Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions?

Paulo Sacramento, Bruno Cabral, Paulo Marques*

CISUC, University of Coimbra, Portugal
Dep. Eng. Informática, Pólo II – Univ. Coimbra,
3030-290 Coimbra, Portugal
{sacra, bcabral, pmarques}@dei.uc.pt

Abstract. The emergence of exception handling mechanisms in modern programming languages made available a different way of communicating errors between procedures. For years, programmers trusted in the correct documentation for error codes returned by procedures to correctly handle erroneous situations. Now, they have to focus on the documentation of exceptions for the same effect. But to which extent can exception documentation be trusted? Moreover, is there enough documentation for exceptions? And in which way do these questions relate to the checked vs. unchecked exceptions discussion? For a given set of Microsoft .NET applications, code and documentation were thoroughly parsed and compared. This showed that exception documentation tends to be scarce and of poor quality when existent. In particular, it showed that 90% of exceptions are undocumented. Furthermore, programmers were demonstrated to be keener to document exceptions they explicitly throw while typically leaving exceptions resulting from method calls undocumented. These results are a contribution to the assessment of the effectiveness of the unchecked exceptions approach.

Keywords: Exception Handling, Reliability, Documentation, .NET

1 Introduction

Since the appearance of exception handling mechanisms, several years ago [1], their importance has been steadily increasing. Being a part of modern object-oriented programming languages like Sun's JAVA and Microsoft's .NET languages, exceptions have been slowly replacing error codes widely used in procedural languages like C.

Whether using error codes or exceptions, if a programmer intends his code to be used by anyone, even himself, he has to spend time and effort documenting his methods, explaining the circumstances in which a given erroneous situation can occur and what the methods do in that event.

* Corresponding author.

This documentation process, being mostly a manual one, is subject to incompleteness and plain error. Documentation absence and errors are bound to cause programming errors, because unless there is a way of examining the source code of the software module a programmer is interacting with, documentation is all he can trust.

These problems are known by programmers who spend their lives wrestling with bad documentation, and especially by those used to the older way of error code identifiers. Nowadays, error codes can still be used, because in modern programming languages, exception handling is mostly an optional way of dealing with erroneous situations. In fact, it is possible, though not likely, to code a whole program in a modern object-oriented programming language without paying any attention to exceptions.

All this means that the extension of the problem of documentation absence and poor quality in modern programming languages is not known. The purpose of this paper is to determine that extension, using a specially designed software tool to analyse a representative group of software components written through and for Microsoft's .NET platform [2].

As we will see later in the paper, this analysis steers us towards the discussion of *checked vs. unchecked exceptions*, which we introduce in Sections 2 and 3 and subsequently address substantially. Given their approaches, at present time this discussion also means *Sun vs. Microsoft*. Sun includes checked exceptions in Java and thus believes that there are certain exceptions which impact the functionality of a method so importantly that they should be explicitly signalled and a programmer using this method should be forced to handle them. Microsoft, on the other hand, does not include support for these exceptions - all of them are unchecked - and so, programmers can ignore all exceptions that a method throws. In either case, documenting exceptions is important because if a programmer intends to handle an exception, whether because he is forced, whether because he wants to, he should know the circumstances in which the exception may occur.

Section 2 of the paper presents some background information needed for a better understanding of the discussed topics. Section 3 discusses related work, which is extensive, while Section 4 concerns our whole analysis methodology. Finally, Section 5 shows the collected results and Section 6 concludes the discussion.

2 Background

The target system for this study was the Microsoft .NET Platform [2]. Although exception handling is similar between different platforms and, to some extent, it is possible to extrapolate the conclusions to other platforms, there are differences. So, some background information needs to be given on the way exception handling works, with special emphasis on .NET.

Try-catch-finally blocks work essentially the same way in .NET as in JAVA or "old" C++[1]. They protect a portion of code in the sense that if some exception gets thrown in that portion and a corresponding *catch* or *finally* clause exists, that clause is made responsible for the handling of that exception. It is up to the programmer to decide what handling means in each case. Normal resumption of execution can be possible in some cases. If a proper clause does not exist, the exception is propagated to the "upper" method in the call stack. This process continues until such a clause is found in one of the methods in the call stack or the call stack is emptied, which usually will mean a program crash.

However, exception handling is more than just *try-catch-finally* blocks. It also encompasses two important aspects, related between them and defined for each platform. One is the conceptual relation between a method and the exceptions it can throw; the other is the existence of an obligation of handling for an exception thrown by a method. These aspects are different for the .NET and JAVA platforms, substantially used throughout the paper as examples.

In JAVA [3], programmers can declare some of the exceptions that a given method, *m1*, throws as a whole, by using the *throws* clause in the method's declaration and explicitly throwing exceptions using the *throw* instruction in the method's body. If they use this possibility, exception information is naturally bound to *m1* and becomes connected to that method (it can even be accessed through reflection). This also has the effect of forcing programmers of another method, *m2*, which calls *m1*, to either setup a *try-catch-finally* block to handle *m1*'s possibly thrown exceptions, or declare *m2* as thrower of those exceptions, using the same process as for *m1*. A JAVA compiler will refuse to compile a program in which a programmer does not use one of these possibilities for all exceptions that methods called by that program are declared to throw. This type of "declared" exception is known as a *checked exception* because the compiler performs some type checking on it during compilation. Other exceptions are known as *unchecked exceptions*. The compiler pays no attention to them. Only the runtime does.

.NET has a different approach than that of JAVA. There are no checked exceptions. All exceptions are unchecked. Programmers cannot, even if they wish to, declare a method as thrower of an exception, and so, the relation between a method and the exceptions it can throw is weaker. Thus, a programmer will never be warned by the compiler if he forgets to handle an exception. Plus, another programmer, reflectively accessing a method entity, has no possibility of discovering which exceptions it may throw. .NET reflection does not give programmers access to exception information related to a method. But a *throw* instruction still exists, which means programmers can, and do, use it to throw exceptions in methods.

Of course Microsoft, as Sun, did not neglect the fact that programmers need to be aware of a method's behaviour in certain exceptional circumstances. Their answer is something that also exists in JAVA and it consists of special documentation tags that programmers can use to document their code in respect to exceptions.

Specially designed tools can then parse the code looking for those tags and generate suitable documentation files. In JAVA, HTML files are generated by

[1] .NET introduced a language known as managed C++, with roughly the same syntax as "old" C++, but being a safer language, susceptible to translation to Microsoft Intermediate Language (MSIL). This means some care is needed when referring to C++ as something independent of .NET.

Javadoc [4]. In .NET, custom-formatted XML files are generated by Visual Studio .NET (VS.NET) [5]. A number of other tools can then convert from this XML to Compiled HTML (CHM) format and from this to HTML. Without this kind of mechanisms, it would be rather difficult to identify and handle erroneous situations through the use of unchecked exceptions.

The question that arises from the previous considerations in the scope of this paper is that of the possibility to determine, by looking at exception documentation quality, the effectiveness of the unchecked exceptions approach.

3 Related Work

The work in this paper was initially inspired by [6] although in the current form, it accomplishes an entirely different objective. Our original goal was to inject faults (unhandled exceptions) into .NET code and evaluate the effect of those faults in several applications. That is why a special software tool was designed – to identify spots where unhandled exceptions could be thrown. Along the way, we temporarily detoured to the analysis here presented.

[6] alerts to the fact that detailed knowledge of programmers about their software only reflects reality to the extent that that knowledge is correct and complete. This, along with the major problem of keeping manually generated documentation in sync with an evolving system [7], presents a difficulty in building robust software.

Although some authors, like DeVale, et al. [8], have tried to improve software reliability by techniques other than exceptions, many others, in what is clearly the dominant trend, have simply accepted exceptions as something the industry has to live with and directed their efforts towards better ways of using exception handling mechanisms. Both Robillard, Murphy [9] and Ferreira [10] have tried to improve the use of exception handling, while avoiding associated problems, by including such concerns very early in the Software Design methodology. But [9] still alerts to the difficulties of doing so. Other authors suggest using Aspect Oriented Programming to the same effect, but at the coding level [11].

The works in [6], a technique for determining failure propagation paths through fault injection, and [12], a technique for coverage testing of recovery code, through fault injection are, in our view, marred by insufficient applicability and extensibility because of their exclusive concern with checked exceptions (they consider a fault to be a checked exception). Obviously, such approaches are of little use in .NET and other platforms like Ruby [13], that don't use checked exceptions at all. And such platforms are currently the rule rather than the exception [14].

The checked vs. unchecked exceptions discussion has had numerous interesting episodes. Ryder and Soffa [1] present an historical overview of some of the older ones, stating that "there is a symbiotic relationship between software engineering research and the design of exception handling in programming languages". They trace the roots of exception handling back to Lisp as early as the mid-1950's and acknowledge J.B. Goodenough's 1975 seminal work [15] in this field. What is interesting to notice is that Ryder and Soffa end by saying that "strong typing in programming languages, desirable in new language designs, was a direct answer to

concerns about software reliability and correctness" which agrees with Goodenough's advocating of "compile-time checking of the completeness of exception handling". This means that for at least the last 25 years, checked exceptions have been regarded as good for reliability. But the modern try-catch construction only appeared a little over 10 years ago in C++ [16], and JAVA, the most popular language to use checked exceptions, is from 1996, which means that hands-on experience with this topic is still recent.

Bruce Eckel, in a famous 2003 article [14], spurred the revival of this discussion, clearly considering checked exceptions in JAVA a failed experiment. Checked exceptions initially seemed promising because of the assumption that static type checking is always the best thing, which was precisely what Goodenough thought. But Eckel says this assumption is wrong largely because of his experience with Python [17], a much more relaxed language with regard to type checking, that even so seems to work very well with regard to reliability. He criticizes JAVA's approach by saying that it puts too much weight on the programmer, making it bad for productivity. Finally, he blames "swallowed exceptions" (programmers' tendency to use empty or almost empty exception handlers) for the failure of checked exceptions. Eckel's solution [18], besides the dumping of checked exceptions, involves much more emphasis on Software testing, especially unit testing.

Eckel's views are supported by Waldhoff [19], who wrote in an article suggestively titled "JAVA's checked exceptions were a mistake" that "checked exceptions are disastrous for the connecting parts of an application's architecture". If this is so, it is definitely not what you want in a language aimed at component-based development.

The group of checked exceptions critics is involuntarily enlarged by Robillard and Murphy. Using a practical example, they conclude that "although checked exceptions have many benefits, they can be expensive to implement". This is due to the fact that checked exceptions force programmers to alter every method in the chain connecting an exception thrower to an exception handler, whenever the group of types of exceptions possibly thrown is modified. In the presence of large method call propagation graphs, this is impractical.

All this seems to render much of Brian Goetz's advice [20] on which exceptions to check and not to check, useless. However, he makes a very good point of saying that not checking exceptions usually means not documenting exceptions. And because authors, whether supporting checked exceptions or not, agree on good documentation as an important part of the solution to the problem of Software reliability connected to exception handling, not documenting exceptions is a problem.

Given their approaches, Microsoft appears to be telling us to always use unchecked exceptions and Sun appears to be telling us to never use unchecked exceptions. In practice, what this means is that Microsoft wants to trust programmers completely, trusting that they will document whatever they see as important. Sun believes that this is unreasonable and that a mechanism to enforce reliability is in order. Either way, the special documentation tags that both companies introduced tell us that they also agree on the importance of good exception documentation.

The implications of bad documentation are largely unknown. [21] was the only reference found that addressed this. In it, Robert Read says that "documentation, if not written properly, can lie" and that this is much worse than bugs or confusion in

the source code. He prefers not having documentation to having bad documentation, saying that "code and documentation cannot be inconsistent if there is no documentation".

4 The Analysis

This section of the paper discusses the whole process that allowed for the determination of the extension of the problem of poor quality documentation or absence in .NET components.

4.1 Strategy

The strategy followed in this work was to take a set of software components and examine both the binary file containing their code looking for unhandled exceptions and the components' documentation to evaluate the extent to which one analysis corresponded to the other. To do this, a specific tool had to be developed, because although .NET provides good reflection mechanisms, no tool could perform the variety of tasks and analyses aimed for. This required us to go through every instruction of a component. Since no access to high-level source code could be assumed (important for the analysis of *common-off-the-shelf* components), this needed to happen at a lower level. All .NET programs, regardless of the original language they are written in are transformed into a low-level common form known as Microsoft Intermediate Language (MSIL or IL), an assembly-like language which is our real object of analysis. The Runtime Assembly Instrumentation Library (RAIL) [22] provides us with this kind of access, effectively establishing a bridge between .NET reflection, which goes as far as the method level, and IL code. Unfortunately, it proved not mature enough for the exhaustive use necessary in this case. Microsoft's ILDASM tool, along with simple text parsing, was used for this purpose instead.

Once the tool was ready and hence a mechanism to compare code and documentation was available, the set of software components (named *Assemblies*, in .NET) was chosen. Some of these components are core parts of certain applications while others extend the functionality of bigger infrastructures. Some were not built with re-use in mind while others were built especially for re-use. Finally, the tool was run for each component and its documentation, and results were gathered.

4.1.1 Analyzer Tool

The Analyser tool is a command-line program written entirely in C#, one of .NET's high-level languages. We only discuss here the characteristics of it that are relevant for the paper.

As input, the Analyser receives a .NET Assembly (a DLL or EXE file) location and optionally a documentation file location, in the Visual Studio .NET generated XML format. A number of switches can be used to specify different options.

As output, an XML report is generated. The format of this report depends on the command-line options but, in general, consists of a list of the methods found in the assembly given as input. For each of those methods, a number of exception detections are depicted. Each of those exception detections represents one of two things:

- that an exception not handled by any *try-catch* blocks can be generated by a given instruction (i.e. line) in the method's code (referred to as *code exceptions*);
- that an exception was identified as possibly thrown by a method in that method's documentation (referred to as *documentation exceptions*).

They result respectively from *code analysis* and *documentation analysis*. At the end of the report, a large number of statistics is displayed, along with some information about exception classification into groups. Also, if the Analyser is instructed to do so, it can automatically check the differences between what it detected in the code and in the documentation. In this case, the report will also contain a section dedicated to *suspects*. Further discussion of suspects will take place in Sections 5 and 6. For now, it is important to know that suspects roughly represent exception detections corresponding to situations where the programmer could have done a better job of documenting his code.

Suspects are important for two reasons. First, the Analyser is very thorough in its analysis and although it detects huge amounts of uncaught exceptions, only a relatively small number can be realistically expected to be documented by a programmer. Second, there are situations where it is simply impossible for the Analyser to evaluate if code exceptions are documented or not, for absence of documentation. So, suspects are a way of filtering the relevant detections.

The documentation analysis process is straightforward. It consists of parsing the given documentation looking for the specific XML tags that identify the documentation of an exception.

The code analysis process is much more complicated. It consists of going through an Assembly's public and protected[2] members, IL instruction by IL instruction, keeping track of entries/exits into/out of try-catch blocks through the use of data structures such as stacks and queues[3]. For each instruction in the code, four different types of detection are performed, searching for possible exceptions thrown by that instruction. The first type of detection consists of a simple search in a manually generated dictionary which associates IL instructions with the exceptions they can throw, as defined in the .NET platform specification [2]. Figure 1 shows an extract of that dictionary.

---

[2] When comparing documentation and code, it is reasonable to analyse only public and protected members, because we expect programmers to document at least the public interface of the components. But the Analyser is also capable of analysing private members and that is easily configurable.

[3] See [12] and [23] for more detailed descriptions of some of the techniques involved in this kind of analysis.

```
add.ovf D6 System.OverflowException
call 28 System.Security.SecurityException
div.un 5C System.DivideByZeroException
ldind.i1 46 System.NullReferenceException
```

Fig. 1. Some lines of the dictionary, adapted. The format is IL *instruction/opcode/list of exceptions*.

We will call this type of detection, IL instruction detection (ILI).

The second type of detection is called method call detection (MC) and is only applied to five IL instructions that correspond to the execution of another method – *call*, *calli*, *callvirt*, *newobj* and *jmp*. For these instructions, we perform documentation parsing looking for exception documentation for the called method.

The third and fourth types of detection are explicit throw detection (T) and explicit rethrow detection (RT). They apply, respectively, to the *throw* IL instruction and the *rethrow* IL instruction. Explicit throw detection is straightforward, but explicit rethrow detection involves keeping track of the type of the exception being rethrown, which is declared at the corresponding catch block (*rethrow* instructions only make sense inside catch blocks).

When these four types of detection are concluded, we have a set of exceptions that a given IL instruction can throw. But obviously, that doesn't mean that they are not being caught. So, we have to check if the analysis is currently being performed inside one or more nested try blocks, to determine which of the previously gathered exceptions are being caught and which are not.

The code analysis process is sketched in Figure 2. Both caught an uncaught exceptions are represented in the final report, because it is of interest to check which types of exception programmers catch and which they do not.

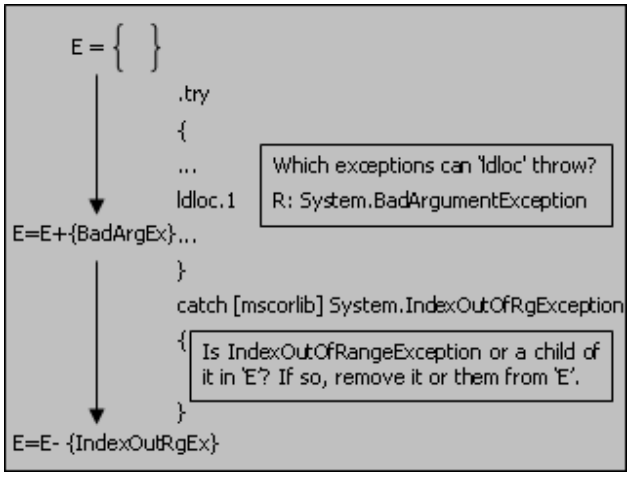

Fig. 2. Scheme of the code analysis process. 'E' represents the set of uncaught exceptions.

Code and documentation analyses produce two sets of exception detections. If so instructed, the Analyser then checks for the differences between these two sets, producing a final set of documented and undocumented exception detections.

4.2.2 The Assemblies

Selecting the study's target assemblies was the most tedious and difficult part of this work. For mainly two reasons: the lack of available and especially popular .NET applications; and the lack of proper documentation for the existing applications.

The first reason is the responsibility of the lack of penetration of the .NET platform. In fact, the .NET platform is still an alternative, rather than a first choice, for developers and decision-makers. For this work, this meant having to search in secondary forums aimed at the sharing of .NET applications (e.g. [24]), opposed to the more usual, like [25].

The second reason is even more serious and penalizing, because there were many cases where promising candidate applications were excluded solely because of lacking proper documentation. Proper documentation means the inexistence of VS.NET XML-format files accompanying the Assemblies. This can have two causes: simple skipping of this step by programmers using VS.NET (or not using of VS.NET at all); or lack of proper documentation tags throughout the source code.

Although the first cause is perfectly possible, especially in cases where VS.NET is not used at all (e.g. the Mono [26] development is completely independent of VS.NET), browsing through the source code of the discarded applications clearly indicates that the lack of proper documentation is quite common.

Actually, both the lack of proper XML documentation files and lack of XML documentation tags in source code make interesting points towards one of the major findings of this study: programmers cannot be trusted to document exceptions.

Eight Assemblies were chosen as the targets for this study. They span through different purposes and sizes. To help in the characterization of the Assemblies, a division into groups of similar purpose was created. This division is shown in Table 1.

Table 1. Group Characterization

Group	Characterization
Applications	Application Assemblies. Low re-use expected. Few public documentation needs.
Libraries	Libraries. High re-use expected and high public documentation needs.
Infrastructure	Infrastructure Assemblies. Highest re-use expected and high documentation needs.

Table 2 presents a summary of the eight assemblies chosen for this study, identifying their source application, and emphasizing their division into the groups in Table 1.

Table 2. Assemblies used in the study

Group	Assembly	Application
Applications	NAnt.Core.dll	NAnt
	NDoc.Core.dll	NDoc
	nunit.framework.dll	NUnit
Libraries	PdfLibrary.dll	PdfLibrary
	SharpZipLib.dll	SharpZipLib
	CpSphere.Mail.dll	CpSphere
Infrastructure	System.Runtime.Remoting.dll	.NET platform
	System.XML.dll	

NAnt [27] is the .NET port of the Ant build tool; NDoc [28] is an extensible code documentation generation tool for .NET; and NUnit [29] is the .NET port of JUnit for JAVA, a testing framework. Due to the size of the applications, only the main assembly of each one was analyzed. Even so, exactly due to their size, they are representative of the rest of the code.

PdfLibrary [30] is a .NET library for the generation of PDF documents; SharpZipLib [31] is a .NET data archiving/compression library supporting all popular standards like Zip, Tar, GZip, BZip, etc; and CpSphere [32] is an implementation of the SMTP protocol which can be used to add mail sending capabilities to .NET applications. The first two libraries are single-Assembly, and that Assembly is the target. For CpSphere, the main Assembly was selected as a target.

Finally, two of the .NET core platform Assemblies were chosen, mainly based on relevance (they are highly used) and documentation availability. Both Mono and Rotor [33] were also considered as sources for the Infrastructure Assemblies. But, while in the first case the documentation style is different than that of VS.NET[4], in the second case, the examined source files (XML documentation is not included) contained no exception documentation.

[4] Some of Mono's Assemblies include excellent exception documentation despite not using Microsoft style documentation tags. This just emphasizes the lack of agreement between different players and the fragility of this method.

5 Results

Table 3 summarizes the results obtained by running the Analyzer for the eight targets, showing the percentage of documented and undocumented exceptions.

ILIs (Section 4.1.1) were not considered in this analysis although they represent a huge amount of exceptions. This is because we think it is not reasonable to expect programmers to document them. They are thrown by the low-level IL instructions at the virtual machine level, which do not correspond to problems that the programmer should usually deal with. Unfortunately, this means that they can still cause problems, but it also means that they are usually poorly documented or documented "by coincidence" (read ahead for an explanation). Normally, programmers will only marginally be aware of them.

Table 3. Documented vs. Undocumented exceptions

Group	Assembly	% documented	% Not Documented
Applications	NAnt.Core.dll	3.36	96.64
	NDoc.Core.dll	0.46	99.54
	nunit.framework.dll	0.00	100.00
Libraries	PdfLibrary.dll	0.00	100.00
	SharpZipLib.dll	21.15	78.85
	CpSphere.Mail.dll	8.43	91.57
Infrastructure	System.Runtime.Remoting.dll	16.19	83.81
	System.XML.dll	23.45	76.55
	Average	**9.13**	**90.87**
	Average for Applications	**1.27**	**98.73**
	Average for Libraries	**9.86**	**90.14**
	Average for Infrastructure	**19.82**	**80.18**

Table 3 shows that for the set of 8 different Assemblies over 90% of the relevant exceptions that the code can throw are not documented. For code directed mainly towards the end-user, this value goes up to almost 99%. For code aimed towards re-use by other programmers (libraries), it stands at about 90%. For infrastructure code, providing basic services for the .NET platform, this value is still as high as 80%.

Table 4 discriminates the types of exceptions that in the previous table are mentioned as documented. Thus, it offers insight into the types of exceptions that programmers are most likely to document.

Table 4. Types of exceptions most likely to be documented

Group	Assembly	% ILIs	% MCs	% Ts	% RTs
Applications	NAnt.Core.dll	0.00	24.49	73.47	2.04
	NDoc.Core.dll	0.00	0.00	100.00	0.00
	nunit.framework.dll	0.00	0.00	0.00	0.00
Libraries	PdfLibrary.dll	0.00	0.00	0.00	0.00
	SharpZipLib.dll	4.55	11.36	84.09	0.00
	CpSphere.Mail.dll	0.00	17.24	82.76	0.00
Infrastructure	System.Runtime.Remoting.dll	0.00	100.00	0.00	0.00
	System.XML.dll	24.37	41.94	33.69	0.00
	Average	**4.82**	**32.51**	**62.34**	**0.34**
	Average for Group A	**0.00**	**12.24**	**86.73**	**1.02**
	Average for Group L	**2.27**	**14.30**	**83.42**	**0.00**
	Average for Group IS	**12.19**	**70.97**	**16.85**	**0.00**

The four most interesting graphs are also shown in the next figure.

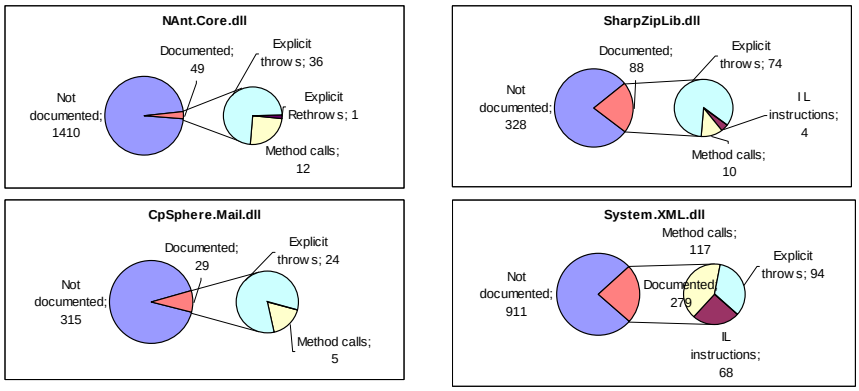

Fig. 3. Documentation of exceptions in four different assemblies.

More than 60% of the documented exceptions represent explicit throws. This value is about 85% for application and library code and about 17% for infrastructure code. This huge discrepancy may be attributed to the presence of outliers in the target Assemblies but is more likely to be caused by very specific documenting styles/procedures of Microsoft for the .NET platform.

Most of the other documented exceptions represent exceptions that result from method calls. This value is about 32.5% for all Assemblies, but only about 13% for application and library code, and as high as 71% for infrastructure code.

This data seems to indicate that Microsoft has an automatic way of documenting its code, particularly in respect to method calls, because unlike most code, where documentation about exceptions resulting from method calls is rare, Microsoft code is much more complete in this regard.

For all assemblies, explicit re-throws represent very low values.

It is very interesting to note that despite not having included ILIs in our analysis, they still appear as documented for two of the target Assemblies. These represent simple coincidences that occur when a programmer involuntarily documents an ILI exception by documenting one of the other types of exception (e.g. imagine that a programmer explicitly throws a NullReferenceException and documents that fact using documentation tags before the method header. If the IL instructions in that method throw a NullReferenceException, he will have documented more than he expected, possibly misleading anyone using his code as to the circumstances in which an exception occurred).

Although the preceding results may be very useful, they still represent a simple analysis, only taking into account what can be determined by looking directly at IL code and respective documentation. They tell us about problems we can expect to encounter when using these Assemblies. But they don't tell us which of these problems are of the direct responsibility of programmers, mainly because they consider cases where there is no documentation available and cases where exceptions were not documented in the existing documentation to be equal.

To solve this problem, the concept of suspects, mentioned in Section 4.1.1, was created. Suspects represent cases where we can state for sure that programmers could have done a better job on documenting their code. There are two types of suspects: code suspects and documentation suspects. Code suspects represent uncaught exceptions, detected in the code analysis and not in the documentation analysis, not originating from ILIs *and that were found in methods for which there is documentation*. Documentation suspects are more serious. They represent exceptions that the programmer possible to occur in his code but that were not detected in the code analysis and are, therefore, impossible to occur. Prior to running the Analyzer, we did not expect to find any documentation suspects, but Table 5, summarizing the results, shows that we still did find some of these cases.

Table 5. Suspects for all eight Assemblies

Assembly	Code	Doc	Total
NAnt.Core.dll	854	0	854
NDoc.Core.dll	387	0	387
nunit.framework.dll	20	0	20
PdfLibrary.dll	28	0	28
SharpZipLib.dll	276	5	281
CpSphere.Mail.dll	221	2	223
System.Runtime.Remoting.dll	155	0	155
System.XML.dll	421	16	437

The documentation suspects we found are all, without exception, due to a specific feature of .NET – properties. Properties are internally implemented in .NET as one or two methods (depending on the fact of the property being read-only or not), a get_<Property> method and, possibly, a set_<Property> method. But the documentation tags only allow documenting a property as a whole (the internal methods are completely transparent to the programmer). This carries more than one consequence. First, it means that if the documentation is not specific enough (it can explicitly say that the exception occurs only in setting the property), the programmer cannot know if the exception occurs in the getting or the setting of the property. Second, it renders attempts to do automatic exception handling or exception analysis like ours even more difficult, because it is short of impossible to have the computer read and interpret what someone wrote. We chose to have the Analyzer signal all these cases as suspects.

Table 6. Type of detections responsible for code suspects

Assembly	Code Suspects	#MCs (%)	#Ts (%)	#RTs (%)
NAnt.Core.dll	854	793 (93%)	60 (7%)	1 (0.1%)
NDoc.Core.dll	387	374 (97%)	13 (3%)	0 (0%)
nunit.framework.dll	20	16 (80%)	4 (20%)	0 (0%)
PdfLibrary.dll	28	28 (100%)	0 (0%)	0 (0%)
SharpZipLib.dll	276	236 (86%)	40 (14%)	0 (0%)
CpSphere.Mail.dll	221	217 (98%)	4 (2%)	0 (0%)
System.Runtime.Rtg.dll	155	139 (90%)	16 (10%)	0 (0%)
System.XML.dll	421	373 (89%)	48 (11%)	0 (0%)

For the cases where we can state for sure that existing documentation is lacking in quality, around 90% of missing documentation is related to insufficient accounting of the exceptions that can occur by calling other methods, the rest being explicit throws, which are fairly well documented (as you would expect).

Finally, it is possible to compare the numbers presented in Figure 3 and Table 6 to get an estimate of the proportion of undocumented cases that are due to the plain absence of documentation. For this, we can take the number of undocumented detections from Figure 3 (joining in the values for the other 4 assemblies) and the

number of code suspects from Table 6. The results of this comparison are shown in Table 7.

Table 7. Proportion of detections due to lack of documentation

Assembly	Undocumented Detections	Code Suspects	Lacking Proportion
NAnt.Core.dll	1410	854	39.4%
NDoc.Core.dll	430	387	10.0%
nunit.framework.dll	20	20	0.0%
PdfLibrary.dll	52	28	46.2%
SharpZipLib.dll	328	276	15.9%
CpSphere.Mail.dll	315	221	29.8%
System.Runtime.Rtg.dll	466	155	66.7%
System.XML.dll	911	421	53.8%

Given the way that the Analyzer statistics were produced, these numbers can be extrapolated to represent the amount of methods for which there is no documentation despite the inclusion of documentation for the assembly. As can be seen, there is a large variability in the results, but there are still cases for which more than 50% of the methods did not have documentation, including both the Microsoft .Net core platform ones.

6 Conclusions

This paper shows the magnitude of the problems of documentation absence and documentation quality in .NET Assemblies. More emphasis was put on the problem of documentation quality but Table 7, together with the great difficulties found when collecting Assemblies for this work, are a testimony of the problem of documentation absence.

Regarding documentation quality, in general 90% of relevant exceptions thrown are not documented. These values range from around 80% to almost 100% growing as the amount of re-use expected declines. For the cases where it is possible to state for sure that existing documentation is lacking in quality, around 90% of missing documentation is related to insufficient accounting of the exceptions that can occur by calling other methods, the rest being explicit throws, which are fairly well documented. This fact indicates that there may be benefits in developing ways of somehow automatically documenting methods by following call chains looking for the exceptions that may be propagated.

Ultimately, this study brings us to the checked vs. unchecked exceptions discussion. It is quite clear that further studies need to be performed, not only in .NET, but especially in JAVA, its main contender. That data will enable a direct comparison of the effect, if any, of using checked exceptions. And despite the claim of this paper, that a much bigger problem is that of programmers not sufficiently documenting their code, given the fact that the majority of undocumented exceptions relate to method calls, those studies might yield more significant conclusions. Why?

Because checked exceptions are a means of getting exception information directly from a method, not having to manually go through all the chain of calls looking for exceptions, which is one of the downsides of unchecked exceptions. If the exception information associated to the method is accurate (which is very likely, because it is a compile-time check), programmers have one less excuse for not documenting their code. And there is still a bigger consequence. Checked exceptions give us the possibility of improving techniques like automatic exception handling and even automatic code documentation.

Thus, our opinion is that checked exceptions, or a variation on them, may prove more beneficial to dependability. Our thought is that checked exceptions will not make programmers be more inclined to document. But they will at least make automation techniques, which seem to deserve a lot of support, much easier. Even so, probably the major conclusion that can be drawn from the use of exceptions and of the checked vs. non-checked exceptions discussion is that currently the error handling mechanisms available in programming languages are not good enough and that more research in this important area is needed.

Acknowledgements

This investigation was partially supported by the Portuguese Research Agency – FCT, through the CISUC Research Center (R&D Unit 326/97).

References

1. B. G. Ryder, M. L. Soffa, “Influences on the design of exception handling”, ACM SIGSOFT project on the impact of software engineering research on programming language design, in ACM SIGSOFT Software Engineering Notes, ACM Press, Vol. 28 (4), July 2003.
2. ECMA International. Standard ECMA-335 Common Language Infrastructure (CLI), ECMA Standard, 2003.
3. 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.
4. “Javadoc Tool Home Page”, http://java.sun.com/j2se/javadoc/.
5. “Visual Studio Home”, http://msdn.microsoft.com/vstudio/.
6. G. Candea, M. Delgado, M. Chen, A. Fox, “Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications”, Proceedings of the 3rd IEEE Workshop on Internet Applications (WIAPP), IEEE Press, June 2003.
7. F. P. Brooks, “The Mythical Man-Month”, Addison-Weisley, Reading, MA, Anniversary edition, 1995. ISBN 0-201-8359-59.
8. J. DeVale, P. Koopman, “Robust Software – No More Excuses”, in Proceedings of the International Conference on Dependable Systems and Networks 2002 (DSN’02), IEEE Press, June 2002.
9. 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, Vol. 25 (6), ACM Press, November 2000.

10. G. R. M. Ferreira, C. M. F. Rubira, R. Lemos, “Explicit Representation of Exception Handling in the Development of Dependable Component-Based Systems”, in Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering (HASE’01), IEEE Press, October 2001.
11. M. Lippert, 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.
12. C. Fu, R. P. Martin, K. Nagaraja, T. D. Nguyen, B. Ryder, D. Wonnacott, “Compiler-directed Program-fault Coverage for Highly Available JAVA Internet Services”, in Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN ‘03), IEEE Press, June 2003.
13. “Ruby Home Page”, http://www.ruby-lang.org.
14. B. Eckel, “Does JAVA need Checked Exceptions?”, http://www.mindview.net/Etc/Discussions/CheckedExceptions
15. J. B. Goodenough, “Exception handling: issues and a proposed notation”, in Communications of the ACM, Vol. 18 (12), ACM Press, December 1975.
16. A. Koenig, B. Stroustrup, “Exception handling for C++”, in “The evolution of C++: language design in the marketplace of ideas”, MIT Press, 1993.
17. “Python Programming Language”, http://www.python.org.
18. B. Eckel, “Strong Typing vs. Strong Testing”, http://mindview.net/WebLog/log-0025.
19. R. Waldhoff, “JAVA’s checked exceptions were a mistake (and here’s what I would like to do about it)”, http://radio.weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html, 1 April 2003.
20. B. Goetz, “To check, or not to check?”, from “JAVA theory and practice: The exceptions debate”, http://www-106.ibm.com/developerworks/java/library/j-jtp05254.html.
21. R. L. Read, “How to be a Programmer: A Short, Comprehensive, and Personal Summary”, 2002, http://samizdat.mines.edu/howto/HowToBeAProgrammer.html.
22. “RAIL - Runtime Assembly Instrumentation Library Project”, http://rail.dei.uc.pt.
23. S. Sinha, M. J. Harrold, “Analysis and Testing of Programs with Exception Handling Constructs”, in IEEE Transactions on Software Engineering, IEEE Press, Vol. 26 (9), September 2000.
24. “The Code Project – Free Source Code and Tutorials”, http://www.codeproject.com/.
25. “Sourceforge.net”, http://sf.net.
26. “Mono”, http://www.go-mono.com.
27. “NAnt – A .NET Build Tool”, http://nant.sourceforge.net.
28. “NDoc Code Documentation Generator for .NET”, http://ndoc.sourceforge.net.
29. “NUnit”, http://www.nunit.org.
30. “PDF Library for creating PDF with tables and text in C#”, http://www.codeproject.com/dotnet/PdfLibrary.asp.
31. “SharpZipLib, The Zip, GZip, BZip2 and Tar Implementation For .NET”, http://www.icsharpcode.net/OpenSource/SharpZipLib/Default.aspx.
32. “CpSphere Email Component for .NET”, http://www.codeproject.com/dotnet/cpSphereEmailComponent.asp.
33. “The Microsoft Shared Source CLI Implementation”, http://msdn.microsoft.com/net/sscli.