International Journal of Applied Engineering Research ISSN 0973-4562 Volume 13, Number 5 (2018) pp. 2130-2140

An Investigation of Exception Handling Practices in .NET and Java Environments

Preetesh Purohit[1] and Vrinda Tokekar[2]

[1]*Ph.D. Scholar, Institute of Engineering and Technology, Devi Ahilya Vishwavidyalaya, Indore, India.*

[2]*Professor and Ph.D. Supervisor, Institute of Engineering and Technology, Devi Ahilya Vishwavidyalaya, Indore, India.*

Abstract

Java, C# and most other modern programming languages depend on exceptions for dealing with unhappy path or abnormal events. Error and exception handling approaches is not yet proper and perfect, though it was a significant improvement over any other mechanisms. It can be claimed that this mechanism is sternly finite, if not, imprecise. This research paper targets to devote to the dialogue by catering computable measures on how software developers are recently dealing with exceptions. We analyzed various software projects in .NET and Java environments. The major finding is that exceptions are not being correctly used as an error recovery mechanism. Handlers for dealing with exceptions are not specially designed for allowing recovery from crisis due to exception and, commonly, one of the following operations are performed by software developers: notification to end user, error logging and termination of application. This research done on exception handling provides a measure helpful for influencing the evolution of upgraded error handling techniques.

Keywords: Software Development Environment, Programming Languages, Exception Handling Approaches

INTRODUCTION

A language for software development must provide the developers with functionalities and features that make it convenient to deal with abnormal and unhappy circumstances, and also to recover from errors, to make the final software product robust. Robust application must be able to deal with the authentication procedures that fail, temporary disconnection of network links, hard disks that are full etc.

Most modern high level programming languages depend on exceptions for dealing with unhappy path or abnormal events. While exception handling was an important development over other approaches like checking return codes, it is not yet perfect. It can be claimed that this mechanism is seriously limited, if not, imprecise. Issues include:

- Software developers throw exceptions which are generic in nature, which make it almost ungovernable to appropriately manage errors and return from abnormal situations without shutting down the application.
- In some programming platform, developers do not catch enough exceptions making applications crash even on minor error situations (particularly relevant in C#/.NET). On the other side, developers catch generic exceptions, not providing proper error handlings, making the programs continue to execute with an untrustworthy state (particularly relevant in Java).
- Developers that try to provide suitable exception handling see their productivity and effectiveness on developing business logic seriously impaired. A task as simple as providing exception handling for reading a file from disk may imply catching and dealing with plenty of exceptions (e.g. SecurityException, FileNotFoundException, DiskFullException, IOException, etc.). As productivity and effectiveness of programmer decreases, cost of project escalates, programmer's motivation diminishes and, as a consequence, software quality suffers.
- Providing suitable exception handling can be quite difficult, error prone, not to say, time consuming. As per the condition, it may be necessary to enclose *try-catch* blocks within loops in order to retry operations; in some scenario it may be necessary to abort/kill the program/process or perform different recovery procedures. Unusual situations, like having to deal with being thrown an exception while trying to close a file on a *catch* of a *finally* block, are not uncommon.

After the launch of Microsoft's .NET framework, the study of error handling approaches in programming languages has been fuelled, and become interesting among scholars.

Presently, the .NET and the Java environments develop the quantum of the modern software development environments for business applications. Remarkably, Microsoft opted to have a different exception handling approach than in Java. While in Java, in most cases, the developer is forced to declare which exceptions can occur in its code and explicitly deal with exceptions that can occur when a method is called. The rational for this is that if the programmer is forced to immediately deal with errors that can occur, or re- throw the exception, the software will be more robust. i.e. the programmer must be constantly thinking about what to do if an error occurs and acknowledge the possibility of errors. On the other camp, in .NET the developer is not forced to declare which exceptions can occur or even deal with them. Whenever an exception occurs, if unhandled, it propagates

across the stack until it terminates the application.

On the .NET's side, the disagreements for not having checked exceptions that are commonly used are [1]:

- Checked exceptions interfere with the programmer's productivity since they cannot concentrate in business logic and are constantly forced to think about errors.
- Since the developer is mostly concentrated in writing business logic and not dealing with errors, it tends to shut-up exceptions, which actually makes things worse. (Corrupt state is much more difficult to debug and correct than a clean exception that terminates an application.)
- Errors should be cleaned by exhaustive testing. A sufficiently accurate test suite should be able to expose dormant exceptions, and corresponding abnormal situations. For the problems that remain latent, it is better that they appear as a clean exception that terminates the application than having them being swallowed in a generic *catch* statement which leads to corrupt state.

Certainly, both sides cannot be completely right. But, overall, the significant message and learning is that in order to develop high-quality robust application, in an effective and productive way, fresh proposal in error handling is requirements of today. The available techniques of exception handling are neither perfect nor sufficient.

This study aims to devote to the conversation by presenting significant quantitative measures on how software developers are using exception handling. We investigated different applications, both for Java and .NET, covering different software varieties (*server-apps, server programs, desktop programs,* software *libraries)*. Comprehensively, this amount to more than three million lines of source code (3410294 LOC) of which above four percent (137720 LOC) are dedicated to exception handling. For this study, we have investigated and processed 18589 *try* blocks and corresponding handlers.

The data presented in this research article is significant to counsel the development of new approaches and mechanisms to exception handling. Other outcomes will support e.g. justify the feasibility of using available methodologies, to implement exception handlers as advices by applying Aspect Oriented Programming (AOP).

The work in this paper is arranged section wise: Section II discusses significant related work; Section III elaborates the application setup used in this research; Section IV explains the methods and approaches used in the analysis; Section V presents the outcomes of the tests and observations about their significance; finally, Section VI wraps up the paper.

NOTEWORTHY CONTRIBUTIONS

After the leading work of John B. Goodenough in the definition of a notation for exception handling [2] and Flaviu Cristian in defining its usage [3], the programming language constructs for handling and recovering from exceptions have not improved much.

Lot of studies have been done over the years for validating the options taken in each different implementation. For example, Alessandro Garcia, *et al.* did a comparative study on exception handling (EH) mechanisms available developing dependable software [4]. Alessandro's work consisted in a survey of exception handling approaches in dozen of object-oriented languages. Each programming language was analyzed in respect to ten technical aspects. The major conclusion of the study was that "none of the existing exception mechanisms has so far followed appropriate design criteria" and programming language designers are not paying enough attention to properly supporting error handling in programming languages.

Saurabh Sinha and Mary Jean Harrold performed an extensive analysis of programs with exception handling constructs and discussed their effects on analysis techniques such as control flow, data flow, and control dependence [5].

R. Miller and A. Tripathi identified several problems in exception handling mechanisms for Object-Oriented software development [6]. In their work, it is shown that the requirements of exception handling often conflict with some of the goals of object-oriented designs, such as supporting design evolution, functional specialization, and abstraction for implementation transparency.

Martin P. Robillard and Gail C. Murphy in their article on how to design "robust Java programs with exceptions", classified exceptions as a global design problem and discussed the complexity of exception structures [7]. In their work, the authors pointed that the lack of information about how to design and implement with exceptions lead to complex exception handling code.

Due to AOP approach to EH, two interesting studies were published emphasizing the separation of concerns in error handling code writing [8][9]. Martin Lippert and Cristina Lopes rewrote a Java application using AspectJ. Their objective was to provide a clear separation between the development of business code and exception handling code. This was achieved by applying error handling code as an *advice* (in AOP terminology) [10]. With this approach they also obtained a large reduction in the amount of exception handling code present in the application. Lippert's paper also accounts the total number of *catch* blocks in the code and the most common exception classes used as parameters for these *catch* statements. One of the measures they present to support their AOP approach is the reduction of the number of different handlers effectively written for each one of the most commonly used exception classes. For the top 5 classes were implemented between 90.0% and 96.5% less handlers.

The aim and objective of this study is motivated by the research work done earlier in different studies [11-20] and here we attempt to verify and validate the results on exception handling approaches by software developers on projects in various categories. The significance of this study is to understand that how much of the software code in

applications is dedicated to handling of abnormal events or unhappy path, more precisely to exception handling and to get recover from exceptional conditions, together with how programmers handle exceptions. We worked to find out how programmers make use of exception handling facilities available in programming languages and to identify practicable shortcomings in their code rather than analyzing the qualitative techniques available in programming languages.

APPLICATION SETUP

Together with software development environment like .NET and Java platforms, C# and Java programming languages are also the targeted environments for this study.

Choosing a group of projects/applications for the research was quite crucial. General programming practices on the specified platforms must be available, in the code written in the selected applications. To obtain fair results the applications must represent "real world" software projects; developed for production and regular use. Also it is necessary to avoid any bias due to immature (prototypes/beta versions) software applications; as they care less about exception handling. Lastly, the source code and binaries of the applications must be available to perform different types of analysis.

Globally, we examined thirty two applications divided into two sub-sets of sixteen .NET programs and sixteen Java programs. Each one of these sub-sets was organized in four categories accordingly to their nature:

- Desktop programs:-**Stand-alone applications**
- Server programs: - **Servers**
- Servlets, JSPs, ASPs and related classes: - **Applications running on servers (Server-Apps)**
- Software libraries providing a specific application-domain, Application Programming Interface (API): - **Libraries**

The complete list of applications for Java and .NET is shown in Table 1 and 2 respectively.

Table 1. Applications analyzed in Java group.

Java	Stand-alone	Eclipse	Extensible development platform and IDE
		Columba	Email Client
		J-Ftp	Graphical Java network and file transfer client
		Compiere	ERP software application with integrated CRM solutions
	Servers	Berkeley DB	High performance, transactional storage engine
		JCGrid	Tools for grid-computing
		Apache Tomcat	Servlet container
		Jboss	J2EE application server
	Server-Apps	Xplanner	Project planning and tracking tool for Extreme Programming
		MobilePlatform	Banks and mobile operators software for SMS and MMS services in cellular networks
		GoogleTag Library	Google JSP Tag Library
		Exoplatform	Corporate portal and Enterprise Content Management
	Libraries	Kasai	Authentication and authorization framework
		JoSQL	SQL for Java Objects querying
		Javolution	Real-time programming library
		Thought River Commons	General Purpose Library

APPROACH AND METHODOLOGY USED

Using different processes the selected test applications were examined and analyzed carefully at source code level (Java and C#) and at binary level (IL code/bytecode and metadata).

Using antlr [11] for C#, and javacc [12] for Java, two parsers were developed to perform the source code analysis. Generated parsers were then customized to take out all the exception handling code into one text file per application. Minute examination of these files was performed, to create records about the content of exception handlers.

Because of grand size of Mono, only its "corlib" module was processed, for all other selected applications the source code was examined thoroughly.

The customized parsers were also used to detect *try* statements inside *while* and *do..while* loops i.e. to identify and collect information about *try* blocks inside loops. As this type of operations in programming is usually done to retry a block of code that has raised an exception in order to recover from an unhappy path or abnormal situation.

As the investigation of the various applications source code is not sufficient by itself when trying to differentiate between the exceptions that the programmer wants to handle and the exceptions that might occur at runtime. It is so because the generated bytecode/IL code can produce more and different exceptions than the ones that are declared in the applications source code by means of throw and throws statements. The main purpose of this article is to understand how developers use the exception handling approaches available in languages available for development of software applications.

Table 2. Applications analyzed in .NET group

.NET	Stand-alone	SQLBuddy	SQL scripting tool for use with Microsoft SQL Server and MSDE
		AscGen	Application to convert images into high quality ASCII text
		SharpDevelop	IDE for C# and VB.NET projects.
		Nunit	Unit-testing framework for all .NET languages
	Servers	DCSharpHub	Direct connect file sharing hub
		Nhost	Server for .Net objects
		Perspective	Wiki engine
		NeatUpload	Allows ASP.NET developers to stream files to disk and monitor progress
	Server-Apps	SushiWiki	WikiWikiWeb like Web Application
		SharpWebMail	ASP.NET webmail application that is written in C#
		PhotoRoom	ASP.NET web site for managing on-line photo albums.
		UserStory.Net	Tool User Story tracking in Extreme Programming projects
	Libraries	NLog	Logging library
		Mono (corlib)	Open-source CLR implementation
		Report.NET	PDF generation library
		SmartIRC4NET	IRC library

Two different applications were developed: one for Java and another for .NET to perform the analysis of the Java class files and of the .NET assemblies. The first application targeted the Java platform and used the Javassist bytecode engineering library [13] to read class files and extract exception handler information. The second one used the RAIL assembly instrumentation library [14] to access assembly metadata and IL code and extract all the information about possible method exceptions, exception handlers and exception protection blocks.

For easy statistical representation, all data was stored systematically in table form.

Only one package (and sub-packages) of classes (Java) or only one file (.NET) of each and every application was examined and analyzed. The names of the packages and files that were used in this study are mentioned in Table 3. These targets are chosen on the basis of their relevance in the implementation of the application core and on the size of the files.

RESULTS

We present the outcomes of this analysis, drawing some significant observations in the following subsections.

Table 3. List of Java Packages and .NET Assemblies examined.

Java	.NET
org.eclipse	SqlBuddy.exe
org.columba	Ascgen dotNET.exe
net.sf.jftp	SharpDevelop.exe
org.compiere	nunit.core.dll
Berkeley DB (all)	DCSharpHub.exe
JCGrid (all)	nhost.exe
org.apache	Perspective.dll
JBoss (all)	Brettle.Web.NeatUpload.dll
MobilePlatform (all)	SushiWiki.dll
XPlanner (all)	SharpWebMail.dll
GoogleTagLibrary (all)	PhotoRoom.dll
Exoplatform (all)	rq.dll (UserStory)
org.manentia.kasai	NLog.dll
JoSQL (all)	mscorlib.dll
Javolution (all)	Reports.dll
ThoughRiverCommons (all)	Meebey.SmartIrc4net.dll

In spite of everything, we should aware that even though we were used thirty two numbers of applications, it is not practicable to conclude the experience of our study to the whole Java/.NET world. To generalize the result to whole world of .NET/Java, it would be mandatory to have more and very meaningful number of applications across various categories of software projects. But we think that the outcomes of our work, acknowledge an appropriate flash into prevailing general coding habits in software exception handling approaches.

A. Code in Applications for Error Handling

To find an important metric; the percentage of source code that is used in error handling practices, and to know the prevailing practices used in programming for error handling, we checked the number of lines of code inside all handlers(catch and finally) to the total number of lines of the program. Chart 1 is used to show the outcomes.

It is quite visible that in .NET there is less code dedicated to error handling than in Java. It can be explained by the fact that in .NET it is not compulsory to handle or declare all exceptions while in Java it is compulsory, thus increasing the total amount of code for error handling in Java. Strangely, there is an exception to this pattern the difference is almost non-existent in the Server application group. We carefully investigated the application's code to better understand the outcomes. We found that programmers in .NET and Java wrote almost identical code for Server class of applications, that means they anticipate similar type of problems like (e.g. missing relevant data, connection and communication problems in database etc.) and coder in both the platforms use the similar remedy and the most general handler action in Server Application group is error logging or user notification.

The outcomes of our study show that in Servers group maximum error handling code was around 7%. Our results found that the work devoted in coding of error protection approaches is not as high as anticipated, even for supremely significant software like servers. Comprehensively, the outcome is around 3% for .NET and around 5% for Java. One unexpected outcome is that the total amount of code devoted to exception handling is much less. It is even more shocking in Java where handling exceptions is almost compulsory even in small software applications. The severe issue is that generally error handling approaches used by programmers are used mostly for user notification/error logging, to abort/terminate the program or to force to continue their application execution in untrustworthy state than to actually recover from abnormal/erroneous situations. The applications used in various categories in this investigation are widely used and matured enough. The compulsion of handling checked exceptions in Java adequately increases the amount of error handling code by almost double the code written in .NET, although it is a meager part of the total code of an application.

B. Analyzing Code in Exception Handlers

To know how software developers use exception handling approaches it is useful to find what type of activities are

performed when an error occurs apart from finding the total amount of code that deals with exceptions.

To analyze and to report on this case we had to carefully inspect sets of ten thousand lines of application source code. We examined all the *catch* and *finally* handlers in all the applications except for Eclipse and JBoss. Out of the 96405 lines of code existing inside of exception handlers only 10% were examined, for those two, due to their size. Even so, they are representative of the rest of the application.

We propose a small set of categories that enable the grouping of related actions to simplify the classification of these error handling actions. Table 4 summarized these categories.

It may possible that a handler may log an error, close an established connection and exit the application i.e. an exception handler may contain actions that belong to more than one category. This type of case is common. In the results, handler like above would be classified in three categories. Three distinct categories: Log, Close and Return are used for representation of these actions.

Table 4. Handler's actions varieties with brief explanation.

Brief Explanation	**Classified as**
It has no code and does nothing more than cleaning the stack, the handler is empty.	Empty
Any type of error logging or user notification is carried out	Log
In the event of an error or in the execution of a finally block some kind of pre-determined (alternative) object state configuration is used	Alternative/Static Configuration
A new object is created and thrown or the existing exception is re-thrown	Throw
The protected block is inside a loop and the handler forces it to abandon the current iteration and start a new one	Continue
The handler forces the method in execution to return or the application to exit. If the handler is inside a loop, a break action is also assumed to belong to this category	Return
The handler performs a rollback of the modifications performed inside the protected block or resets the state of all/some objects (e.g. recreating a database connection)	Rollback
The code ensures that an open connection or data stream is closed. Another action that belongs to this category is the release of a lock over some resource	Close
The handler performs some kind of assert operation. This category is separated because it happens quite a lot. Note that in many cases, when the assertion is not successful, this results in a new exception being thrown possibly terminating the application	Assert
A new delegate is added	Delegates (for .NET only)
Any kind of action that does not correspond to the previous ones	Others

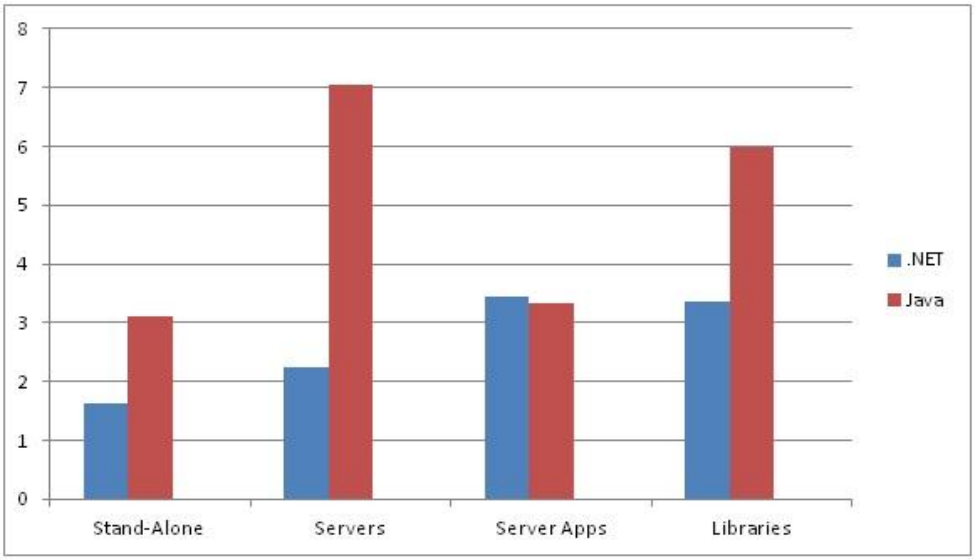

Chart 1. Quantitative analysis of exception handling code in applications

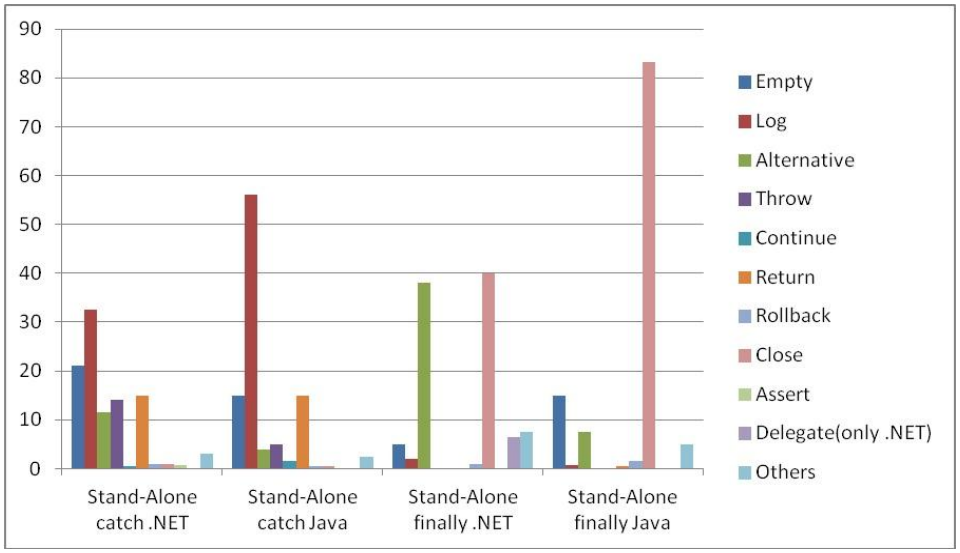

Chart 2. All handlers count for various actions in .NET and Java platforms for Stand-Alone applications

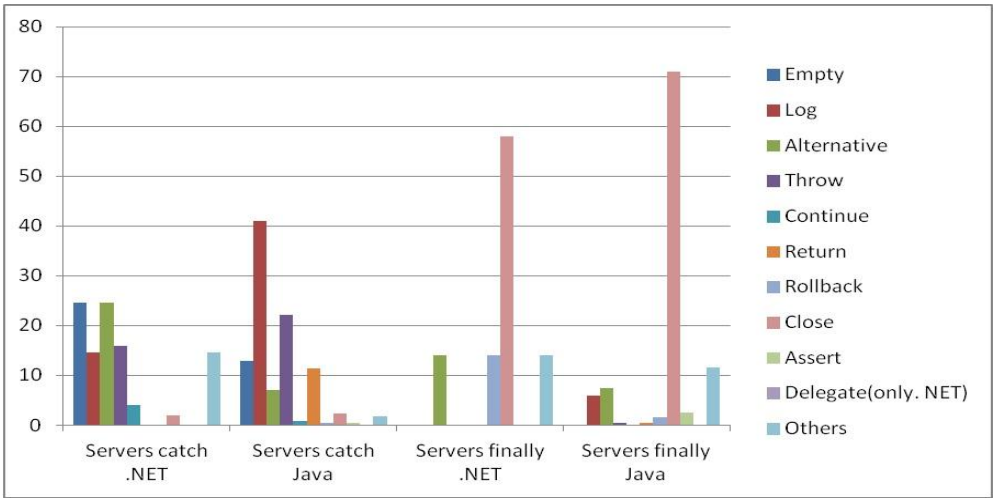

Chart 3. All handlers count for various actions in .NET and Java platforms for Servers applications

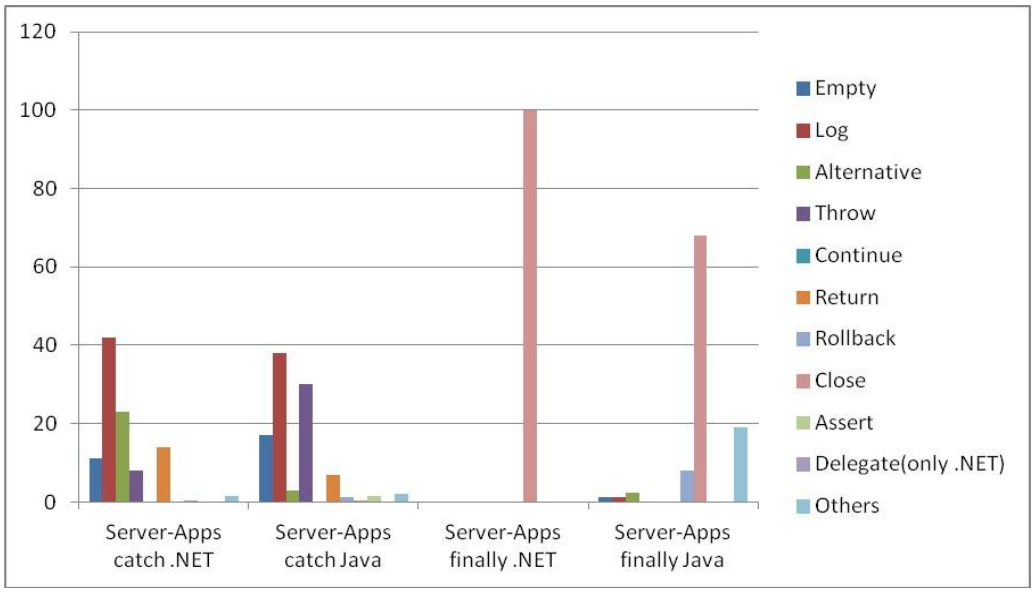

Chart 4. All handlers count for various actions in .NET and Java platforms for Server-Apps applications

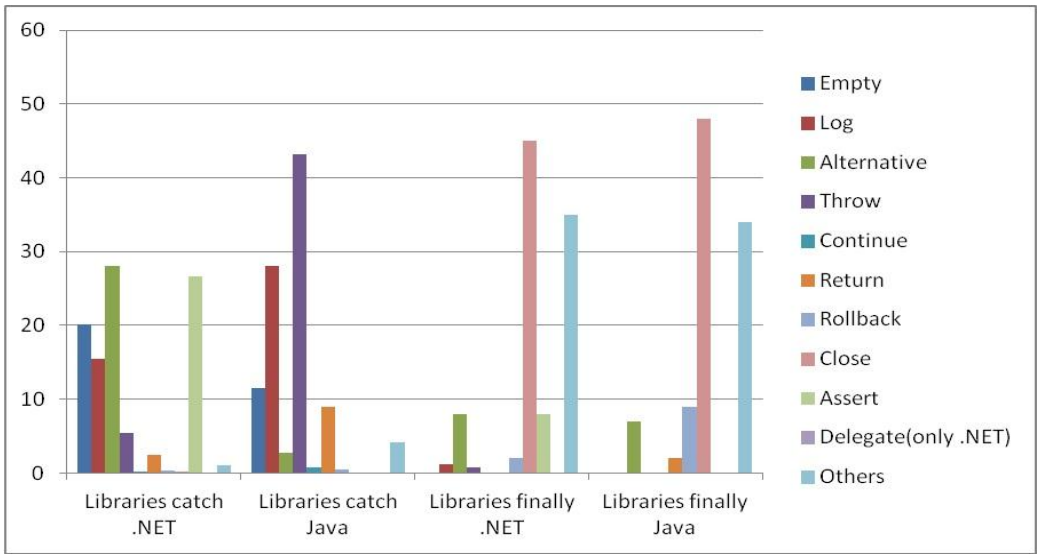

Chart 5. All handlers count for various actions in .NET and Java platforms for Libraries applications

We opted for doing counts in the same chart for *catch* and *finally* handlers on one type of application at a time in .NET and Java platforms. To ensure that small size software do not bias the results towards their specific error handling strategy, the distribution of handler actions for each software application was calculated as a weighted average as per the number of actions found in each application.

In next four charts, i.e. in chart 2 to 5, the outcomes obtained are shown. The average of results for single application set type for .NET and Java *catch and finally* handlers are shown in charts 2 to 5.

From the charts it can be observed that for .NET *catch* handlers, 60% to 75% of the total distribution of handler actions is composed of three categories: *Empty, Log* and *Alternative Configuration*, in the all four application groups.

Since there are no checked exceptions in the CLR and, therefore, developers are not obliged to handle any type of exception. But while analyzing different applications we found, *Empty* handlers are the second largest in Libraries and Stand-alone applications and the most common type of handler in Servers group. This outcome was obviously not anticipated in .NET programs. Checked exceptions can occasionally lead lazy developers to "silence exceptions" with *Empty* handlers only to be able to compile their project. But after analysis of the source code, we concluded that its usage in .NET is not related with compilation but with avoiding premature program termination on non-fatal exceptions. A peculiar example is the existence of various linear protected blocks containing different ways of performing an operation. This approach ensures that if one block fails to achieve its goal, the execution can continue to the next block

without any error being generated.

The most common action in the handlers of all the applications in Server-Apps and Stand-alone groups is *Logging errors*. One of the most common actions in the handlers of all the applications is also *Logging errors*. This typically corresponds to the generation of an error log, the notification of the user about the occurrence of a problem and the abortion of the task, it generally occurred in web applications and desktop applications. This plan is reinforced by the value of the *Return* action category in both of these application groups which is alike and the highest of all four groups.

The figure of *Alternative configuration* actions reports on the usage of alternative computation or object's state reconstruction when the code inside a protected block fails in achieving its objective. In some cases they are used to completely replace the code inside the protected block. These actions are by far the most individualized and specialized of all.

Asserts ensure that if an error occurs, the cause of the error is well known and reported to the user/programmer. *Assert* operations are the second most common error handling action in the *Libraries* applications group.

Others category actions are mainly related with thread stopping and freeing resources, in *Servers*, there is also a high distribution value for the *Others* category.

Throw action is another category with some weight in the global distribution. It is mainly due to the component based and layered development of software. Components and layers usually have a well defined interface between them. To encapsulate all kinds of exceptions into only one kind when passing an exception object between layers or software components is a moderately well-known approach. A new throw is typically generated.

Continue, Rollback, Close, Assert, Delegate and *Others* actions are rarely used in .NET. On the contrary; *Empty, Log, Alternative Configuration, Throw* and *Return* are the actions most frequently found in the *catch* handlers of .NET applications.

The results for *catch* handlers in Java programs for one type of application group at a time are also shown in Chart 2 to 5. We found some similarity with .NET in the Stand- alone and Server-Apps groups only. It is possible to see the same type of clustering found in .NET. The cluster of categories that concentrate the highest distribution of values is composed by *Empty, Log, Alternative Configuration, Throw* and *Continue* actions.

Empty category surprised us once again due to lower distribution values than the ones found in .NET. This implies that the checked exception mechanism has little or no weight on the decision of the developer to leave an exception handler empty: other logic must exist to substantiate the presence of empty handlers besides silencing exceptions. In .NET this occurs quite commonly for building alternative execution blocks. In Java exception mechanisms are also used as control/execution flow construct of the language apart from to handle "exceptional situations". (Note that even the Java API sometimes forces this. For example, the revelation of an end-of-file can only be done by being thrown an exception.)

The *Log* and *Throw* actions are highly correlated in Java. We found that in most of the cases, when an object is logged then it is also thrown. In *Server-apps*, *Server* and *Stand- alone* application groups the *Log* actions category takes the first place. In the *Libraries* group, *Log* is only outperformed by *Throw*, another popular action in the *Server-Apps* and *Server* groups.

Between 7% and 15% of all handlers terminate the method being executed are returning a value or not a value. A common action in all the application groups is *Return*.

The results for *finally* handlers for one application group at a time in .NET are shown in Chart 2 to 5. As compared to results for *catch* handlers the distribution of the several actions is different in outcomes for *finally* handlers. In our applications under test, *finally* handlers are principally used to closing the connections and release the resources. *Close* is the most common handler action category in .NET, for all application groups.

The second most used handler action in all application groups with the exception of only *Libraries* application group is *Alternative configuration*. Some type of conditional test that enables or not enables the execution of some predetermined configuration; is a peculiar block of code generally found in *finally* handlers. This alternative configuration is also done in some cases while resetting some state then they were classified as *Rollback* and not Alternative.

Others is another common category present in *finally* handlers of .NET applications. Actions include in *Others* category are deletion of file, stream flushing, event firing and termination of thread, among other less frequent actions. It is also prevalent to rollback previously done actions or reset object's state in *Server* applications.

Lastly, there are few empty *finally* blocks in *Stand-alone* applications.

In Chart 2 to 5, for Java applications the situation is very identical to the one found in .NET. *Close* is the most notable category in all application groups. Similar to .NET there are also some actions classified as *Others*, but in Java they have more weight in the distribution denoting a higher programming heterogeneity in exception handling.

In Java *finally* handlers, *Alternative configuration* and *Rollback* actions are also used as handler actions.

Lastly, in this study of exception handling, it is possible to notice that there is some undistinguished ground between application groups in .NET and Java. For the most part of the test suite, the most common actions in all *catch* handlers are *Empty* and *Log*, and the most used action in *finally* handlers is *Close*.

CONCLUSIONS

This paper tried to reveal how developers use the exception handling technique available in Java and C# i.e. for the two modern programming languages. And, even though we have particularized the outcomes separately for both environments and discovered few dissimilarities, in the principal outcomes are just identical.

We found that the amount of code used in error handling in .NET is much less than what would be expected and it is true even in Java where programmers are forced to declare or handle checked exceptions.

More significant is the confirmation that most of the exception classes used as *catch* arguments are largely general and do not serve as exact handling of errors, as expected. We have also observed that most of the times these exception handlers are empty or are solely dedicated to logging of exceptions and re-throwing of exceptions or return, exit the program, or method. On the contrary, the exception objects "caught" by these handlers are from very definitive types and firmly tied to application logic. This shows that, even though software developers are very involved in throwing the exception objects that best fit a specific exceptional situation, they are not so enthusiastic in implementing handling code with the same degree of specialization.

In general these outcomes lead us to the decision that, exceptions are not being rightly used as an error handling tool. This also indicates that if the programming community at large does not use them rightly, possibly it is a sign of a severe design fault in the mechanism: exception constructs, as they are, are not fully suitable for managing application errors. Work is indeed required on error handling mechanisms for programming languages. The minute detail of the errors occurred was not precisely and specifically deals by exception handlers. The largest desirable practice is logging the issue or notifying and alerting the end user about the error and terminate the on-going action. To "silence" exceptions, empty handlers were used; they frequently hide severe issues or stimulate bad utilization of programming language exception handling constructs.

Few of the troubles discovered, like the cloning or duplication of code between exception handlers, and the mixing of business logic code with exceptions handling code, among other problems are still to be handled and depict a significant research area.

Now we know, at best for this group of applications, what type of exceptions developers like to handle and what type of exceptions are generally caught. We would like to stretch our analysis to running software, actually accounting what type of exceptions do really occur and how this relates to the code developers are forced to write for error handling. Further, we would like to extend our work to include other categories of software projects (mobile-apps etc.) developed using various programming languages and in different environments.

References

[1] E. Gunnerson. C# and exception specifications. Microsoft, 2000. Available online at: http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820

[2] J. B. Goodenough. Exception handling: issues and a proposed notation. In Communications of the ACM, 18, 12 (December 1975), ACM Press.

[3] F. Cristian. Exception Handling and Software Fault Tolerance. In Proceedings of FTCS-25, 3, IEEE, 1996 (reprinted from FTCS-IO 1980, 97-103).

[4] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. In Journal of Systems and Software, 2, November 2001, 197-222.

[5] S. Sinha, and M. Harrold. Analysis and Testing of Programs with Exception-Handling Constructs. In IEEE Transactions on Software Engineering, 26, 9 (SEPTEMBER 2000), IEEE.

[6] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In Proceedings of ECOOP'97, LNCS 1241, Springer-Verlag, June 1997, 85-103.

[7] M. P. Robillard, G. C. Murphy. Designing robust JAVA programs with exceptions. In Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 25, 6 (November 2000), ACM Press.

[8] M. Lippert, and C. Lopes. A Study on Exception Detection and Handling Using Aspect- Oriented Programming. In Proceedings of the 22nd International Conference on Software Engineering, Ireland 2000, ACM Press, 2000.

[9] F. Filho, C. Rubira, and A. Garcia. A Quantitative Study on the Aspectization of Exception Handling. In Workshop on Exception Handling in Object-Oriented Systems (held in ECOOP 2005), Glasgow, Scotland, July 2005.

[10] T. Elrad, R. E., Filman, and A. Bader. Aspect-Oriented Programming. In Communications of the ACM, ACM Press, New York, USA, October 2001, Vol.44 (10), 29-32. ISSN 0001- 0782.

[11] T. Parr. ANTLR - Another Tool for Language Recognition. University of San Francisco, 2006. Available online at: http://www.antlr.org/.

[12] Javacc - Java Compiler Compiler. Available online at: https://javacc.dev.java.net/.

[13] S. Chiba. Load-Time Structural Reflection in Java. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00), Springer-Verlag, LNCS 1850, Sophia Antipolis and Cannes, France, June 2000.

[14] B. Cabral, P. Marques, L. Silva. RAIL: Code Instrumentation for .NET. In Proceedings of the 2005 ACM Symposium On Applied Computing (SAC'05), ACM Press, Santa Fé, New Mexico, USA, March 2005.

[15] J. Gosling, B. Joy, G. Steele, G. Bracha. The JAVA Language Specification. Sun Microsystems, Inc, Mountain View, California, U.S.A., 2000. ISBN 0-201-31008-21.

[16] ECMA International. Standard ECMA-335 Common Language Infrastructure (CLI). ECMA Standard, 2003. Available online at: http://www.ecma-international.org/publications/standards/ecma-335.htm.

[17] A. Goldberg , and D. Robson. Smalltalk-80: the language and its implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1983.

[18] B. Meyer. Eiffel: the Language. Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1992.ISBN 0-13-247925-7.

[19] B. Cabral, P. Marques. Making Exception Handling Work. In Proceedings of the Workshop on Hot Topics in System Dependability (HotDep'06), USENIX, Seattle, USA, November 2006.

[20] B. Cabral, P. Marques. Exception Handling: A Field Study in Java and .NET, Proceedings of the 21st European conference on Object-Oriented Programming, Berlin, Germany, August 2007.