Specifying and Dynamically Monitoring the Exception Handling Policy

Joilson Abrantes
Dep. of Informatics and Applied Mathematics
Federal University of Rio Grande do Norte
Brazil
joilson@ppgsc.ufrn.br

Roberta Coelho
Dep. of Informatics and Applied Mathematics
Federal University of Rio Grande do Norte
Brazil
roberta@dimap.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). Such policy is usually undocumented and implicitly defined by the system architect. For this reason, developers may think that by just including catch-blocks in the code they can deal with exception conditions. This lack of information may turn the exception handling into a generalized "goto" mechanism making the program more complex and less reliable. This work proposes a domain-specific language called ECL (Exception Contract Language) to specify the exception handling policy and a runtime monitoring tool which dynamically checks this policy. The monitoring tool is implemented in the form of an aspect library, which can be added to any Java system without the need to change the application source code. We applied this approach to a large-scale web-based system and to a set of versions of the well-known JUnit framework. The results indicate that this approach can be used to express and to automatically check the exception handling policy of a system, and consequently support the development of more robust Java systems.**

Keywords - exception handling; monitoring; dynamic analysis.

I. INTRODUCTION

Modern applications have to cope with an increasing number of abnormal computational states that arise as a consequence of faults in the application itself (e.g., access of null references), noisy user inputs or faults in underlying middleware or hardware. The exception handling (EH) mechanism [24] is one of the most frequently used techniques for developing robust systems, enabling such applications to detect and recover from these exceptional conditions.

Although exception handling mechanisms have been embedded in several mainstream programming languages (e.g. Java, C++, C#), studies have shown that the exception handling code is often poorly understood and the least-tested part of software systems [13], [14], [15], [16]. The exception handling behavior of a system is poorly understood, because it is generally spread over several implementation artifacts, and often the exception handling constructs (e.g., throw statements and try-catch blocks) lead the developers into believing that by just including EH constructs in the code that they can (i) deal with exceptional situations, and (ii) focus on the development of "happy path" scenarios [17]. This "ignore-for-now" approach may turn the exception handling into a generalized "goto" mechanism [17] making the program more complex and even less reliable. As a consequence they may negatively affect the system, favoring the introduction of failures such as uncaught exceptions [30], [20] - which can lead to system crashes, making the system even less robust [5].

Work has shown that the lack of information about how to design and implement exceptional conditions leads to complex and spaghetti-like exception structures [12]. To the best of our knowledge, few studies have been proposed to better understand and check the exception handling behavior of systems. Some of them are based on the use of static analysis tools [20][21][22] and others on automated testing tools [1][7]. Both approaches, however, have intrinsic limitations.

The static analysis approaches [20][21][22] propose tools to discover the paths that exceptions take from signalers (i.e., elements that throw exceptions) to handlers (i.e., elements responsible for catching them). However, due to the limitations inherent in static analysis approaches combined with the characteristics of modern languages (e.g., inheritance, polymorphism and virtual calls) such approaches usually report many false positives. On the other hand, approaches based on the definition of test cases [1][7] limit the ability of checking exception handling behavior to the execution of test scenarios. Moreover, the high number of signaling and handling sites to be tested may lead to the test explosion problem [18][19].

None of the work mentioned above enables the developer to specify the exception handling behavior of a system and to check such behavior while the system is running on its production environment. Doing so, we could use the input data provided by real users or acceptance testers in order to check the exception handling behavior of a system.

This work proposes a domain-specific language (DSL) to specify the exception handling policy of a system (i.e., a set of design rules that specify the exception handling behavior of a system such as, how exceptions should be handled and thrown by its main elements). More often than not, such policies are not documented, and usually remain implicit in the form of a set of exception handling constructs spread over implementation artifacts. Moreover, this work also provides a runtime monitoring tool which verifies whether or not the exception handling behavior of a system is in accordance with the handling policy defined beforehand.

To evaluate the proposed language and the supporting tool, we conducted two case studies. We specified and checked the

(DOI reference number: 10.18293/SEKE2015-133)

exception handling policy of large-scale web system (SIRH) – which contains 593 KLOC of Java source code, 14781 throw statements and 2912 catch-blocks - and the well-known JUnit testing framework - for which four releases were evaluated. Results have shown that the proposed approach could be used to specify and dynamically check the exception handling policy of both systems. The contribution of this work is two-fold:

- It introduces a domain-specific language to specify the exception handling policy of a system.
- It presents a runtime monitoring tool implemented to support the dynamic check of the exception handling policy.

The remainder of this paper is organized as follows: Section II presents the main concepts related to this work; Section III presents a motivating example for using the proposed approach; Section IV presents the domain-specific language to define the exception handling policy and the supporting tool which checks such rules during runtime; Section V presents the case studies conducted in this work, and finally, Section VI presents the conclusions and future work.

II. THE EXCEPTION HANDLING MECHANISM

In order to support the reasoning for exception handling behavior of a system we present the main concepts of an exception-handling mechanism. An exception handling mechanism is comprised of four main concepts (i.e., the exception, the exception signaler, the exception handler, and the exception model - that defines how signalers and handlers are bound [14]) and two supporting concepts (i.e., the exception types and the exception interface) described next.

Exception Raising. An exception is raised by a method when an abnormal state is detected. In Java an exception is thrown using the throw statement [23].

Exception Handling. The exception handler is the code invoked in response to a raised exception. It can be attached to protected regions (e.g. methods, classes and blocks of code) [14]. In Java the handler is represented by the try-catch block [23].

Handler Binding. In many languages as in Java, the search for the handler to deal with a raised exception occurs along the dynamic invocation chain. This is claimed to increase software reusability, since the invoker of an operation can handle it in a wider context [11].

Exception Interfaces [11]: The caller of a method needs to know which exceptions may cross the boundary of the called method. In this way, the caller will be able to prepare the code beforehand for the exceptional conditions that may happen during system execution. For this reason, some languages provide constructs to associate a method's signature with a list of exceptions that this method may throw. However, they are most often neither complete nor precise [20], because languages such as Java provide mechanisms to bypass this mechanism by throwing a specific kind of exception, called *unchecked exception*, which does not require any declaration on the method signature.

Exception Types. Object-oriented languages usually support the classification of exceptions into exception-type hierarchies. The exception interface is therefore composed of the *exception types* that can be thrown by a method. Each handler is associated with an *exception type* that specifies its handling capabilities - which exceptions it can handle. In Java, exceptions are represented according to a class hierarchy, in which every exception is an instance of the `Throwable` class [23].

III. MOTIVATING EXAMPLE

Consider a layered-information system structured in three layers: the data layer (which accesses the database); the business layer and the presentation layer. One of the exception handling design rules that could be defined in this system is the following: ***the exceptions thrown by the Data layer should be a subtype of `DAOException` and should be handled in the Presentation layer.*** However, usually such rules are informally defined in the system documentation or, more often than not, remain undocumented as an implicit knowledge of the development team.

Both ways of dealing with the exception handling rules threaten the development of robust systems. Firstly, once documented such documentation may become outdated and be of little use. Secondly, the undocumented rules may become unknown for new members of the development team, and as a consequence, will not be followed. Moreover, none of these scenarios support the automatic check of such rules during system compilation or execution.

Let's consider that in such a system an instance of `DAOException` is thrown by the Data layer and is mistakenly handled by a generic handler defined in the Facade class (defined in the Business layer). *How can we check if the aforementioned rule is obeyed?* The approach shown in Section IV enables the developer to define and check such an exception handling rule.

IV. THE PROPOSED APPROACH

We propose an approach based on a DSL (Section IV-A) and a dynamic analysis tool (Section IV-B) to enable developers to define and verify the exception handling behavior of a system. More specifically, this approach allows the developer to create design rules for the exceptional flow, and check if such rules related to the exception handling code are neglected during the application execution.

A. The Exception Contract Language

We propose a domain-specific language called ECL (Exception Contract Language) whose main goal is to allow the creation of design rules for the exception handling behavior. Figure 1 partially illustrates the grammar of ECL language in BNF. In this version of BNF used, non-terminal symbols are written in bold, terminals are written with capital letters. In addition, the {} indicates zero or more repetitions of A. In order to simplify the reasoning of the grammar we omitted the definition of terminals such as ModID (which refers to a name of any identifier).

```
S: Rule
Rule: signaler QualifiedNameWithWildcardSignaler
      exception SetOfNames
      handler SetOfNames ;
SetOfNames: QualifiedNameWithWildcard {:
      QualifiedNameWithWildcard }
QualifiedNameWithWildcardSignaler: QualifiedNameWithWildcard  | *
      QualifiedNameWithWildcard: QualifiedName |
      QualifiedName+ | QualifiedName* | QualifiedName(..)
      QualifiedName: ModID{.ModID}
```

Figure 1. Exception Contract Language (ECL) in Backus-Naur Form notation.

The main elements of ECL are:

- **signaler**: this element represents a method, class or package which can throw one or more types of exceptions.
- **exception**: identifies the types of exceptions thrown by the signaler.
- **handler**: this element represent the methods, classes or packages that will be responsible for handling the types of exceptions set to be launched by the signaler.

Figure 2 shows an example of an exception handling design rule created using ECL. This rule specifies that an exception of type `BusinessException` launched by `login(..)` method defined in `SignSystemBean` class, must be handled by any method defined on `LoginFilter` class.

```
signaler { br.com.ufrn.login.web.SignSystemBean.login(..) }
        exception {br.com.ufrn.arq.business.BusinessException}
            handler { br.com.ufrn.arq.filters.LoginFilter.* };
```

Figure 2. Example of ECL design rule.

The ECL language also supports the use of wildcards. The first is ∗: it matches any series of characters that can appear in a Java identifier. So, for example, in Figure 1 it matches all methods defined in `LoginFilter` class. The second is + wildcard, which can be combined over types. It means 'match any subtype'. In Figure 1 we could add + to the exception name, and in doing so the contract would be related to `BusinessException` and its subtypes. We developed an Eclipse plug-in using XText framework to support the definition of design rules in ECL[1].

B. Dynamic Analysis of Exception Handling

A runtime monitoring tool was developed to check such rules while the program is running. This tool works in the background, analyzing whether defined design rules are being neglected. If a rule is not obeyed, a notification is sent to a remote server which will store the non-compliance. The remote server that receives such data, stores the notifications and provides this information to other applications which can generate EH reports, and mine such data.

[1] The language manual and the Eclipse plug-in is available at: https://bitbucket.org/jvidalabrantes/daeh-tool/wiki.

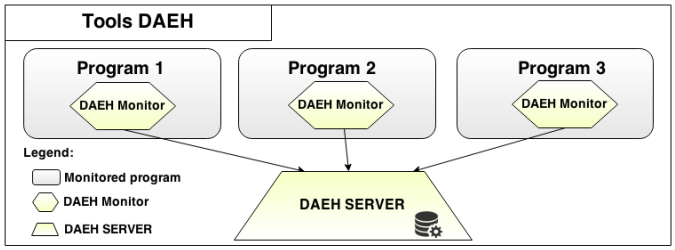

Figure 3. DAEH architecture.

The monitoring tool is called DAEH (Dynamic Analysis of Exception Handling), and consists of a set of monitors responsible for monitoring any Java application and a central server responsible for receiving the notifications from monitors and storing them. Figure 3 illustrates DAEH architecture. This architecture enables the implementation of other applications that communicate with the DAEH server which may query the monitoring information and perform any kind of data analysis.

C. DAEH Monitor

The DAEH monitor is added to the application to be monitored and performs the verification of exception handling design rules defined using ECL [2]. Such a monitor is implemented as an aspect library which is combined at load-time with the application code to be monitored. This library was implemented using AspectJ and load-time weaving [8]. Since the monitor instrumentation is performed when the application classes are loaded into the Java Virtual Machine, there is no need to change the application source code. During the load-time weaving the DAEH monitor (i) loads the exception-handling design rules file and (ii) instruments every place where an exception is handled (every catch block). Hence, every time an exception is handled inside the system the DAEH monitor checks whether or not the handling action is breaking one of the existing exception handling design rules.

V. Case Studies

This approach was used in two case studies: SIGRH - an enterprise large-scale web-based system developed in Java and the well known JUnit framework (from which 4 releases were used). Table I illustrates the characteristics of both systems.

TABLE I. Metrics of Systems

Metrics	SIGRH	JUnit 4.6	JUnit 4.7	JUnit 4.8	JUnit 4.9
LOC	593.276	13098	14049	14373	15684
# of classes	3841	268	290	293	308
# of methods	51408	1724	1853	1885	2041
# of catch-blocks	2912	156	152	153	164
# of throw statements	1775	110	122	123	131

Since SIGRH had no exception design rules explicitly documented, we needed to talk with the system architect in

[2] The ECL manual and plug-in and DAEH tool is available at: https://bitbucket.org/jvidalabrantes/daeh-tool/wiki.

order to document the exception handling policy in the form of ECL rules. As a result of this talk, five main exception handling design rules were documented. Table II illustrates one of them. This rule states that instances of `BusinessException` thrown by any method should be handled by any method of `ViewFilter` class.

TABLE II. EXAMPLE OF DESIGN RULE CREATED FOR THE MONITORED SYSTEM

Id	Exception Handling Design Rule
3	*signaler { * }* *exception { br.ufrn.arq.erros.BusinessException }* *handler { br.ufrn.arq.web.ViewFilter.* };*

After defining the rules, we added the DAEH monitor to the application server on which the SIGRH system was running for acceptance tests. In a 5-day period, the DAEH server received 12,027 notifications of broken design rules. Table III shows the number of violations per design rule.

TABLE III. NUMER OF DESIGN RULES VIOLATIONS (DRVS).

SIGRH		JUnit	
Contract Id	***# DRVs***	***Version***	***# DRVs***
1	6	4.6	0
2	0	4.7	0
3	12,015	4.8	0
4	6	4.9	0
5	0	-	-

As can be seen, only three design rules were violated (i.e., rules 1, 3 and 4). Figure 4 illustrates one of such notifications. Analyzing the violations associated with design rule 3 we observed that all of them were caused by 8 handlers defined in different locations outside the `ViewFilter` class (specified in the design rule illustrated in Table II). Such violations occurred often (i.e., 12,015) because the same pieces of code were exercised more than once during acceptance testing.

<Exception:class br.ufrn.arq.erros.BusinessException >
expected: <Handlers:[br.ufrn.arq.web.ViewFilter.]>*
but was <Handler: UserMBean. login()>

Figure 4. Design rule violation message.

The proposed approach was also used to define and monitor the exception handling design rules of the JUnit testing framework. The design rules were defined manually by inspecting the source code of the framework. Manual inspection was needed because the JUnit documentation had no reference to the exception handling policy adopted in the framework. This task was possible since JUnit is a small-scale framework and very well structured. As a result of this task we were able to define 9 exception handling design rules. Table IV illustrates two of them.

TABLE IV. EXAMPLE OF SET DESIGN RULES FOR JUNIT

Exception Handling Design Rule
signaler{ org.junit.experimental.max.MaxHistory.readHistory(..) } *exception {org.junitt.max.CouldNotReadCoreException}* *handler {org.junit.experimental.max.MaxHistory.forFolder(..)};*
*signaler { * }* *exception {junit.framework.AssertionFailedError}* *handler {junit.framework.TestResult.* : junit.tests.*};*

In order to exercise the framework and to check for exception design rule violations, we ran the test suite that comes with the framework and added the DAEH monitor to the JVM were the tests were executed. Although the rules where defined for version 4.6 of JUnit, we used the same set of rules to check the exception handling behavior of a set of subsequent versions (i.e., 4.6, 4.7, 4.8 and 4.9). Our goal was to check whether there had been changes in the exceptional behavior as the framework evolved. To our surprise, none of the specified contracts broke across the subsequent versions of JUnit. Such behavior can be explained by the fact that JUnit is a very stable framework and that although the exception handling design rules are not explicitly documented, they are adequately maintained by the development team.

VI. DISCUSSIONS

Exception handling policy: a global design problem. The definition of the exception handling policy is a global design problem [12]. However, none of the languages which have embedded EH mechanisms provide a way to specify and check such a policy. Due to this lack of guidance developers, tend to focus their design activities on the normal behavior of the application [2], [3] and forget the exceptional behavior design [4]. In this work, we propose a language to express the exception handling policy of a system in the form of simple design rules, which link signaling and handling sites. Such sites can be methods, classes or packages. The ECL language and the supporting monitoring tool proposed in this work are the first step towards providing an infrastructure to help developers in specifying and analyzing the exception handling behavior of a system as a whole.

Limitations of the proposed approach. The way the exception handling policy is expressed in ECL could be improved to (i) specify the handling action (i.e., what should be performed inside the try-catch block) or to (ii) express a complete set of rules (i.e., if no rule is specified for a signaler no exceptions are allowed). However, the current grammar was sufficient to express all exception handling design rules needed during the execution of the case studies.

VII. RELATED WORK

Two approaches [1][7] extended the JUnit testing tool to support the definition of automated tests for the exception handling behavior. The limitations of both approaches are two-fold: the developer needs to manually implement each test case, and each test case focuses on one single exception flow (i.e. throw-catch pair) at a time. Since most Java systems may contain dozens or even hundreds of exception flows it is hard to choose which ones should be tested. Our approach tackles this limitation since the exception handling design rules involve

higher level modules than single methods (i.e., classes or packages), enabling the checking of several flows at a time.

Terra and Valente [9] proposed a dependency constraint language for specifying acceptable and unacceptable relations among the elements of a system architecture. Such restrictions are statically checked in order to detect the points in the source code that violate the defined relations. This language allows the developer to define which exceptions a given module (i.e., method, class or package) can throw. However, it does not address the handling capabilities of modules nor how handlers and signalers can be bound. Our approach supports the specification of both handling and signaling design rules and check such rules at runtime.

Brunet and Guerrero [10] proposed a tool called DesignWizard that enables the developer to define design rules in the same programming language of the analyzed application, in the form of a set of JUnit test cases. Although such a tool extends the JUnit framework, the checking of design rules is performed statically based on ASM framework. DesignWizard does not support the definition of design rules related to exception signaling and handling capabilities nor how they are bound.

Jin et al proposed JavaMOP [25] a monitoring framework specific to Java programs. [20]. JavaMOP allows the definition of properties based on event specifications and generates AspectJ code for monitoring - weaved into the target program in compile time. When a specification is validated or violated, user-defined actions are executed. User-defined actions can be any Java code from logging to runtime recovery. Our approach differs form JavaMOP as our approach is specific to the monitoring and checking of exception handling design rules. The syntax of ECL is simpler than the one needed to specify properties in JavaMOP, and there is a single action available in our approach (send the violation information to DAEH server).

VIII. Conclusions and Future Work

This paper introduces a domain-specific language to specify the exception handling policy of a system, which is, more often than not, undocumented and implicitly defined – negatively impacting the system robustness [12]. This work also presents a runtime monitoring tool to support the dynamic checking of such an exception handling policy. Two case studies were conducted to evaluate the proposed approach. Our findings indicate that the approach can be used to specify and dynamically check the exception handling design policy of a system. We are currently working on evaluating the needs for adding new language constructs to ECL.

References

[1] R. Di Bernardo, R. Sales, F. Castor, R. Coelho, N. Cacho, S. Soares Agile Testing of Exceptional Behavior. In Proc. of 25th Brazilian Symposium on Software Engineering, 2011.

[2] H. Shah, et al., Why do developers neglect exception handling In Proc. of the 4th International Workshop on Exception handling, 2008.

[3] R. A. Maxion and R. T. Olszewski, Eliminating exception handling errors with dependability cases: a comparative, empirical study, Software Engineering, IEEE Transactions on, vol. 26, 2000.

[4] H. Shah, Gerg, C. and M. J. Harrold,, Why do developers neglect exception handling?. In Proc. of the 4th International Workshop on Exception handling, 2008.

[5] F. Cristian. Exception handling and software fault tolerance.IEEE Trans. Comput. 31(6):531540, 1982.

[6] J. Kienzle. On exceptions and the software development life cycle. In Proc. of the 4th International Workshop on Exception Handling, 2008.

[7] R. Sales, R. Coelho, Preserving the ExceptionHandling Design Rules in Software Product Line Context: A Practical Approach, In Proc. of the 1sr Workshop on Exception Handling on Contemporary Systems, 2011.

[8] Ramnivas Laddad. 2003. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich, CT, USA.

[9] R. Terra and M. Valente. A dependency constraint language to manage object-oriented software architectures. Softw. Pract. Exper., 39(12):1073–1094, 2009.

[10] J. Brunet, D. Guerrero, J. Figueiredo. Design Tests: An Approach to Programmatically Check your Code Against Design Rules. In Proc. of *ICSE'09,* 2009.

[11] R. Miller and A. Tripathi, Issues with exception handling in object-oriented systems, In Proc. of *ECOOP'97*. 1997.

[12] M. P. Robillard and G. C. Murphy, Designing robust Java programs with exceptions, In Proc. of *FSE 2000*.

[13] A. Garcia, C. Rubira *et al.*, Extracting error handling to aspects: A cookbook, In Proc. of *ICSM 2007*. IEEE.

[14] A. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu, "A compara- tive study of exception handling mechanisms for building dependable object-oriented software," *Journal of systems and software*, v. 59, n. 2, , 2001.

[15] B. Cabral and P. Marques, "Exception handling: A field study in Java and .Net," in *Proceedings of ECOOP 2007*. Springer, pp. 151–175.

[16] R. Coelho, A. von Staa, U. Kulesza, A. Rashid, and C. Lucena, "Un-veiling and taming liabilities of aspects in the presence of exceptions: a static analysis based approach," *Information Sciences*, v.181, n.13, 2011.

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

[18] M. Bruntink, A. V. Deursen, T. Tourwe. Discovering faults in idiom-based exception handling. In Proc. of *ICSE'06*, 2006.

[19] G. J. Myers. The Art of Software Testing. New York: John Wiley & Sons, 2004..

[20] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. von Staa, and C. Lucena, Assessing the impact of aspects on exception flows: An exploratory study, In Proc. of *ECOOP 2008*.

[21] C. Fu, B. Ryder. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In Proc. of *ICSE'07*, 2007.

[22] M. Robillard, G. Murphy. Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems. In ACM Trans. Softw. Eng. Methodol, 2003.

[23] J. Gosling, *The Java language specification*. Addison-Wesley Profes-sional, 2000.

[24] JB. Goodenough. Exception handling: Issues and a proposed notation. Communic. of the ACM 1975.

[25] D. Jin, P. Meredith, C. Lee, G. Rosu. JavaMOP: Efficient parametric runtime monitoring framework. In Proc. of *ICSE'2012*.