Why Do Developers Neglect Exception Handling?

Hina Shah, Carsten Görg, Mary Jean Harrold
College of Computing, Georgia Institute of Technology, Atlanta, Georgia, U.S.A.
{hinashah,goerg,harrold}@cc.gatech.edu

ABSTRACT

In this paper, we explore the problems associated with exception handling from a new dimension: the human. We designed a study that evaluates (1) different perspectives of software developers to understand how they perceive exception handling and what methods they adopt to deal with exception handling constructs, and (2) the usefulness of a visualization tool that we developed in previous work for exception handling. We describe the design of our study, present details about the study's participants, describe the interviews we conducted with the participants, present the results of the study, and discuss what we learned from the study. Based on our analysis, we suggest several future directions, including the proposal of a new role for the software-development process—*exception engineer*, who works closely with software developers throughout all phases of the software-development life cycle and who concentrates on the integration of exception handling into the core functionality of programs.

Categories and Subject Descriptors

D.1.5 [**Software**]: Object-oriented programming—*Exception Handling*; H.1.2 [**Information Systems**]: User/Machine Systems—*Software psychology*; H.5.2 [**Information Systems**]: User Interfaces—*Evaluation/methodology*

Keywords

Exception handling, interactive visualization, human aspects, user studies.

1. INTRODUCTION

The importance of error handling in software development has been known for a long time, and considerable research has been performed to develop tools and techniques to aid developers in incorporating exception handling into their programs. Lemos and Romanovsky [1] propose an approach that separates the handling of requirements-related, design-related, and implementation-related exceptions during the software life cycle. Lippert and Lopes [6] propose that aspect-oriented programming can help in separating the code of exception handling from the main core functionality code. However, in these proposed approaches both types of code (main functionality code and exception handling code) would still be implemented by the same developer. Filho and colleagues [2, 3] suggest lexically separating error-handling code from normal code so that both code types can be independently modified. In addition they propose leveraging aspect-oriented programming to enhance the separation between error-handling code and normal code. Zhang and colleagues [10] propose a different approach that provides programmers with more intuitive exception-handling behavior and control.

All these approaches concentrate on enhancing the separation between main functionality code and exception-handling code. However, "a number of recent field studies have identified that error handling design in industrial applications typically exhibits poor quality independently of the underlying programming language and application domain."[1] Thus, to gain a better understanding of the problems with exception handling, we took a different approach: exploring the human dimension using a set of studies.

First, we conducted a survey with 34 software developers to better understand the needs of developers related to exception-handling constructs in Java programs. Based on the survey results, we developed a tool called ENHANCE [8] (ExceptioN HANdling CEntric visualization).[2] ENHANCE offers three views of the exception-handling constructs in a Java program to aid software developers in understanding exception-handling constructs in their programs. We briefly describe ENHANCE by showing its three views (in Figure 1) on the NANOXML program[3]—a Java program, which contains about 2700 lines of code.

The *Quantitative View* presents high-level information about throw-catch pairs at the level of packages, classes, or methods. For example, Figure 1(a) shows the Quantitative View for the exception dependencies in NANOXML at the package level. There are 11 throws in the nanoxml package that may be caught by catch blocks in the nanoxml.sax package.

The *Flow View* provides details about selected exceptions flows at abstracted level. The view presents type definitions, throw clauses, and catch clauses as abstract icons on separate layers, and the exception flow is represented as links between them. For example, the highlighted path in Figure 1(b) shows that an exception of type "FileNotFoundException" can be thrown from the throw statement at line 2038 in method nanoXML.XMLMethod.addMethod and this exception can be caught at the catch statement at line 2039 in the same method.

[1]Quote from the call for papers for the 4th International Workshop on Exception Handling, `http://www.comp.lancs.ac.uk/computing/WEH.08/cfp.htm`

[2]Details of this tool can be found in Reference [8].

[3]`http://nanoxml.sourceforge.net/orig/`

The *Contextual View* provides low-level information by embedding exception-handling constructs and their flows in an abstract code view of the system. The example in Figure 1(c) shows the propagation path of an exception across two packages: a throw in the method XMLElement.skipBogusTag in package nanoxml is caught by the catch block in method SAXParser.parse in method nanoxml.sax after it is propagated through five other methods.

To evaluate our visualization, we conducted interviews with three graduate students from the software-engineering group at the Georgia Institute of Technology. The goal of this study was to understand 1) how software developers currently deal with exception-handling constructs and 2) whether EnHanCe can provide assistance in helping them better perform their tasks. The results of this preliminary evaluation study were interesting. It was surprising to find that the participants often ignored exception-handling constructs.

To investigate further the findings from the preliminary evaluation, we performed a more extensive study with nine software developers. This extensive study concentrates on understanding the approach software developers adopt to deal with exception-handling-related tasks, such as designing, coding, reviewing, refactoring, testing, and debugging. Additionally, the study investigates in more detail whether our EnHanCe tool can help software developers to better perform tasks related to exception-handling constructs.

Our study results revealed that some developers have shifted their perspective on exception handling from the intended proactive approach (i.e., how to handle possible exceptions) to a reactive approach (i.e., using exception handling as debugging aids). In addition, some developers dislike being forced to implement exception-handling constructs and therefore, neglect to implement them thoughtfully. Both results explain the poor quality of error handling. To address this problem, we propose a new role for the software-development process—*exception engineers*—who could be designated developers who specialize in designing, implementing, and integrating exception handling constructs.

The main contributions of this paper are:

- a presentation of the results of a study we conducted to understand the human dimension in exception handling,
- an analysis of the results of our study and what we learned from it,
- a proposal for a solution to the issues raised by our study.

In the next section (Section 2), we describe the study in detail. Then, we present the results of the study (Section 3). Next, we discuss what we learned from the study (Section 4), Finally, we conclude and discuss future work (Section 5).

2. STUDY DESCRIPTION

In this section, we describe in more detail our motivation for conducting studies about the human aspects that are related to exception handling, how we designed our interview protocol, and the strategy we adopted to select our participants. We also discuss the methodology we followed to conduct the interviews and the analysis.

2.1 Motivation

As we stated in Section 1, in our previous work [8], we performed a preliminary evaluation of our visualization. In that evaluation, we conducted a study with three graduate students from the software-engineering group at the Georgia Institute of Technology. The study results revealed interesting findings about approaches that software developers adopt while dealing with exception-handling constructs (e.g., the *ignore-for-now approach* in which developers ignore exception handling until there is an error or until they are forced to address it). The results of the preliminary study motivated us to conduct further investigations whose main goals are to understand what approaches software developers adopt to deal with exception-handling tasks, to understand why they adopt those approaches, and to evaluate our visualization to determine whether it can help the software developers to better perform their tasks.

2.2 Protocol and Participant Details

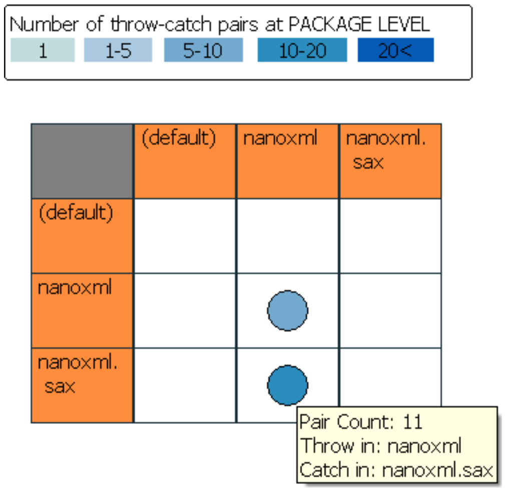
We used the results of our preliminary study [8] to design our new study. In the preliminary study we had created an initial interview protocol based on our past experiences and we had conducted the preliminary study using this interview protocol. The participants in the preliminary study provided feedback for improving our interview protocol. Based on this feedback, we iteratively modified the semi-structured interview protocol for the participants. After we completed the design of the interview protocol, we contacted potential participants by emailing invitations to software developers who had Java experience. We concentrated the study on developers with considerable experience in Java because currently our visualization tool, EnHanCe, supports only Java programs. Nine software developers agreed to participate in the study: eight of them were summer interns at a large multinational organization and one was a full-time employee at the same organization. All the participants had experience with Java, but most even had experience working on programming languages such as C/C++ and during the interviews they provided information about exception-handling approaches they generally adopted irrespective of the languages they used. The intern participants had between one and ten years of prior industrial software-development experience whereas the full-time employee had twenty-five years of industrial software-development experience. And all participants had experience working on large software projects.

2.3 Study Design

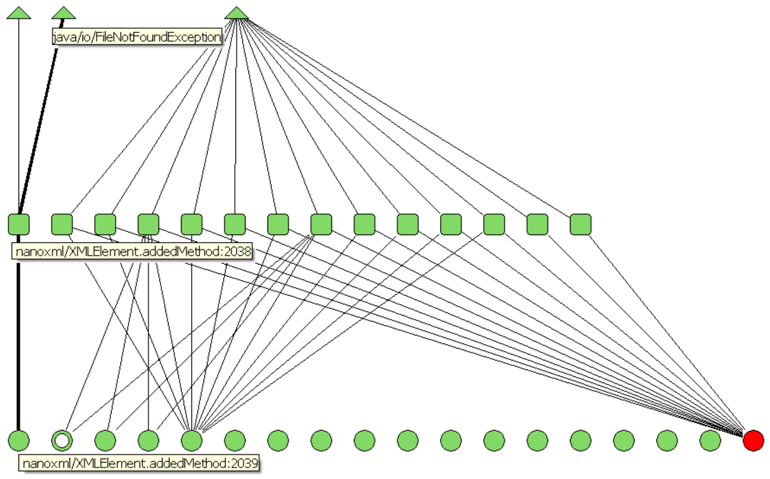
Based on our study goals, stated above, we designed our study to consist of two parts: (1) understanding how the participants approach the problem of understanding exception handling and how they adopt those approaches, and (2) evaluating our visualization using the EnHanCe tool. While conducting the user studies, we first asked questions about the participants' current approaches for understanding exception handling information, then we demonstrated the tool (by running the tool on the NanoXML sample program) and asked them to play around with it, and finally we asked them questions related to the tool evaluation. Each study lasted for approximately one hour.

In Part 1 of the study, we asked the participants to explain how they deal with exceptions. The set of questions covered in the interview protocol includes:

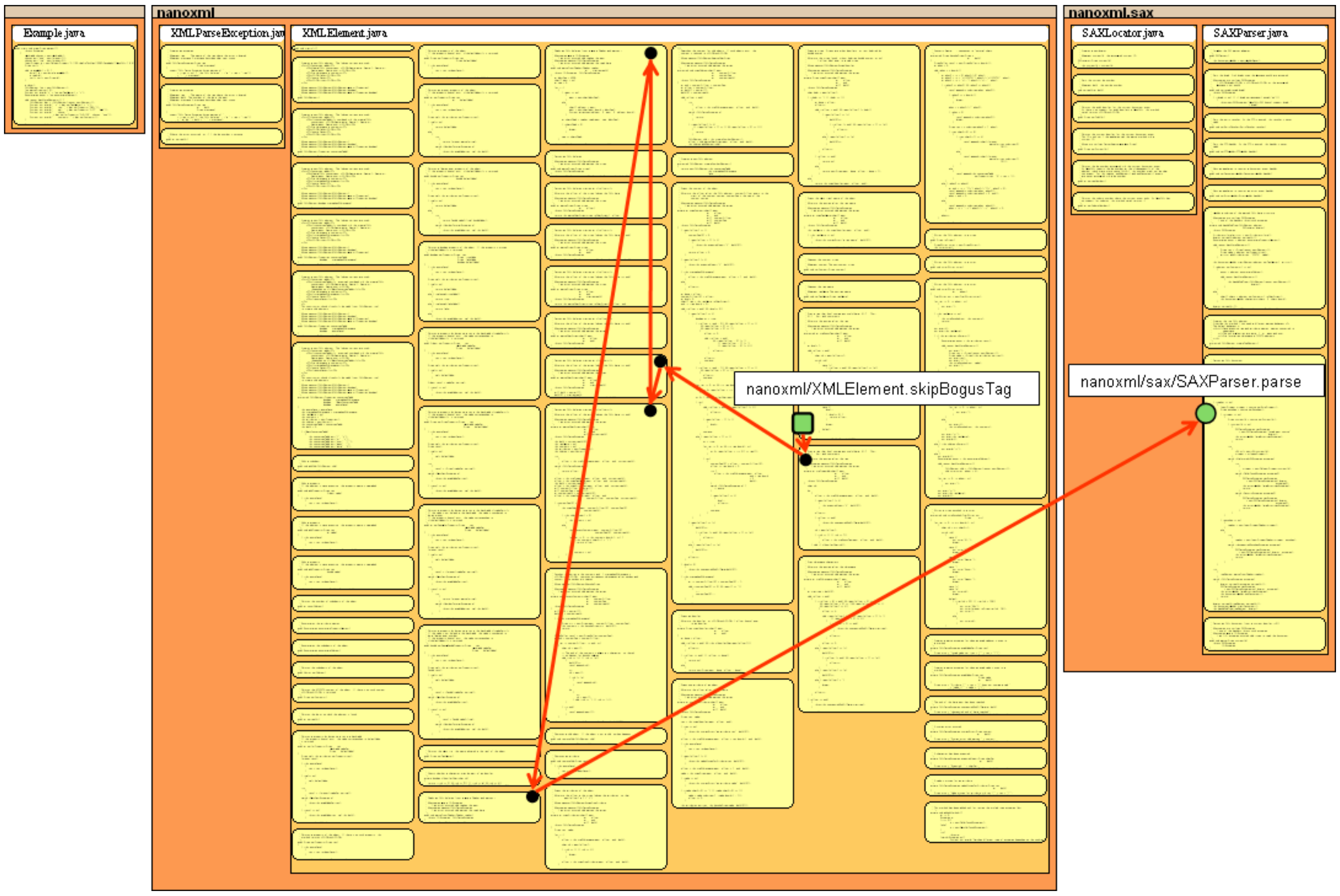
- What approach do you follow to understand exception-flow information in a program?
- For what do you use exception-handling constructs in a program? How do you use them?
- When working with code (e.g., coding, testing, reviewing, and understanding) how often do you pay attention to the functionality associated with exception handling?
- Do you avoid using exception handling in programs? If yes, why?



(a) The Quantitative View showing exception dependencies at the package level.


java/io/FileNotFoundException
nanoxml/XMLElement.addedMethod:2038
nanoxml/XMLElement.addedMethod:2039

(b) The Flow View showing exception flows in NANOXML.



(c) The Contextual View showing the exception flow of an exception embedded in the source code.

Figure 1: The three views of the ENHANCE visualization tool.

In Part 2 of the study, we first demonstrated how to use ENHANCE and introduced each of ENHANCE's views to the participants. We then let the participants use the tool to explore the views using the NANOXML program.[3] After that we asked the participants several questions about the visualization:

- For each view, how can that view be used to support the comprehension of exception-flow-related tasks?
- Would such a visualization better motivate you to deal with the exception-handling parts of the code?

2.4 Method

Before beginning the interview with each participant, we explained the goals of the study, and informed the participants that their information would be anonymous and that they could stop the interview at any time, without being required to give a reason. Additionally, we asked permission to audio record the conversation. Before leaving, we asked their permission to send emails to them in case we required any clarifications at a later time.

We adopted a straight-forward approach for the analysis. We worked through each interview to produce a summary, and then read through the entire field notes multiple times to find cross-cutting sets of common themes.

3. STUDY RESULTS

In this section, we present the results of the interviews with the eight intern participants. Because the full-time employee participant had significantly more experience than the other participants in the study, to maintain the consistency of the participants' demographics, we excluded the full-time employee participant's data from this section. However, we include the full-time employee participant's data in the discussion in Section 4. Based on the analysis of the interview data, several common themes emerged. We present each of them in turn.

Using exception handling for debugging

All the participants we interviewed stated that they use exception handling primarily for debugging purposes. They indicated that when an exception occurs, they use the information provided by the stack trace to understand what caused the exception. All participants, except one, were interested in understanding the path the exception follows from the point it is thrown to the point it is caught. One participant stated

> *I don't need to understand the entire path of [the] stack trace because all I need is the point where the exception is thrown, and that's it.*

The participants mentioned that on some occasions they use the names of the exceptions to understand the context of the surrounding program code (e.g., an exception-code block with a *ParseException* indicates that the code functionality around this exception code block deals with parsing). Additionally, one participant stated that customized exceptions are very useful. The participant mentioned that whenever customized exceptions are used, the participant tries to understand the exception-handling measures (implemented by the application developer) associated with the customized exception to gain a deeper understanding of the core functionality that is implemented. However, most of the participants agreed that in cases where Java's defined exceptions (e.g., *ClassNotFoundException*) are used, they tend to ignore understanding the exception handling implemented around these exceptions.

Adopting the ignore-for-now approach

Another common attitude held by the participants is that they do not think that exception handling is a high-priority task. They believe that dealing with exception handling is tangential to the actual tasks of the program—handling the program's main functionality. Thus, neglecting a thoughtful implementation of exception-handling constructs is allowable. Participants think that it is time consuming and hence, a waste of time, to design exception-handling code in advance. They believe that a thoughtful implementation of exception handling can be postponed until the error actually occurs. They can then identify the cause and fix the error. Statements made by participants include

> *I ignore exceptions because I don't need to take care of them. Whenever something goes wrong just fix the error, it's not worth spending the time.*
>
> *I care only about the right path.*
>
> *I try to keep things [program] simple as far as possible by implementing some simple exception handling stuff.*
>
> *At the beginning I tend to avoid it [exceptions], but when they occur I handle them.*

The only exception to the *ignore-for-now* behavior occurs in scenarios where the code on which the participants were working already had some useful implementation of exception handling. In such scenarios, the participants agreed that they try to mimic the existing code. Thus, in general, participants try to avoid handling exceptions unless some support structure is already available.

Perceiving forced exception handling in Java

A third common attitude held by the participants was that they use the exception handling because they perceive that the language (i.e., Java) forces them to use it. They explained this by stating that they will not use exception handling if the compiler does not prompt them with compile-time errors when the appropriate exception-related code was missing (e.g., declaration of throws clause, or implementation of respective try-catch block). Some participants mentioned that this behavior was partly because they had prior experience with programming languages such as C and C++, where no compulsion exists to implement exception-handling code. Statements made by participants include

> *I don't love to use it [exception handling] but for some features of the language, I have to use it.*
>
> *I need to catch exceptions because Java requires me to do it. I just fulfill the language's requirements.*

Using the EnHanCe visualization tool

Seven out of the eight intern participants thought it was difficult to understand EnHanCe's *Quantitative View* and its usage. The participants complained about the layout and the usage of information that was being displayed. They thought that the dependency-relation information that the view provides might be more useful for project managers than for them. Only one participant disagreed, and that participant suggested another use of the view: understanding the exception concentration in the program and identifying exceptions that are not caught.

However, all eight participants thought that the other two views were quite useful. When viewing the unreachable blocks that the *Flow View* displays, participants mentioned that they were unaware that such unreachable blocks actually exist in programs. Many made interesting conclusions about the program based on the exception patterns revealed in the view. Also, participants thought that the *Contextual View* is useful because it reveals the complete paths of exception flows in the program, giving information about the number of levels an exception is thrown before it is handled (e.g., distance between throw and catch [9]).

Another common attitude that the study revealed is that, despite the availability of a visualization for aiding exception-handling related tasks, participants were not motivated to change their approach to dealing with exception handling. However, they also noted that at times when they are forced to deal with exception handling, such a tool would make their work simpler. Participants comments include

> *Such a tool will not motivate me to deal with exception-handling better, it will only help me spend less time.*
>
> *I might have used it, but currently, my focus will still be more on functionality.*

4. DISCUSSION

In this section we discuss the results of our analysis and we propose a solution to solve some of the problems related to exception handling that were exposed in the analysis. We also discuss our visualization in the context of the analysis results and solution we propose.

Usage shift to Debugging

The analysis results show that there has been a shift from the way exception handling was originally designed to be used (error handling and recovery) to the way it is currently used (debugging and understanding). For instance programming languages such as Java provide exception handling constructs that were originally designed to be used for systematically handling error conditions in a program by taking the necessary action based on the situation (i.e., provide scaffolding either to recover from an error or to allow a graceful exit when an unrecoverable error has occurred) [5]. However, the results clearly indicate that developers tend to use exception-handling constructs mostly for debugging purposes (e.g., to understand the flow of the program when an error occurs).

We observed that this shift in the usage pattern of exception handling has direct implications on the way it is implemented by developers. Developers view exception handling more as a feature that aids them in debugging and they tend not to invest time in implementing code for proper handling of error conditions unless its implementation helps with debugging. This behavior explains to some extent why most developers implement only basic exception-handling code, such as printing stack traces in catch blocks even if the situation demands to handle the exception in a more sophisticated way (e.g., release resources in database scenarios, perform clean ups, or print to log files).

The shift in use of exception handling from error recovery to debugging and the developer's ignore-for-now attitude that we discussed in the previous section indicate that the attitude of developers is more reactive whereas exception handling was designed to support developers to be proactive. This reactive nature, to some extent emerges because developers tend to consider exception-handling functionality as lower priority than the main functionality.

However, the significantly more experienced participant had a different and interesting perspective on this topic. In addition to use exception handling for debugging purposes, this participant considered the exception handling also as a good way of documenting potential problems. Thus, these findings could indicate that there is a difference in the approach adopted by significantly more experienced developers as compared to not as experienced developers. To investigate these differences in more detail, we intend to conduct similar studies with more experienced developers as part of our future work.

Forced Exception Handling

In programming languages such as C, exception-handling mechanism support is not as extensive and easy as it is in Java [4, 7] because these languages do not provide an elegant framework for exception handling. Because of the lack of such a framework software developers tend to ignore implementing exception handling.

To address this problem, Java's exception-handling mechanisms were designed such that they support the software developers in implementing the exceptions-handling functionality in an easy and elegant manner.[4] However, our study results show that Java's design was perceived as a force by most of the participants what indicates that the software developers were not willing to handle exceptions but the language imposed compulsions to handle exceptions.

Therefore, the two similar attitudes of "avoiding exception handling" that developers carry on both the occasions — when no support for exception handling is available and when support is available — indicate that developers are less willing to deal with exception handling.

Nevertheless, the significantly more experienced participant in our study had a completely different perspective on Java's forced exception handling. This participant found Java's exception handling beneficial and easy to use. An interesting quote from the conversation is:

> *It's so cheap to put a little check and it's not time consuming too, it doesn't slow me down infinitely. And the throw-catch makes life so simple.*
>
> *C had signals to indicate errors and they were really tough to use, but Java makes that life simpler by providing some scaffolding for exception handling.*

This opinion highlights another difference of attitude between the significantly more experienced software developer and the less experienced developers. We intend to conduct further investigations through detailed studies to confirm such findings.

Need for Exception Engineers

In the past, there has been considerable research performed in separating error-handling code from main-functionality code. However, despite the code separation, the responsibility of implementing both, the main as well as the exceptional functionalities is the task of the same developer. Under such situations, the developers tend to give low priority to exception handling because they believe that it does not form a part of their main functionality. Thus, because developers are burdened with the responsibilities of managing both the main functionality and the exceptional functionality, they might continue ignoring the exception-handling functionality.

To address this problem we propose to separate the main functionality from the exception functionality in a different way. Whereas separating error handling code from the core functionality code is the current approach, we propose that the separation should be taken one step further where error-handling developers are separate from main functionality developers. We recommend that a new role should be introduced in the software development process—the role of *exception engineers*. With a similar functionality as a *test engineer* (the primary focus of a test engineer is to test the program for bugs), the *exception engineer*'s role would be to focus on exceptional code while working closely with the developers of the core functionality code. Thus, now the software development life cycle would involve three types of engineers working in harmony: (1) the developers who focus on developing the main functionality, (2) the test engineers who focus on failing the functionality, and (3) the exception engineers who focus on recovering from failures or gracefully exiting the program on unrecoverable failures.

[4] `http://web.archive.org/web/20050417232501/http://technetcast.ddj.com/tnc_program.html?program_id=63&page_id=3`

Exception engineers may work closely with the developers through the different phases in the software development life cycle with a primary focus on determining the exceptional conditions that may occur in the system (e.g., creating use cases with exceptional and failure scenarios [1] during the design phase, writing exception handling code during the implementation phase, and testing the exceptional path during the testing phase). Special training sessions may be organized for exception engineers in which they are educated about the common exceptional situations that have been encountered in the past. Also, they may be exposed to some of the common practices highly-experienced software developers adopt to deal with exceptional situations.

This solution would address the ignore-for-now approach as well as debugging problems. It would resolve the ignore-for-now problem because now the core functionality developers do not have to focus any more on writing exception-handling code. By defining a dedicated role for the exception engineers, they will concentrate on the implementation of exceptional functionality, which was previously handled by the main developers. Also, the core functionality developers could now work closely with the exception engineers to incorporate better debugging code along with error handling. Exception engineers could use programming paradigms that support the separation of main functionality from the cross-cutting functionality of exception handling (e.g., aspect-oriented programming) to develop the exception-related code in parallel along with the main software developers.

Introducing a new role of exception engineers would demand a close coordination between them and the other developers; it would be interesting to investigate further the coordination between the exception engineers and the other developers. This will not only help to unfold unseen problems but also help to design the role of exception engineers in more detail.

Visualization for Exception Handling

Our studies about the visualization revealed that developers tend to use visualization tools like ENHANCE less frequently because their main focus is rarely on implementing code related to exception handling. However, they stated that if their role requires them to work extensively with exception handling (e.g., working on critical projects), then they would definitely use such tools to save time and efforts.

An interesting quotation from one of the participants was

> *If I was a power user of exceptions then I would definitely use this tool a lot.*

Additionally, the studies also showed that the participants thought that two of the views, the *Flow View* and the *Contextual View*, provided useful information about exception handling. However, only one of the participants thought that the *Quantitative View* provided useful information. As a part of the future work, we intend to re-design the *Quantitative View* in such a way that it clearly indicates what information it provides.

Thus, the study results of the visualization indicate that such a visualization would be helpful for developers whose work is focused on exception handling. If the role of exception engineers would be established successfully, then they could benefit from the ENHANCE visualization tool.

5. CONCLUSIONS AND FUTURE WORK

Software developers need to understand the complex mechanisms of exception handling in large software systems to be able to develop, debug, and maintain those systems. In this paper, we presented the results of studies we conducted to gain a better understanding of the human dimension in tasks related to exception handling.

The study results showed that developers are not satisfied with the existing exception-handling mechanisms in Java. They do not like the force that languages such as Java impose to implement exception handling constructs. Furthermore, sometimes the complex mechanisms are even overwhelming and developers wish to have a better support to understand and implement exception-handling constructs.

To address this problem, we proposed to introduce a new role of *exception engineers* who would be dedicated to the design, implementation, and maintenance of the parts of programs dealing with exceptions. Whereas such a role could improve the software-development process related to exception handling on the one hand, it also would introduce new challenges on the other hand. Exception engineers would need a proper training, the communication between exception engineers and developers of the core functionality would have to be streamlined, and features such as aspect-oriented programming would have to be considered to support the divided work flow of implementing exception-related and core functionality program code. We plan to investigate these challenges as part of our future work.

Participants in our study considered two views of our visualization tool ENHANCE—the *Flow View* and the *Contextual View*—to be very helpful in understanding, implementing, and debugging exception handling constructs. Therefore, we believe that ENHANCE would be a valuable tool for exception engineers. The *Quantitative View* was not considered to be as helpful, and we plan to re-design this view.

Finally, we intend to conduct additional studies with more experienced developers (more than 20 years of experience) from different organizations to understand whether experience and organizational cultures play important roles in the way exception handling is approached and used.

Acknowledgements

This work was supported in part by NSF awards CCF-0429117, CCF-0541049, and CCF-0725202 to Georgia Tech. The participants in our study provided valuable information. The anonymous reviewers gave helpful comments on an earlier version of this paper.

6. REFERENCES

[1] R. de Lemos and A. B. Romanovsky. Exception Handling in the Software Lifecycle. *International Journal of Computer Systems Science and Engineering*, 16(2):119–133, 2001.

[2] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–162, New York, NY, USA, 2006. ACM.

[3] F. C. Filho, A. Garcia, and C. M. F. Rubira. Error handling as an aspect. In *BPAOSD '07: Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development*, pages 1–6, New York, NY, USA, 2007. ACM.

[4] N. H. Gehani. Exceptional C or C with exceptions. *Softw. Pract. Exper.*, 22(10):827–848, 1992.

[5] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[6] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM.

[7] B. G. Ryder and M. L. Soffa. Influences on the design of exception handling: Acm sigsoft project on the impact of software engineering research on programming language design. *SIGPLAN Not.*, 38:16–22, June 2003.

[8] H. Shah, C. Görg, and M. J. Harrold. Visualization of exception handling constructs to support program understanding. In *Proceedings of the ACM Symposium on Software Visualization*, pages 19–28, 2008.

[9] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, pages 348–357, 1998.

[10] L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in java. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 175–184, New York, NY, USA, 2007. ACM.