Exception Handling: A Field Study in Java and .NET

Bruno Cabral, Paulo Marques

CISUC, Department of Informatics Engineering,
University of Coimbra, Portugal
{bcabral, pmarques}@dei.uc.pt

Abstract. Most modern programming languages rely on exceptions for dealing with abnormal situations. Although exception handling was a significant improvement over other mechanisms like checking return codes, it is far from perfect. In fact, it can be argued that this mechanism is seriously limited, if not, flawed. This paper aims to contribute to the discussion by providing quantitative measures on how programmers are currently using exception handling. We examined 32 different applications, both for Java and .NET. The major conclusion for this work is that exceptions are not being correctly used as an error recovery mechanism. Exception handlers are not specialized enough for allowing recovery and, typically, programmers just do one of the following actions: logging, user notification and application termination. To our knowledge, this is the most comprehensive study done on exception handling to date, providing a quantitative measure useful for guiding the development of new error handling mechanisms.

Keywords: Exception Handling Mechanisms, Programming Languages.

1 Introduction

In order to develop robust software, a programming language must provide the programmer with primitives that make it easy and natural to deal with abnormal situations and recover from them. Robust software must be able to perceive and deal with the temporary disconnection of network links, disks that are full, authentication procedures that fail and so on.

Most modern programming languages like C#, Java or Python rely on exceptions for dealing with such abnormal events. Although exception handling was a significant improvement over other mechanisms like checking return codes, it is far from perfect. In fact, it can be argued that the mechanism is seriously limited if not even flawed as a programming construct. Problems include:

- Programmers throw generic exceptions which make it almost impossible to properly handle errors and recover for abnormal situations without shutting down the application.
- Programmers catch generic exceptions, not proving proper error handling, making the programs continue to execute with a corrupt state (especially relevant in Java). On the other hand, in some platforms, programmers do

not catch enough exceptions making applications crash even on minor error situations (especially relevant in C#/.NET).

- Programmers that try to provide proper exception handling see their productivity seriously impaired. A task as simple as providing exception handling for reading a file from disk may imply catching an dealing with tens of exceptions (e.g. `FileNotFoundException`, `DiskFullException`, `SecurityException`, `IOException`, etc.). As productivity decreases, cost escalates, programmer's motivation diminishes and, as a consequence, software quality suffers.
- Providing proper exception handling can be quite a challenging and error prone task. Depending on the condition, it may be necessary to enclose *try-catch* blocks within loops in order to retry operations; in some cases it may be necessary to abort the program or perform different recovery procedures. Bizarre 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. Dealing with such issues correctly is quite difficult, error prone, not to say, time consuming.

To make things interesting, the debate about error handling mechanisms in programming languages has been recently fuelled with the launch of Microsoft's .NET platform.

Currently, the Java Platform and the .NET platform constitute the bulk of the modern development environments for commercial software applications. Curiously, Microsoft opted to have a different exception handling approach than in Java. In .NET the programmer 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 other hand, in Java, in most cases, the programmer 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 .NET's camp, the arguments for not having checked exceptions that are normally used are [1]:

- Checked exceptions interfere with the programmers' productivity since they cannot concentrate in business logic and are constantly forced to think about errors.
- Since the programmer 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 "exonerated" by exhaustive testing. I.e. 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.

Obviously, both camps cannot be 100% right. But, overall, the important message is that in order to develop high-quality robust software, in a productive way, new advances in error handling are needed. The existing mechanisms are not adequate nor suffice.

This paper aims to contribute to the discussion by providing quantitative measures on how programmers are currently using exception handling. We examined 32 different applications, both for Java and .NET, covering 4 different software categories (*libraries*; *stand-alone applications*; *servers*; and *applications running on servers*). Overall, this corresponds to 3,410,294 lines of source code of which 137,720 are dedicated to exception handling. For this work, we have examined and processed 18,589 *try* blocks and corresponding handlers. To our knowledge, this is the most comprehensive study done to date on exception handling.

The data presented on this paper is important to guide the development of new mechanisms and approaches to exception handling. Other results will help e.g. justify the feasibility of using existent methodologies, like applying Aspect Oriented Programming (AOP) to implement exception handlers as advices.

The rest of this paper is organized as follows: Section 2 discusses related work; Section 3 describes the application set used in this study; Section 4 explains the methodology used in the analysis; Section 5 presents the results of the tests and observations about their significance; finally, Section 6 concludes the paper.

2 Related Work

Since the pioneering 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 changed much. Nevertheless, programming languages designers have always suggested different approaches for implementing these mechanisms.

Several studies have been conducted over the years for validating the options taken in each different implementation. For instance, 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 twelve object-oriented languages. Each programming language was analyzed in respect to ten technical aspects associated with EH constructs: exception representation; external exceptions in signatures; separation between internal and external exceptions; attachment of handlers to program constructs (e.g. to statements, objects, methods, etc.); dynamism of handler binding; propagation of exceptions ; continuation of the flow control (resumption or termination); clean-up actions; reliability checks; and concurrent exception handling. After the evaluation of all the programming languages in terms of exception mechanisms, 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]. In the analysis, the authors also presented techniques to create intraprocedural and interprocedural representations of Java programs that contain EH constructs and an algorithm for computing control dependences in their presence. Using that work, the authors performed several studies and showed that 8.1% of the methods analyzed used some kind of exception mechanism and that these constructs had an important influence in control-dependence analysis.

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. Being specific: object-oriented programming does not support a complete exception specification (extra information may be needed for the exception context not supported by an object interface); state transitions are not always atomic in exception handling; exception information needs to be specific, but functions can be overloaded to have a different meaning in different situations; the exception handling control flow path is different from the normal execution path and is up to the programmer to differentiate both of them. Thus, the modification of object-oriented frameworks for adaptation to exception handling can have the following effects in terms of: *Abstraction*, change of abstraction levels and the usage of partial states; *Encapsulation*, the exception context may leak information that reveals or allows the access to the exception signaler private data; *Modularity*, design evolution may be inhibited by exception conformance; *Inheritance* anomalies may occur when a language does not support exception handling augmentation in a modular way.

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 and spaghetti-like exception handling code. The main factors that contribute to the difficulty of designing exception structures are the global flow of exceptions and the emergence of unanticipated exceptions. To help control these factors, the authors refined an existent software compartmenting technique for exception design and report about its usage in the rewriting of three Java programs and the consequent improvements they observed.

More recently, due to a new 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. Some of the results presented show that without aspects, the amount of code for exceptions is almost 11% of all the code; with aspects it represents only 2.9%. 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. F. Filho and C. Rubira conducted a similar study but they were not so enthusiastic in their results. The authors presented four metrics to evaluate the AOP approach to exception handling: separation of concerns; coupling between components and depth of inheritance tree; cohesion in the access to fields by pairs of method and advice; and dimension (size and number) of code. The work reports that the improvements of using AOP do not represent a substantial gain in any of the presented metrics showing that reusing handlers is much more difficult than is usually advertised. Handler reuse depends of the type of exceptions being handled, on what the handler does, the amount of contextual information needed; and what the method raising the exception returns and what the `throws` clause actually specifies.

The objective of this study is different from its predecessors. It does not target the quality of the mechanisms available in programming languages but the usage that programmers make of them. The emphasis is on understanding how programmers write exception handling code, how much of the code of an application is dedicated to error recovery and identifying possible flaws in their usage.

3 Workbench

The target platforms of this study were the .NET and Java environments, as well as the C# and Java programming languages.

Selecting a set of applications for the study was quite important. The code present in the applications had to be representative of common programming practices on the target platforms. Also, care had to be taken so that these would be "real world" applications developed for production use (i.e. not simply prototypes or beta versions). This was so in order not to bias the results towards immature applications where much less care with error handling exists. Finally, in order to be possible to perform different types of analyses, both the source code and the binaries of the applications had to be available.

Globally, we analyzed 32 applications divided into two sub-sets of 16 .NET programs and 16 Java programs. Each one of these sub-sets was organized in four categories accordingly to their nature:

- **Libraries**: software libraries providing a specific application-domain API.
- **Applications running on servers (Server-Apps)**: Servlets, JSPs, ASPs and related classes.
- **Servers**: server programs.
- **Stand-alone applications**: desktop programs.

The complete list of applications is shown in Table 1.

Table 1. Applications listed by group.

.NET	Libraries	SmartIRC4NET	IRC library
		Report.NET	PDF generation library
		Mono (corlib)	Open-source CLR implementation
		NLog	Logging library
	Server-Apps	UserStory.Net	Tool User Story tracking in Extreme Programming projects
		PhotoRoom	ASP.NET web site for managing on-line photo albums
		SharpWebMail	ASP.NET webmail application that is written in C#
		SushiWiki	WikiWikiWeb like Web application
	Servers	NeatUpload	Allows ASP.NET developers to stream files to disk and monitor progress
		Perspective	Wiki engine
		Nhost	Server for .Net objects
		DCSharpHub	Direct connect file sharing hub
	Stand-alone	Nunit	Unit-testing framework for all .NET languages
		SharpDevelop	IDE for C# and VB.NET projects
		AscGen	Application to convert images into high quality ASCII text
		SQLBuddy	SQL scripting tool for use with Microsoft SQL Server and MSDE
Java	Libraries	Thought River Commons	General purpose library
		Javolution	Real-time programming library
		JoSQL	SQL for Java Objects querying
		Kasai	Authentication and authorization framework
	Server-Apps	Exoplatform	Corporate portal and Enterprise Content Management
		GoogleTag Library	Google JSP Tag Library
		Xplanner	Project planning and tracking tool for Extreme Programming
		Mobile platform	Banks and mobile operators software for SMS and MMS services in cellular networks (not open-source)

Java	Servers	Jboss	J2EE application server
		Apache Tomcat	Servlet container
		JCGrid	Tools for grid-computing
		Berkeley DB	High performance, transactional storage engine
	Stand-alone	Compiere	ERP software application with integrated CRM solutions
		J-Ftp	Graphical Java network and file transfer client
		Columba	Email Client
		Eclipse	Extensible development platform and IDE

4 Methodology

The test applications were analyzed at source code level (C# and Java sources) and at binary level (metadata and bytecode/IL code) using different processes.

To perform the source code analysis two parsers were generated using antlr [11], for C#, and javacc [12] for Java. These parsers were then modified to extract all the exception handling code into one text file per application. These files were then manually examined to build reports about the content of exception handlers.

The source code of all application was examined with one exception. Due to the huge size of Mono, only its "corlib" module was processed.

The parsers were also used to identify and collect information about *try* blocks inside loops (i.e. detect *try* statements inside *while* and *do..while* loops). This is so because normally this type of operations corresponds to retrying a block of code that has raised an exception in order to recover from an abnormal situation.

The main objective of this article is to understand how programmers use the exception handling mechanisms available in programming languages. Nevertheless, the analysis of the applications source code is not enough by itself when trying to distinguish between the exceptions that the programmer wants to handle and the exceptions that might occur at runtime. This is so because the generated IL code/bytecode can produce more (and different) exceptions than the ones that are declared in the applications source code by means of `throw` and `throws` statements.

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

All data was stored on a relational database for easy statistical treatment.

Table 2. List of Assemblies and Java Packages analyzed.

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

For each application only one file (.NET) or package (and sub-packages) of classes (Java) was analyzed. Table 2 shows the names of the files and packages that were used in this study. The criterion followed to select these targets was the size of the files and their relevance in the implementation of the application core.

5 Results

In the following subsections we will present the results of this study, drawing some observations about their significance.

Nevertheless, we should caution that although the number of applications that were used was relatively large (32), it is not possible to generalize the observations to the whole .NET/Java universe. For that, it would be necessary to have a very significant number of applications, possible consisting in hundreds programs. Even so, due to the care taken in selecting the target applications, we believe that the results allow a relevant glimpse into current common programming practices in exception handling.

5.1 Error Handling Code in Applications

One important metric for understanding current error handling practices is the percentage of source code that is used in that task. For gathering this metric, we compared the number of lines of code inside all *catch* and *finally* handlers to the total number of lines of the program. The results are shown in Figure 1.

It is quite visible that in Java there is more code dedicated to error handling than in .NET. This difference can be explained by the fact that in Java it is compulsory to handle or declare all exceptions a method may throw, thus increasing the total amount of code used for error handling. Curiously, there is an exception to this pattern. In the

Server Application group, the difference is almost non-existent. To explain this result we examined the applications' source code. For this class of applications, both in Java and .NET, programmers wrote quite similar code. Meaning that they expect the same kind of errors (e.g. database connections loss, communication problems, missing data, etc.) and they use the same kind of treatment (the most common handler action in this type of applications is logging the error).

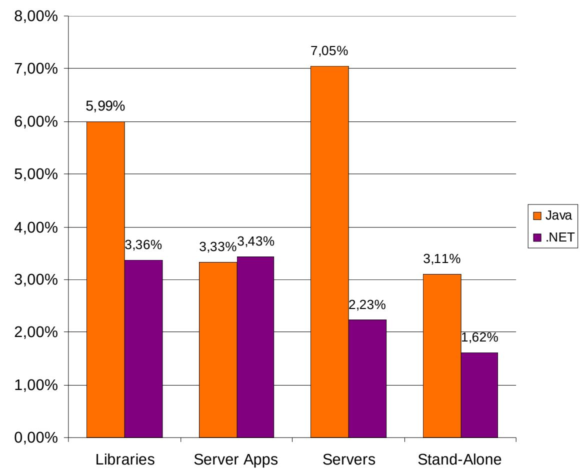

Fig. 1. Amount of error handling code.

One surprising result is that the total amount of code dedicated to exception handling is much less than what would be expected. This is even more surprising in Java where using exceptions is almost mandatory even in small programs. Our results show that the maximum amount of code used for error handling was 7% in the *Servers* group. Overall, the result is 5% for Java, with a 2% standard deviation, and 3% for .NET, with a standard deviation of 1%. It should be noted the applications used in this study are quite mature, being widely used. We reason that the effort dedicated to writing error protection mechanisms is not as high as expected, even for highly critical applications like servers. The forceful of declaring and catching checked exceptions in Java effectively increases (almost doubles) the amount of error handling code written, even though it is still represents a small fraction of all the code of an application. The critical issue is that normally error handling code is being used more to alert the user, to abort the applications or to force them to continue their execution, than to actually recover from existing errors.

5.2 Code in Exception Handlers

Apart from measuring the amount of the code that deals with errors, to find out how programmers use exception handling mechanisms, it is important to know what kind of actions are performed when an error occurs.

To be able to report on this subject we had to inspect sets of ten thousand lines of application source code. As a matter of fact, we covered all the handlers (*catch* and *finally*) in all the applications except for JBoss and Eclipse. For these two, due to their dimension, only 10% of the 96,405 lines of code existing inside of exception handlers were examined. Even so, they are representative of the rest.

Table 3. Description of the Handler's actions categories.

Category	Description
Empty	The handler is empty, is has no code and does nothing more than cleaning the stack
Log	Some kind of error logging or user notification is carried out
Alternative/ Static Configuration	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
Throw	A new object is created and thrown or the existing exception is re-thrown
Continue	The protected block is inside a loop and the handler forces it to abandon the current iteration and start a new one
Return	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
Rollback	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)
Close	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
Assert	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
Delegates (only for .NET)	A new delegate is added
Others	Any kind of action that does not correspond to the previous ones

To simplify the classification of these error handling actions we propose a small set of categories that enable the grouping of related actions. These categories are summarized in the previous table.

Note that an exception handler may contain actions that belong to more than one category. In fact, this is the common case. For instance, a handler can log an error, close a connection and exit the application. These actions are represented by three distinct categories: Log, Close and Return. Thus, in the results, this handler would be classified in all these three categories.

Since *catch* and *finally* handlers have different purposes, we opted for doing separate counts for each type of handler. Finally, the distribution of handler actions for each application was calculated as a weighted average accordingly to the number of actions found in each application. This is so that small applications do not bias the results towards their specific error handling strategy.

The results obtained for each application group are shown in next four graphs.

The graph of Figure 2 shows the average of results by application group for .NET *catch* handlers. In the four application groups 60% to 75% of the total distribution of handler actions is composed of three categories: Empty, Log and Alternative Configuration.

Empty handlers are the most common type of handler in *Servers* and the second largest in *Libraries* and *Stand-alone applications*. This result was completely unexpected in .NET programs since there are no checked exceptions in the CLR and, therefore, programmers are not obliged to handle any type of exception. Checked exceptions can sometimes lead lazy programmers to “silence exceptions” with empty handlers only to be able to compile their applications. From the 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 typical example is the presence of several linear protected blocks containing different ways of performing an operation. This technique assures that if one block fails to achieve its goal, the execution can continue to the next block without any error being generated.

Logging errors is also one of the most common actions in the handlers of all the applications. In fact, is the most common action in *Server-Apps* and *Stand-alone* groups? Considering web applications and desktop applications, 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. This idea is re-enforced by the value of the Return action category in these two application groups which is the identical and the highest of all four groups.

The number 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. These actions are by far the most individualized and specialized of all. In some cases they are used to completely replace the code inside the protected block.

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

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

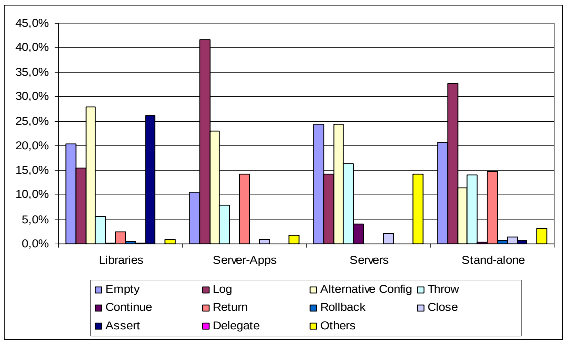

Fig. 2. Catch handler actions count for .NET programs.

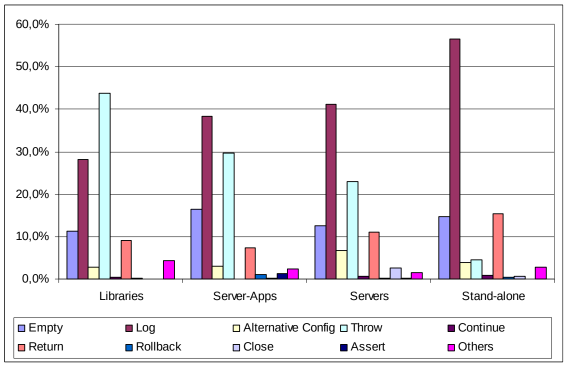

Fig. 3. Catch handlers' actions count for Java programs.

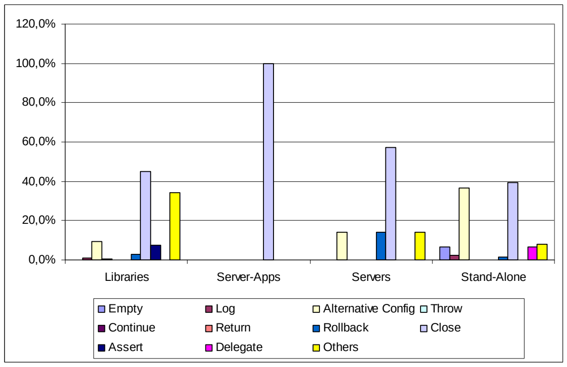

Fig. 4. Finally handlers' actions for .NET programs.

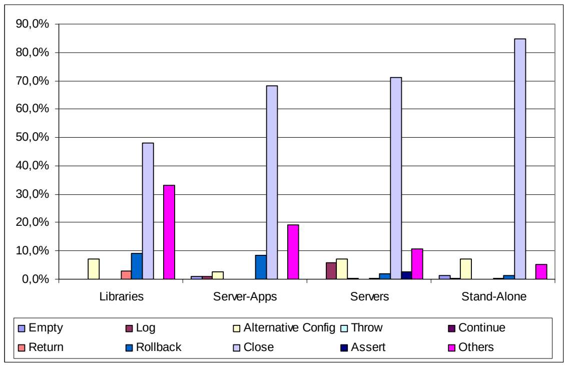

Fig. 5. Finally handlers' actions for Java programs.

Another category of actions with some weight in the global distribution is the Throw action. This is mainly due to the layered and component based development of software. Layers and components usually have a well defined interface between them. It is a fairly popular technique to encapsulate all types of exceptions into only one type when passing an exception object between layers or software components. This is typically done with a new `throw`.

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

Figure 3 shows the results for *catch* handlers in Java programs. Only in the *Stand-alone* and *Server-Apps* groups we found some similarity with .NET. Despite this fact, 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.

The distribution values on the Empty category surprised us once again. This value is lower than the ones found in .NET. This suggests that the checked exception mechanism has little or no weight on the decision of the programmer to leave an exception handler empty: another reason must exist to justify the existence of empty handlers besides silencing exceptions. In .NET this happen quite frequently for building alternative execution blocks. We risk saying that in Java exception mechanisms are no longer being used only to handle "exceptional situations" but also as control/execution flow construct of the language. (Note that even the Java API sometimes forces this. For instance, the detection of an end-of-file can only be done by being thrown an exception.)

The Log actions category takes the first place for *Server-apps*, *Server* and *Stand-alone* application groups and the second place in *Libraries* group. In this last group, Log is only surpassed by Throw, another popular action in the *Server-Apps* and *Server* groups. In Java, the Log and Throw actions are highly correlated. We observed that in the majority of cases, when an object is thrown the reason why it happens is also logged.

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

Figure 4 illustrates the results for *finally* handlers in .NET. The distribution of the several actions is different from the one found in *catch* handlers. Nevertheless, is visible that the most common handler action category in .NET, for all application groups, is Close. I.e. *finally* handlers, in our test suite, are mainly used to close connections and release resources.

Alternative configuration is the second mostly used handler action in all application groups with the exception of *Libraries*. A typical block of code usually found in *finally* handlers is composed by some type of conditional test that enables (or not) the execution of some predetermined configuration. In some cases, these alternative configuration is done while resetting some state. In those cases, they were classified as Rollback and not Alternative.

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

Finally, on *Stand-alone* applications there are some empty *finally* blocks that we can not justify since they perform no easily understandable function.

In Java applications (Figure 5) the scenario is very similar to the one found in .NET. Close is the most significant category in all application groups. There are also some actions classified as Others, which are similar to the ones of .NET. In Java they have more weight in the distribution, indicating a higher programming heterogeneity in exception handling.

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

It is possible to observe that there is some common ground between application groups in Java and .NET in what concerns exception handling. For the most part, Empty and Log the most common actions in all *catch* handlers and Close is the most used action in *finally* handlers.

5.3 Exception Handler Argument Classes

After identifying the list of actions performed by handlers, we concentrated on finding out if there is some relation between *catch* handlers for the same type of exception classes. For this, we developed two programs: one to process .NET's IL code and another to process Java bytecode. These IL code/bytecode analyzers were used to discover what exceptions classes were most frequently used as *catch* statement arguments. We opted by performing this analysis at this level and not at source code level because it is simpler to obtain this information from assemblies or class files metadata than from C# or Java code.

Figure 6 shows the most common classes used as argument of `catch` instructions in .NET applications. The results are grouped by application type and the values represent the weighted average of the distribution among applications of the same group. Thus, programs with the largest number of handlers have more weight in the final result.

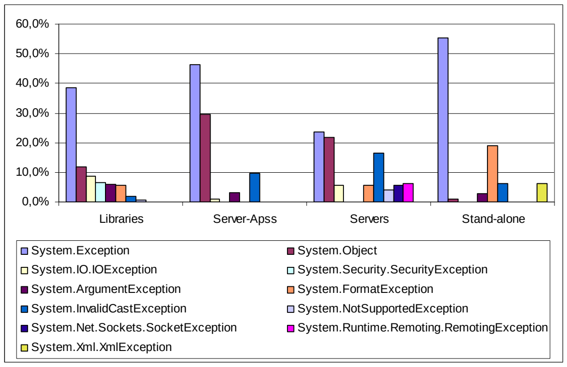

Fig. 6. .NET catch's arguments classes.

It is possible to observe that programmers prefer to use the most generic exception classes like `System.Exception` and `System.Object` for catching exceptions. Note that .NET, not C#, allows any type of object to be used as an exception argument. When the argument clause of a *catch* statement is left empty, the compiler assumes that any object can be thrown as an exception. This explains the large presence of `System.Object` as argument.

The use of generic classes in *catch* statements can be related to the two of the most common actions in handlers: Logging and Return. This means that for the largest set of possible exceptions that can be thrown, programmers do not have particular exception handling requirements: they just register the exception or alert the user of its occurrence. Nevertheless, there are a lot of handlers that use more specific exception classes. These different handlers do not have any weight by themselves in the distribution but all the code that actually tries to perform some error recovery operations is concentrated around these specialized handlers.

I/O related exception handlers are fairly used in *Libraries* and *Servers*. Also invalid arguments types, number and format errors are treated as exceptions by all the applications as shown by the presence of `System.ArgumentException` handlers and `System.FormatException` handlers.

There are not many differences between Java and .NET in terms of *catch* arguments. Figure 7 shows the results for Java. It is possible to conclude that the most generic exception classes are the preferred ones: `Exception`, `IOException`, and `ClassNotFoundException`. We tried to found out why `ClassNotFoundException` is so commonly used by analyzing the source code. For the most part, most of the handlers associated to the use of this class are empty, just log the error or throw a new kind of exception. Others try to load a parent class of the class not found or another completely different class. In general, these handlers are associated with "plug-in" mechanisms or modular software components using dynamic class loading.

Finally, we did an analysis of all the applications source code to find out what was the distribution of handler actions by *catch* handler argument class for the most commonly used classes. The results can be found in Figure 8.

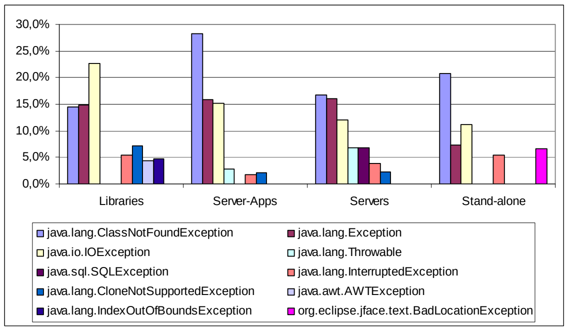

Fig. 7. Java catch's arguments classes.

The results are quite different from one type of exception class to another. Even so, it is still possible to say that the dominant handler actions are the ones belonging to the categories: Empty, Log, Alternative Configuration, Throw and Return.

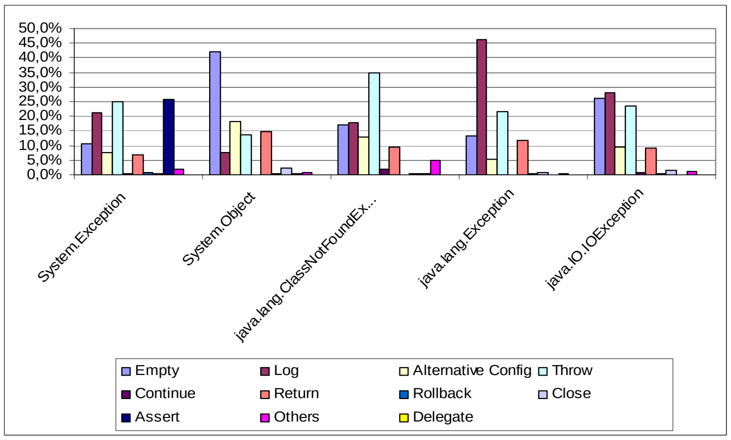

Fig. 8. Handler action distribution for the most used catch handler classes.

It is interesting to notice that in .NET `catch` instructions with no arguments are directly associated with the largest number of Empty handlers.

In Java, in particular for `ClassNotFoundException`, alternative configuration actions are common. This behavior is understandable if we consider that, if a class is not found then a new one should be suggested as alternative. (This is quite common in database applications, while loading JDBC drivers.)

5.4 Handled Exceptions

On the last section, we reported the exceptions that are used in *catch* statements. Nevertheless, a *catch* statement can catch the specific exception that was listed or more specific ones (i.e. derived classes). We will now discuss exception handling code from the point of view of possible handled exceptions. As described in section 4 we used IL code/bytecode analyzers to collect all the exceptions that the applications could throw because this information is not completely available at source code level. I.e. the set of exceptions that an application can throw at runtime is not completely defined by the applications source code `throw` and `throws` statements. Therefore, a profound analysis of the compiled applications was required for gathering this information.

5.4.1 Exception Universe

In Java, thanks to the checked exception mechanism, we are able to discover and locate all the exceptions that an application can throw by analyzing its bytecode and metadata. To know what exceptions may be thrown by a method it is necessary to know:

- All the exceptions that the bytecode instructions of a method may raise accordingly to the Java specs [15]
- All the exception classes declared in the `throws` statement of the methods being called
- All the exceptions that are produced inside a protected block and are caught by one of its handlers
- All the exception classes in the method own `throws` statement

In .NET this is a more difficult task because there are no checked exceptions. To discover what exceptions a method may raise is necessary to know:

- All the exceptions that can be raised by each one of the IL instructions accordingly to the ECMA specs of the CLR [16]
- All the exceptions that the method being called may raise
- All the exception classes present in explicit `throw` statements
- All the exceptions that are produced inside a protected block and are not caught by one of its handlers

When we started to work on which exceptions could occur in .NET and Java, the results of the analysis were quite biased. This happened because:

- Almost all instructions can raise one or more exceptions, accordingly to CLR ECMA specs and Java specs, making the total number of exceptions reported grow very fast and the occurrence of other types of exceptions not directly associated with instructions almost irrelevant;
- In most cases, the exceptions that each low-level instruction could actually throw would not indeed occur since some code in the same method would prevent it (e.g. an explicit program termination if a database driver was not found, thus making all `ClassNotFoundException` exceptions for that class irrelevant). Since it is not possible to detect this code automatically, although the results could be correct, the analysis would not reflect the reality of the running application or the programming patterns of the developer.

To obtain meaningfully results we decided to perform a second analysis not using all the data from the static analysis of bytecode and IL code instructions. In particular, we filtered a group of exceptions that are not normally related to the program logic, and that the programmer should not normally handle, considering the rest. The list of exceptions that were filtered (i.e. not considered) is shown in Table 4.

Table 4. Java and .NET exception classes for bytecode and IL code instructions.

JAVA	.NET
java.lang.NullPointerException	System.OverflowException
java.lang.IllegalMonitorStateException	System.Security.SecurityException
java.lang.ArrayIndexOutOfBoundsException	System.ArithmeticException
java.lang.ArrayStoreException	System.NullReferenceException
java.lang.NegativeArraySizeException	System.DivideByZeroException
java.lang.ClassCastException	System.Security.VerificationException
java.lang.ArithmeticException	System.StackOverflowException
	System.OutOfMemoryException
	System.TypeLoadException
	System.MissingMethodException
	System.InvalidCastException
	System.IndexOutOfRangeException
	System.ArrayTypeMismatchException
	System.MissingFieldException
	System.InvalidOperationException

5.4.2 Results for handled exceptions

Being aware of the complete list of exceptions that an application can raise and of the complete list of handlers and protected blocks, it is possible to find out which are the most commonly handled exception types. The results for .NET applications are shown in Figure 9; the values represent the average of results by application group where every application had a different weigh in the overall result according to the total number of results that they provided. It is possible to observe that the results are very different from application group to application group. For instance, in the *Libraries* group, the most commonly handled exceptions are `ArgumentNullException` and `ArgumentException`, resulting from bad parameter use in method invocations. In the remaining three groups the number one exception type is Exception, this can be a symptom of the existence of a larger and more differentiated set of exceptions that can occur. If many different exceptions can occur it is viable to assume that the most generalized type (i.e. `Exception`, `IOException`, etc.) becomes the most common one.

Seeing exception types like `HttpException`, `MailException`, `SmtpException` and `SocketException` in this top ten list and observing a distribution with such variations from application group to application group, we are confident to say that the type of exceptions that an application can raise and, in consequence, handle is strictly related with the application nature.

There is a mismatch between the type of classes used as arguments to `catch` instructions and the classes of the exceptions that are handled, i.e. throw statements use the exception classes that best fit the situation (exception) but the handlers that will eventually “catch” these exceptions use general exception classes like .Net’s and Java’s Exception as their arguments.

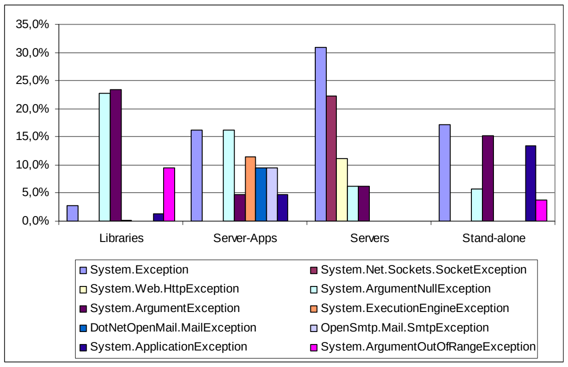

Fig. 9. Most commonly handled exception types in .NET.

In Java, as in .NET, there is a large spectrum of exception types being handled. The results for Java are illustrated in Figure 10. The huge distinction helps to differentiate `IOException` as the most "caught" exception type in all application groups. It is also possible to observe that the exception types are tightly related to the applications. For instance in *Stand-alone* applications, three of the exception classes are from Eclipse. Due to its size Eclipse carries a large weight in its application group results and, as we are able to observe, its "private" exceptions are present in this top ten.

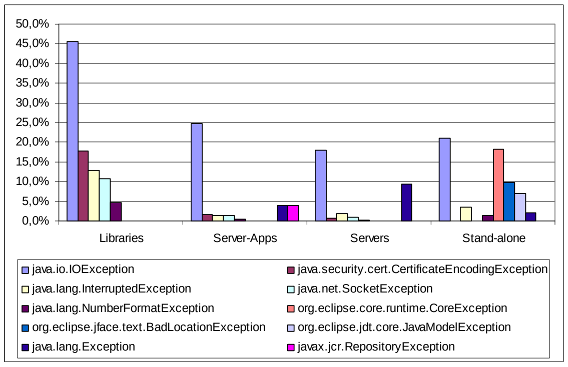

Fig. 10. Most commonly handled exceptions in Java.

5.4.3 Call Stack Levels Analysis

The analysis of the applications bytecode and IL code allows us to discover the number of levels in the call stack that an exception travels before it is caught by some handler. Note that an exception is caught if the *catch* argument class is the same of the exception or a super-class of it.

One result that we can directly associate with the checked exceptions mechanism is the difference in the number of levels that an exception covers before it is caught by some handler in Java and .NET.

In Figure 11 it is possible to observe that in Java almost 80% of the exceptions are caught one level up from where they are generated, 15% two levels up, 5% three levels up and all the remaining are caught as high as five levels. On the other hand, in .NET, exceptions can cover up to seventeen levels and the distribution of the exceptions per levels covered is much sparser than in Java. The .NET programmer is not forced to catch exceptions and, as a result, exceptions can be caught much later in the call stack and most of times by exception handlers with general *catch* arguments.

In .NET, 5% of the exceptions are caught before they cover any level in the call stack. This result is unexpected and could only be explained by a detailed analysis of the IL code in the assemblies and of the source code of the programs. At first we thought that this could be the result of some code tangling at compile time but the analysis showed that the exceptions were originated in throw instructions inside the protected blocks of methods. Programmers raised these exceptions to pass the execution flow from the current point in the method to code inside a handler – i.e. they use exceptions as a flow control construct.

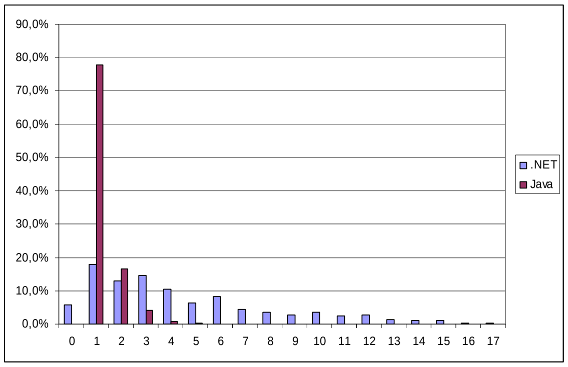

Fig. 11. Call stack levels for caught exceptions.

5.4.4 Handler size

Another interesting measure that we withdraw from the analysis of assemblies IL code and metadata was related with handler's code size or, more precisely, the count of opcodes inside a handler. This analysis could only be conducted in .NET because the metadata in the assemblies clearly identifies the *begin* and *end* instructions for each handler while in Java only the information about the beginning of a handler is available. To discover where a handler finishes we would have to do a static flow control analysis and find the join point in the code after the first instruction in the handler, which is outside of the scope of this paper.

The graph in Figure 12 shows that the largest set of handlers in *Server-Apps*, *Servers* and *Stand-alone* applications groups have 8 IL Code instructions. In the *Libraries* group more than 40% of the handlers have 3 instructions. The second largest set of handlers in all groups has 5 instructions. Obviously, there are bigger handlers but their number is so low that we excluded them from the graph to improve its reading.

These results made us curious about what was happening in these handlers and what were the instructions in question. We analyzed all the IL code in all the handlers and found some interesting facts:

- In the 526 handlers with size 8, 500 (95%) invoked a `Dispose()` method in some object; from this 500 there were two major sets of handlers with the exact same opcodes, one with 329 elements and the other with 166; the remaining 5 handlers were different between them; these handlers were all *Finally* handlers.
- In the set of handlers with 5 instructions there were 194 elements; 74 disposed of some object; 24 created and throwed a new exception; 36 stored some value.
- 484 of the 498 handlers of size 3 were *Finally* handlers; 426 handlers had exactly the same opcodes and were responsible for closing a database connection; other 34 handlers also had the same code and invoked a `Finalize` method in some object.
- The largest set of handlers with size 2 was empty handlers in the source code and its actions consisted in cleaning the stack and returning; others rethrowed the exception, and the rest called some `Assert` method.

These lead us to the conclusion that many of the handlers with few instructions are very similar between them and that the majority are *Finally* handlers that do some kind of method dispose or connection closing.

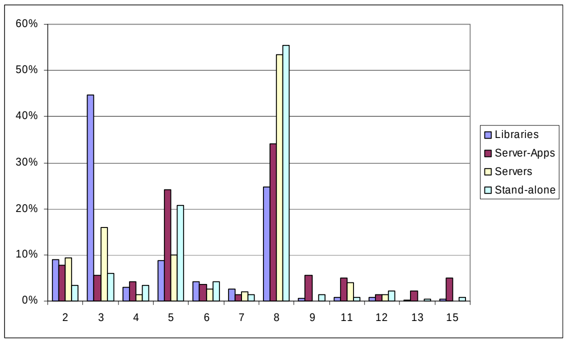

Fig. 12. Handlers size in number of IL code instructions for .NET.

5.4.5 Types of handlers

Knowing that the majority of the handlers with few instructions were *finally* blocks we tried to discover which was the relation between the total number of protected blocks, the total number of *catch* handlers and the total number of *finally* handlers.

The data in Table 5 shows that for the 1565 protected blocks found in the .NET applications there are 1630 handlers; 1144 protected bocks (73%) have *finally* handlers; but only 29% have *catch* handlers. On Java there are 18389 handlers distributed by 17024 protected blocks; 8109 protected blocks (48%) have *finally* handlers; 9402 (55%) have *catch* handlers.

Table 5. Number of protected blocks, catch handlers and finally handlers.

	Protected Blocks	Handlers	Protected Blocks with Finally Handlers	Protected Blocks with Catch Handlers
.NET	1565	1630	1144	450
Java	17024	18389	8109	9402

In our test set of applications, .NET programmers use much more *finally* handlers, relatively to the total number of handlers, than Java programmers.

In the graph of Figure 13 it is possible to see that Java applications have higher maximum values of *catch* handlers per protected block, the average number of catch blocks per *try* block is almost identical in all the application groups for the two platforms and has the approximate value of one. The standard deviation values are also very low meaning that the largest number of protected blocks has only one *catch* handler.

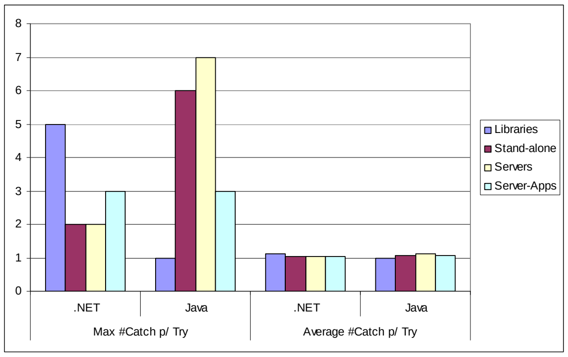

Fig. 13. Number of catch handlers per protected block.

5.4.6 Checked vs. Unchecked Exceptions

As mentioned before, the checked exceptions mechanism influences the way Java programmers use the exception detection and handling language constructs. But programmers can, alternatively, use unchecked exceptions in Java. For instance, there are some libraries specialized in using only unchecked exceptions (e.g. Java NIO).

In the programs that were analyzed, we compared the number of `catch` instructions that have an unchecked exception class as argument with the total number of `catch` instructions. The results are displayed in Table 6. It is possible to observe that except for the *Stand-Alone* application group, where the usage reaches 36.7%, for the remaining groups, values are very low, never exceeding 9%. Nevertheless, unchecked exceptions are indeed being used and, besides their extensive usage by some dedicated libraries, they are largely used to report on underlying system errors.

Table 6. Usage of Unchecked exceptions in Java catch handlers.

	Unchecked
Libraries	8,90%
Servers	8,50%
Stand-Alone	36,70%
Server-Apps	6,50%

5.4.7 Retry functionality

Neither Java or .NET have nothing like a "retry" block functionality that would enable the programmer to execute a *try* block in a cycle until it succeeds or reaches a certain condition. Other languages like Smalltalk [17] or Eiffel [18] have this kind of construct.

In Java and .NET, if a programmer wants to mimic this functionality he has to insert a protected block inside a cycle, for instance, insert a *try* block inside a *while* or *do-while* cycle.

Using source code parsers for accounting the number of protected blocks found inside cycles or loops we were able to obtain the total number of these occurrences. In Java we found 1082 cases and in .NET 16.

This analysis can be considered as some sort of blind analysis because we do not know if the programmer really intended to do a "retry". Nevertheless, 6% of all *catch* handlers were inside loops and if the programmer really intended to do a "retry", which seams to be the most reasonably reason, that would be a fairly interesting result to justify the addition of this functionality to both languages.

5.5 Making Exception Handling Work.

The results discussed in the previous sections show that programmers, most of the time, do not use exception handling mechanisms correctly or, at least, they do not use them for error recovery. These practices lead to a decrease in software quality and dependability. It is clear that in order to develop high-quality robust software, in a highly productive way, new advances are needed. Some authors have already started looking for new approaches. In our line of work we are currently approaching the problem by trying to create automatic exception handling for the cases where "benign exception handling actions" can be defined (e.g. compressing a file on a disk full exception). In general, we are trying to free the programmer from the task of writing all the exception handling code by hand, forcing the runtime itself to automatically deal with the problems whenever possible. A complete description of the technique is out of scope of this paper, but the interested reader can refer to [19] for a discussion of the approach.

6 Conclusion

This article aimed to show how programmers use the exception handling mechanisms available in two modern programming languages, like C# and Java. And, although we have detailed the results individually for both platforms and found some differences, in the essential results are quite similar. To our knowledge, this is the most extensive study done on exception handling by programmers in both platforms.

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

More important is the acknowledgment that most of the exception classes used as *catch* arguments are quite general and do not represent specific treatment of errors, as one would expect. We have also seen that these handlers most of the times are empty or are exclusively dedicated to log, re-throw of exceptions or return, exit the method, or program. On the other hand, the exception objects "caught" by these handlers are from very specific types and closely tied to application logic. This demonstrates that, although programmers are very concerned in throwing the exception objects that best fit a particular exceptional situation, they are not so keen in implementing handling code with the same degree of specialization.

These results lead us to the conclusion that, in general, exceptions are not being correctly used as an error handling tool. This also means that if the programming community at large does not use them correctly, probably it is a symptom of a serious design flaw in the mechanism: exception constructs, as they are, are not fully appropriate for handling application errors. Work is needed on error handling mechanisms for programming languages. Exception handlers are not specific enough to deal with the detail of the occurring errors; the most preferable behavior is logging the problem or alerting the user about the error occurrence and abort the on-going action. Empty handlers, used to "silence" exceptions, will frequently hide serious problems or encourage bad utilization of programming language error handling constructs.

Some of the problems detected, like the duplication of code between handlers, and the mingling of business code with exceptions handling code, among other problems are still to be tackled and represent an important research target.

We now know, at least for this set of applications, what type of exceptions programmer prefer to handle and what type of exceptions are commonly caught. In the future we would like to extend our analysis to running software, actually accounting what type of exceptions do really occur and how this relates to the code programmers are forced to write for error handling.

Acknowledgments. This investigation was partially supported by the Portuguese Research Agency – FCT, through a scholarship (DB/18353/98), and by CISUC (R&D Unit 326/97).

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, 2000.
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. 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.
14. 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.
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.