException SecurityException RuntimeCoreException

In the target system, the *ViewFilter* class acts as a gateway between the classes implementing business-specific rules and the GUI layer. R1 specifies that the methods in the *ViewFilter* class are responsible for handling a set of system-defined exceptions: *ArqException*, *CoreException*, *DaoException*, *SecurityException*, and *RuntimeCoreException*.

The R2 rule specifies that none of the methods in the target system can implement a general handler. That is, no method in the target system can implement a catch block declaring as its argument the *Exception* type; instead, developers are expected to implement a specific handler. The R3 rule indicates that the data-access layer of the target system may only throw instances of *DaoException*, whereas the R4 rule defines that the view layer may only throw four specific system-defined exceptions: *ArqException*, *CoreException*, *SecurityException*, and *RuntimeCoreException*.

After defining the exception handling rules, we contacted the development team and a group of 6 developers volunteered to participate in our study, agreeing to use the tool on their daily regular coding tasks. The EPE tool was installed in their Eclipse IDE and we gave them a 20-min lecture about the tool, covering its motivation, purpose and main concepts and features. We also prepared some tutorial material in case they had any doubt. It is worth mentioning that they received the exception handling policy of the target system fully specified, so they only had to use the EPE tool. Then, they used the tool on a daily basis in their regular coding tasks - the tool was invoked every time the code was compiled.

Each participant used the tool from 3 to 8 working days - due to company's internal issues the developers could not start using the tool on the same day. Overall, we collected the EPE usage logs (e.g., frequency and instances of policy violations and tips) of 30 working days and analyzed them. During that period, the tool detected a total of 388 distinct occurrences of violations or tips: 12 improper throwing violations, 202 improper handling violation, and 174 tips about potential handling sites. In other words, the EPE found 388 points of the code that had a relation to the pre-defined EH policy and could give information about them to the developer, approximately 7 violations and 6 tips per person per day.

Table IV summarizes the number of violations and tips related to each rule. R2 was the most violated rule; this rule indicates that the generic exception type *java.lang.Exception* cannot be handled by the methods in the target system.

TABLE IV
OCCURRENCES DETECTED BY THE TOOL DURING EVALUATION

Rule	Occurrences	Verification type
R1	174	Potential handler
R2	202	Improper handler
R3	10	Improper throwing
R4	2	
Total	388	-

In 202 distinct points in the changed source code, a generic handler was detected. Handling an overly-general exception type, such as the *java.lang.Exception*, is a known exception handling antipattern [13], and is also the cause of recurring exception handling-related faults reported in the literature [14], [7]. It is important to emphasize that these results are relative to the classes edited by developers during the tool evaluation period. The tool did not analyze the whole source code of the target system since it works based on the compilation of the code being edited on the IDE. Thus, the actual number of exception handling policy violations in the target system can actually larger than what we have observed.

A. Developers' Feedback

After we had analyzed the logs of the EPE tool, we contacted the participants to get their impressions about the tool usage. We applied a questionnaire comprised by 2 open-ended questions, 2 multiple choice questions, and 9 Likert-scale questions. The respondents could complete the questionnaire in about 10 minutes. We used open coding technique [11] to iterate through the open-ended responses.

Next we provide a summary of the developers' responses and a selection of quotes to illustrate the developers' main impressions. To ensure participants anonymity, they are identified with the convention P#.

1) Support EH Policy Comprehension: Almost all the developers mentioned that the tool helped them to understand the target system exception handling policy, either by highlighting the code that was violating an existing rule or by presenting information concerning the potential handlers of an exception

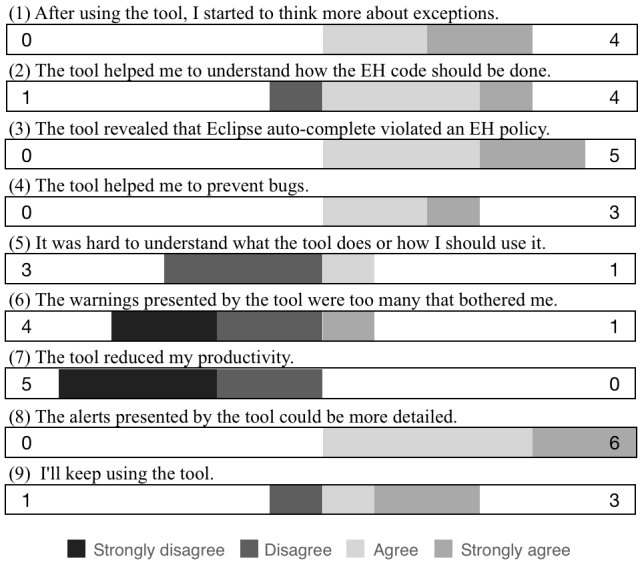

Fig. 3. Developer's responses to Likert-scale questions.

being thrown. The Fig. 3 illustrates the developer's responses to some questions related to the EH policy comprehension - see questions (1) and (2). Iterating over the open-ended responses, we could observe that some developers also recognized that the warnings about the exception handling rules made clear the EH signaling and handling responsibilities of each layer of the system, helping them to understand how exception handling should be done in the system.

> *"The tool helped me to implement the EH code. The tool feedback helped me to think about the exception handling responsibilities of each layer."* [P1]

In addition, the developers were also asked if the tool helped them to implement the code according to the system exception handling policy, as illustrated by the Fig.4. 5 out of 6 developers agreed that the tool alerted them to at least one improper handling. At the same time, only few developers affirmed that the tool warned them about an improper throwing exception.

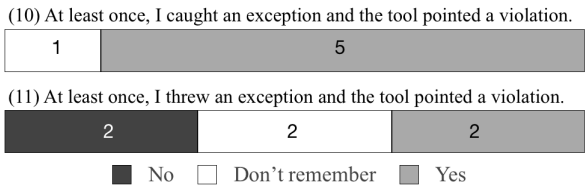

Fig. 4. Developer's feedback about EH violations visualization.

2) Be Careful with Quick-Fix IDE Recommendations: Every time an exception is thrown in Eclipse IDE Java editor, the IDE provides two quick-fix recommendations with options to: (i) automatically declare the exception on the methods' signature; or (ii) automatically surround the method throwing the unhandled exception with a try-catch block. Almost all developers mentioned that at least once these IDE suggestions violated one of the exception handling rules defined for the system (e.g., by automatically creating a try-catch block for an exception that should be handled elsewhere). The question (3) of Fig. 3 illustrates the developer's responses about the Quick-Fix IDE recommendation. Moreover, one of the developer P2 emphasized this issue in one of the open-ended questions.

> *"If I had used the tool since the beginning (of the project) I would have learned of how to use exceptions in a better way, or at least I would have known that I some coding I doing was wrong (IDE-auto complete)."* [P2]

3) Enabled to Prevent Bugs: On the open-ended questions, some developers mentioned that the tool helped on bug prevention. They mentioned that they were used to think about exceptions only when they need to fix a bug related to the exception handling code (e.g., uncaught exceptions). However, after using EPE tool they started to think about exceptions during the development routine, and doing so they could prevent bugs.

> *"The tool showed me that an exception that I was about to throw should not have been used anymore."* [P3]

Moreover, the responses to one of the Likert-scale questions (see question (4) of Fig. 3) show that half of the participants agreed that the tool could prevent faults on the exception handling code.

4) Overall Impressions: Questions (5), (6), (7), (8) and (9) of Fig. 3 summarizes the developers' general impressions after using the EPE tool. Half of them think that it is not hard to use and understand the tool. Most of the developers disagreed that the tool presents too many warnings. Iterating over the responses to the open-ended questions, we could observe that some developers mentioned that current IDEs may indeed show a plethora of warnings - and in such cases, the developers may not pay attention to them. But since the tool highlighted the code that was breaking a rule it could get his/her attention.

Moreover, almost all the developers disagreed that the tool could negatively impact productivity. All the developers also agreed that the tool should improve the warning messages in order to include further details about the problem and how to solve it. Some of the developers also mentioned such the limitation in one of the open-end questions. However, despite the limitations, half of developers would like to keep using the tool.

> *"The tool could suggest good practices for solving the problem and show possible solutions."* [P5]

After conducting this case study, we could observe through developers' feedback that EPE tool could bring the exception handling policy to the implementation routine. Based on the definition of an exception handling policy and a set of

intra-method analysis it was possible to confront the code to the policy and provide feedback to the developer. Both the quantitative and qualitative evaluation illustrate promising results of the proposed tool.

B. The Study Data

The data produced in the study is publicly available at the *Exception Policy Expert* (EPE) tool website hosted on GitHub[2]. Specifically, we provide: (i) the questions that guided the semi-structured interviews; (ii) the EPE tool source-code; (iii) the data collected during the case study - the list of exception handling design rules, the number of violations, the questionnaire submitted to the developers and their responses.

VI. Discussion and Lessons Learned

The EH policy made explicit. The EPE tool provides a way of making the exception handling policy explicit to developers all the time while they are programming. By using the EPE tool, developers could rapidly discover whether an EH code violated or adhered to the exception handling policy. Moreover, the EPE tool also helped developers to perceive that the default quick-fix recommendations provided by their IDE often violated the exception handling policy defined for the target system.

Different Types of EH Bugs. The lack of EH policy is not the only source of bugs in the EH code. The EPE tool can warn the developer if a piece of code violates a previously defined exception handling policy. However, it does not look for other kinds of bugs on the exception handling code, such as: looking for empty catches or null-dereferences. The EPE tool can only be used to express rules related to exceptions that are explicitly signaled (i.e., using throw statement). Hence it cannot be used to express rules related to exceptions that are implicitly signaled by the JVM due to violations of semantic Java constraints (e.g., java.lang.NullPointerException, java.lang.ArrayIndexOutOfBounds). Other static analysis tools (which could detect these other types of EH bugs) could, therefore, be used in combination with EPE tool.

External Validity. Both the semi-structured interviews and the tool evaluation were conducted on a single company and on a single development team implementing a single project. Although the findings of both studies cannot be generalized they represent initial findings of the approach usefulness in a real development context - involving experienced developers and a real industrial Java application.

EH Policy Definition. The proposed approach is based on the definition of the EH policy [9], [6]. During the case study, we could get the architects' opinions about the most important EH rules - the ones that, in their point of view, should be obeyed by all developers. One may argue that such specified set of rules may not represent the whole EH policy. However, since our goal was to evaluate the ability of the proposed approach to make developers more aware of EH rules that compose the policy, to use a potential subset of the rules that compose the EH policy does not weaken our findings. The case study revealed that the developers could be warned about EH rules violations. By doing so, the approach tends to prevent such EH bugs to be revealed later in the production environment or even silenced and hardly discovered (i.e., when the exceptions are silenced/swallowed) [13].

Language used to specify the EH rules. The current version of the tool has a built-in parser for ECL. It could be used to express all EH rules suggested by the system architects during the case study. However, it was out of the scope of this study to evaluate ECL expressiveness and usability. It is worth mentioning that the tool can be extended to support other DSLs that enable the definition of signaling and handling rules such as the DSL proposed by Barbosa et al. [6].

VII. Related Work

How Developers deal with the EH Code. Previous studies have investigated how developers deal with the exception handling code of a system [12], [2]. Shah et al. [12] interviewed nine Java developers to gather an initial understanding of how they perceive exception handling and what methods they adopt to deal with exception handling constructs while implementing their systems. Such study revealed that developers tend to ignore the proper implementation of exception handling in the early releases of a system, postponing it to future releases, or until defects are found in the EH code. In a subsequent study, Shah et al. [2] interviewed more experienced developers in order to contrast the viewpoints of experienced and inexperienced developers regarding exception handling. The authors observed that although experienced developers consider the EH code indistinguishable or inseparable from the development of the programs main functionality, they recognize that also adopted the "ignore from now" approach previously in their career. Such works focused on implementation issues of exception handling code, they did not investigate how developers deal with the exception handling policy during development. In the first step of this work we conducted a set of interviews to investigate how developers deal with the exception handling policy of a system.

Faults in Exception Handling. Although exception handling mechanisms have been proposed as means to improve software robustness, empirical studies showed the recurrent failures in software systems were caused by faults in exception handling code. The studies conducted by Barbosa et al. [14] and by Ebert et al. [7] analyzed exception handling-related failures reported in open-source projects and proposed categorizations of exception handling faults. Our solution can identify some of the categories proposed by Barbosa et al. and by Ebert et al. For example, the verifications presented in Section IV can identify faults categorized as: "Exception caught at the wrong level", "Lack of a handler that should exist", "Exception not thrown", "Throwing wrong type", among others. Therefore, our solution has great potential to assist developers in not introducing faults in exception handling code.

Understanding Exception Propagation. Early works in the exception handling literature focused on assisting developers

[2] *https://github.com/lets-ufrn/saner2018*

in understanding the propagation of exceptions in the source code [15], [16], [17]. These solutions statically compute the paths where exceptions flow in the source code from raising sites to handling sites. Their main limitation is the lack of explicit exception handling policies, thus making developers responsible for understanding the implemented exception handling code and identifying where violations are located. Our work complements these solutions with the notion of exception handling policies and exception handling violations. We believe that these solutions also complement ours in the sense that we could borrow some of their visualizations to graphically display the violations identified by our tool apparatus.

Exception Handling Policy Compliance Checking. Although design rules have long been used for architecture compliance checking [18], only recently works in the exception handling literature employed design rules for exception handling policy compliance checking. Abrantes and Coelho [9] proposed a domain-specific language (DSL) for specifying exception handling policies in Java systems. Their solution dynamically verifies the exception handling policy by monitoring at runtime the program execution. Whenever the specified policy is violated, notifications are sent to a remote server, where they are stored and later analyzed. Similarly, Barbosa et al. [6] and Juarez Filho et al. [19] also proposed DSLs for specifying exception handling policies in Java systems, but their solutions statically verify the specified policies at build time. Our work is similar to theirs in the sense that we all specify exception handling policies. However, we continuously verify exception handling policies during development tasks, as soon as the code is saved and compiled. Therefore, our solution provides rapid feedback about violations in the source code, allowing developers to repair these violations in early phases of the development process.

Recommendation Systems for Exception Handling Coding. Recommender systems are also being explored for assisting developers implementing the exception handling code of their systems. The solutions proposed by Barbosa et al. [20] and Rahman and Roy [21] assist developers in implementing handling actions in catch blocks of Java systems. Their solutions mine source code repositories and use search strategies to recommend code snippets with handling actions that are potentially relevant for a given system being implemented. Thus, developers may adapt and use such code snippets to implement their own exception handling code. Our works are similar in the sense that we aim to assist developers implementing exception handling code. If on one hand, their works focus on suggesting handling actions based on code snippets, our work checks whether or not the exception handling code adheres to specified policies. Thus, we believe our solutions are complementary.

Assistance on Repairing Violations. Recently, Barbosa and Garcia [22] proposed a recommender system able to assist developers in repairing exception handling violations. Given a policy violation, their solution is able to recommend a sequence of modifications to be performed in the source code in order to repair the violation. We believe that their work complements our work by assisting developers in the process of repairing violations in the source code. This is similar to just one of the analysis of our approach that recommends which exceptions developers should use in a given handler. Moreover, Barbosa and Garcia's solution is not integrated to the IDE, so it does not continuously provide feedback to developers, as our solution does.

VIII. Concluding Remarks

In this work we presented EPE tool, embedded on the IDE (i.e., Eclipse plug-in), that makes the exception handling policy explicit to the developer while s/he is implementing the exception handling code. The tool warns developers about policy violations related to the changed code and also provide suggestion of the potential handlers for the exception been thrown. This tool is based on three intra-method analysis that checks the program AST against a set of rules comprising the main handling and signaling capabilities of the application modules.

One case study was conducted to evaluate the proposed tool in a real development context. The results have shown that the proposed approach could detect several of EH policy violations during development. Overall the tool was used in 30 working days and could detect 214 distinct EH violations (i.e., 202 *improper handling* violations and 12 *improper throwing* violations). Moreover, the tool provided 174 tips about potential handling sites - points in the code where the exception being thrown should be handled. Moreover, the developers who used the tool were contacted and their opinions collected. The results of the quantitative and qualitative evaluation performed in this study indicate that the tool could indeed be used to make the exception handling policy of a system explicit to the developer.

Acknowledgment

We would like to thank Leandro Beserra for his support in conducting the case study at SINFO/UFRN, and the software architects and developers that participated in this study. This work is partially supported by CNPq - Proc. 459717/2014-6 and INES - grant CNPq/465614/2014-0.

References

[1] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[2] H. Shah, C. Gorg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 150–161, 2010.

[3] F. Castor Filho, A. Garcia, and C. M. F. Rubira, "Extracting error handling to aspects: A cookbook," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 134–143.

[4] S. Sinha and M. J. Harrold, "Analysis of programs with exception-handling constructs," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 348–357.

[5] B. Cabral and P. Marques, "Exception handling: A field study in java and. net," in *European Conference on Object-Oriented Programming*. Springer, 2007, pp. 151–175.

[6] E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus, "Enforcing exception handling policies with a domain-specific language," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 559–584, 2016.

[7] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.

[8] J. Viega and J. Vuas, "Can aspect-oriented programming lead to more reliable software?" *IEEE software*, vol. 17, no. 6, pp. 19–21, 2000.

[9] J. Abrantes and R. Coelho, "Specifying and dynamically monitoring the exception handling policy." in *SEKE*, 2015, pp. 370–374.

[10] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," *ECOOP'97Object-Oriented Programming*, pp. 85–103, 1997.

[11] K. Charmaz, "Constructing grounded theory: A practical guide through qualitative research," *SagePublications Ltd, London*, 2006.

[12] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in *Proceedings of the 4th international workshop on Exception handling*. ACM, 2008, pp. 62–68.

[13] J. Bloch, *Effective java*. Pearson Education India, 2008.

[14] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa, "Categorizing faults in exception handling: A study of open source projects," in *Software Engineering (SBES), 2014 Brazilian Symposium on*. IEEE, 2014, pp. 11–20.

[15] B.-M. Chang, J.-W. Jo, and S. H. Her, "Visualization of exception propagation for java using static analysis," in *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*. IEEE, 2002, pp. 173–182.

[16] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in java server applications," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 230–239.

[17] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 2, pp. 191–221, 2003.

[18] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*. IEEE, 2007, pp. 12–12.

[19] L. Juarez Filho, L. Rocha, R. Andrade, and R. Britto, "Preventing erosion in exception handling design using static-architecture conformance checking," in *European Conference on Software Architecture*. Springer, 2017, pp. 67–83.

[20] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *Software Engineering (SBES), 2012 26th Brazilian Symposium on*. IEEE, 2012, pp. 171–180.

[21] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 285–294.

[22] E. A. Barbosa and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," *IEEE Transactions on Software Engineering*, 2017.

[23] F. Cristian, "Exception handling and software fault tolerance," in *Reliable Computer Systems*. Springer, 1985, pp. 154–172.

[24] B. Jakobus, E. A. Barbosa, A. Garcia, and C. J. P. de Lucena, "Contrasting exception handling code across languages: An experience report involving 50 open source projects," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 183–193.

[25] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[26] M. Greiler, A. van Deursen, and M.-A. Storey, "Test confessions: a study of testing practices for plug-in systems," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 244–254.

[27] M. P. Robillard and G. C. Murphy, "Designing robust java programs with exceptions," in *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6. ACM, 2000, pp. 2–10.

[28] L. Singer, F. Figueira Filho, and M.-A. Storey, "Software engineering at the speed of light: how developers stay current using twitter," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 211–221.

[29] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 112–121.

[30] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," *Journal of systems and software*, vol. 59, no. 2, pp. 197–222, 2001.

[31] D. Mandrioli and B. Meyer, *Advances in object-oriented software engineering*. Prentice-Hall, Inc., 1992.