Research On Exception Handling

Author: Vaishnavi G. Shende

Vaishnavi G. Shende M.C.A. Tilak Mahavidhyalay Pune Maharastra

Ex prof.: Mrs.Shweta Nigam

---***---

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.

2. 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 , for C#, and javacc 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 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 to read class files and extract exception handler information. All data was stored on a relational database for easy statistical treatment..

3. MODELING AND ANALYSIS

In this section we identify the possible sources of Java language-level exceptions, pro-pose a mechanism for transforming them to UML state chart events and introduce a pattern (state chart design convention) for handling the events in the state chart similar ly as exceptions are handled in Java programs. We model event-driven systems by using UML State harts. The State Machine package of UML specifies a set of basic concepts (states and transitions) and several advanced features (state hierarchy, orthogonal decomposition, history states etc.) to be used for modeling discrete behaviour through finite state-transition systems. The. operational semantics is expressed informally by the standard in terms of the operations of a hypothetical machine that implements a state chart specification. The example discussed in this article is the traffic supervisor system in the crossing of a main road and a country road. The controller provides higher precedence to the main road, i.e., it does not wait until the normal time of switching from red to yellow-red if more than two cars are waiting at the main road (the arrival of a car is indicated by a sensor). Cars running illegally in the crossing during the red signal are detected by a sensor and recorded by a camera. For simplicity reasons only the statechart dia-gram of the light control of the main road is investigated here (Fig. 1).

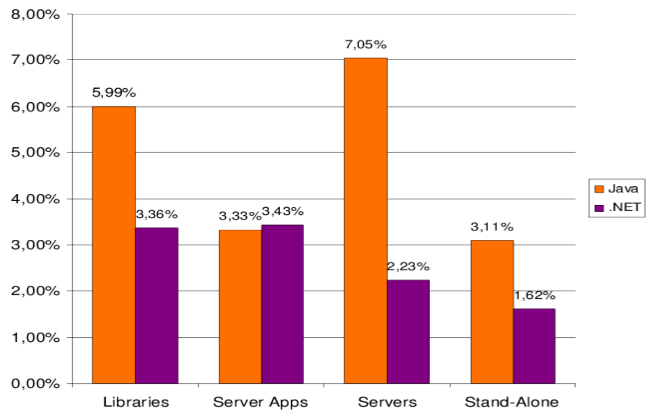

Fig 1. Amount of error handling code.

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 connections loss, communication problems, missing data, etc.) that they expect the same kind of errors (e.g. database and they use the same kind of treatment (the most common handler action in this type of applications is logging the error).

.

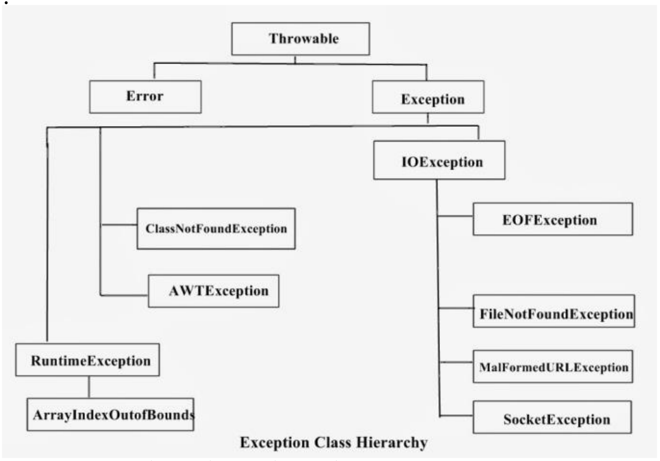

Fig 2. Exception Class Hierarchy

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
• 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
• 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:
• 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)

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 Fig. 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 argumentsIn 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

• Libraries: software libraries providing a specific application-domain API.

• Applications running on servers (Server-Apps): Servlets, JSPs, ASPs and related classes. • Servers: server 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.

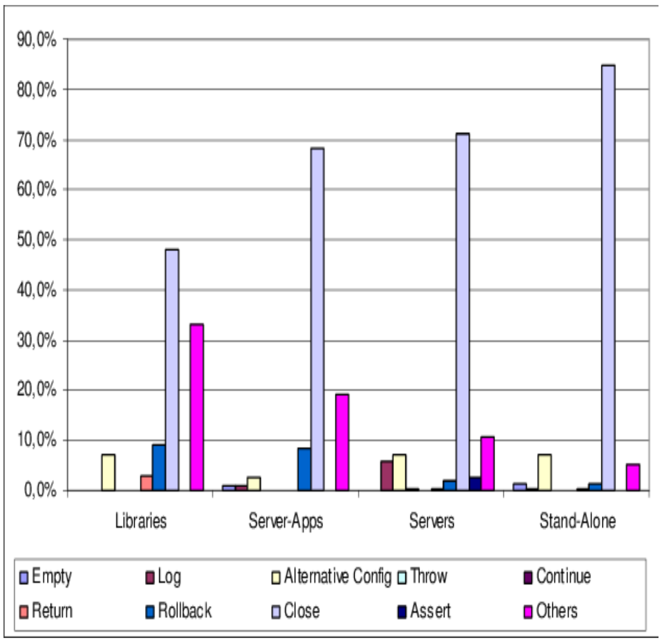

Fig 3. Finally handlers' actions for Java programs

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 (Fig3.) 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

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:application group results and, as we are able to observe, its "private" exceptions are present in this top ten.c

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 ormore 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.

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.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

CONCLUSIONS

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

ACKNOWLEDGEMENT:

I would like to express my sincere gratitude to my advisor Prof & H.O.D. of M.C.A. Department Mrs. Shweta Nigam for the continuous support of my MCA study & research for her patience, motivation , enthusiasm , and immense knowledge .Her guidance helped me in all the time of research and writing of this paper. I could not have imagined having a better advisor and mentor for my MCA study.

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.