Towards An Integrated Classification of Exceptions

Annelise Janse van Rensburg\textsuperscript{a}  \quad Karen Renaud\textsuperscript{b}

\textit{University of South Africa}

\textsuperscript{a} aисg@lantic.net \quad \textsuperscript{b} renauv@unisa.ac.za

Abstract : Exceptions refer to the situations that are not modelled by the system, or to deviations between what is planned and what really happens. In order to develop software systems which are more robust, thus delivering higher availability at a lower operating cost, the occurrence of exceptions needs to be reduced and the effects of the exceptions controlled. The number of exceptions arising during a software product’s execution is not an indication of its quality, since the execution depends on facts beyond the control of the software developer, such as, for example, the environment at execution time. The reaction of the product in response to the exceptions is, on the other hand, a definite measure of the quality. Errors that occur in programs, which result in exceptions being raised, are not all the same. A distinction should be made between the different types in order to categorize them and thus to be able to handle categories in a uniform way. This paper proposes a preliminary integrated classification which is derived from existing classifications.

Keywords: Exceptions, classification, software development

Computing Review Categories: D.2.2, D.2.3, D.2.9, D.2.10

1. Introduction

Application developers often do not consider exceptions in designing their applications. Even the ubiquitous UML does not make provision for exception handling as part of the design process [10]. This may well be done deliberately perhaps because the extra complexity introduced by considering exception handling would complicate the already complex design process even more.

When one or more programmers are implementing a system they will often handle exceptions as they see fit at the time, without giving thought to a general policy, hence the differing standards of exception-handling within a single application.

It is far better to have an exception-handling policy which programmers can adhere to. It may not seem realistic to incorporate exception handling from the beginning of the development process but the problems encountered when it is not done seem to suggest the contrary – and one feels that it should be compulsory to have an explicit exception-handling policy spanning the whole development process [3, 12, 13, 14]. However, this can only work if exceptions can be classified and different types handled according to specific norms – depending on the type of application.

There are many different classifications, however, and it can be difficult to decide which one to use, and which one will be best for a particular application. We have therefore identified the need to establish an integrated classification which programmers and system designers could use to establish a consistent and overall exception-handling strategy throughout their application development process.

Section 2 defines exceptions and discusses their nature. Section 3 explores different classifications of exceptions. Section 4 gives an integrated classification of exceptions. Section 5 concludes the paper.

2. Exceptions

Although a wide variety of definitions exist for exactly what an exception constitutes, the following two definitions give an accurate description:

- “Exceptions, situations that cannot be correctly processed by computer systems, occur frequently in computer-based information processes” [8].
- “Exceptions in our view refer to facts or situations that are not modelled by the information systems, or deviations between what we plan and what actually happen”[7].

Exceptions are a reality that must be dealt with because of the possible severity of the effects of exceptions and the unpredictability of the frequency of occurrences [5, 6, 8]. Exceptions are a real threat to any system, not only for the present but also for the future – software will
never be exception-free, hence the better the handling mechanisms developed, the higher the quality.

The efficacy of exception-handling influences the reliability of all software systems and also influences the interaction experience of all users interacting in one way or the other with a computerized system – directly or indirectly.

There are three basic views concerning exceptions, namely [8]:

1. Exceptions are unpredictable, random events that should be ignored.
2. Exceptions are errors, signalling problems of which the causes must be found and eliminated.
3. Exceptions are a normal tendency, due to changing environments and trade-offs. They should be efficiently detected and handled.

The first view is incorrect – for obvious reasons. Not all exceptions are unpredictable and neither are they random events. The effects, and the possible consequences of, exceptions on which human lives may depend make it impossible to ignore such events.

The second view is unrealistic. Exceptions can be caused by errors, but even in the case of errors it is not always possible to find the cause and to eliminate it. It is also impossible to predict and eliminate all possible errors during the systems implementation process, especially when the system is interacting with different platforms, different operating systems and different application software in a variety of dynamic environments.

The third view reflects reality.

In order to detect and handle exceptions effectively a thorough understanding of exceptions is required. This includes:

1. The identification of possible causes of exceptions.
2. The effects on the system.
3. Reporting.
4. The consequences on the environment.
5. The handling of the exceptions.

A classification of exceptions is required because exceptions, which occur in programs, are not all the same. Distinctions should be drawn between the different exception types in order to categorize them and to be able to handle categories in a uniform way. It may not be efficient to write a different exception handler for each exception.

3. Exception Classification

Many different exception classifications can be found in the literature, often formed from different perspectives. These perspectives range from general to specific, aimed at the causes of the exceptions or the effects or even the origin of the exceptions. Different classifications are listed below, some of which are very specific to particular languages and others of which attempt to be more generic.

3.1 Dickson et al. [4]

According to Dickson et al. there are two broad types of exceptions [4]:

- Expected.
- Unexpected.

Expected exceptions are those that are explicitly planned and provided for. Unexpected exceptions are simply described as unanticipated.

3.2 Duesing and Diamant [2]

Duesing and Diamant make another general classification in their work with the C++ SoftBench CodeAdvisor [2]. They describe exceptions as if belonging to two sets:

- The one set constitutes the set of all real exceptions.
- The other set – those of all reported exceptions including “false positives” (exceptions reported which are not exceptions at all).
The intersection of these sets form a third set constituting the set of all real reported exceptions as can be seen in the diagram in Figure 1.

### 3.3 Luo et al. [7]

A general analysis of exceptions by Luo et al. depicts exceptions on three orthogonal axes [7]. The labels given to the different axes are:
- Detectable.
- Resolvable.
- Known.

**Detectable** exceptions are those a system is capable to detect, i.e. the ability of the system to notice the occurrence of an exception. Exceptions classified as **resolvable** imply that the system is able to provide some form of a solution when the exception occurs. **Known** exceptions refer to those the system have currently has knowledge of, **unknown** meaning those beyond the systems current knowledge. The lengths of these axes extend as the number of detectable, resolvable or known exceptions increases, as depicted in the Figure 2.

### 3.4 Mandrioli and Meyer [1]

Mandrioli and Meyer formed the following classification of exceptions with reference to their experience with Eiffel systems in practice [1]. This classification is done while keeping the cause of the exception in mind and the following four types result:

1. An exception is found when an explicit assertion is violated.
2. A **called routine that fails** can give rise to an exception.
3. An **abnormal event** that has taken place in the **operating system or underlying hardware** causes an exception.
4. When an attempt is made to apply a **routine to a non-existing object** an exception is also created.

### 3.5 Garcia et al. [9]

Garcia et al. classify exceptions into three classes according to the origin of the exception in the code [9]:

---

**Figure 1 - Duesing and Daimant's Classification**

**Figure 2 - The classification of exceptions according to Luo et al [7].**
- **Interface exceptions** – signalled when conflict arises in the interface of a software component when input is provided different from what is expected.
- **Failure exceptions** – signalled when the services requested cannot be provided, in other words, the component or routine is unable to complete execution.
- **Internal exceptions** – raised by the component whenever an exception occurs internal to the component.

### 3.6 Bail [3]
Bail also makes use of the possible origins of exceptions as a basis for classification [3]. He states that there are two possible causes for exceptions –

- **Software defects.**
- **Erroneous data.**

Software defects refer to the errors in the code’s logic. This may be caused due to any number of reasons – programmer errors, design errors or even compiler errors. The source of erroneous data may be improper or unexpected data supplied by the user or incorrect and missing data in a database.

### 4.4 Comparison and Summary

Table 1 is a summary of all the exception classifications, with arrows pointing between the different classifications indicating correspondences. For some classifications more than one classification corresponds to the same heading in another classification.

<table>
<thead>
<tr>
<th>General Classification</th>
<th>Causes</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>Reported</td>
<td>Known</td>
</tr>
<tr>
<td>Real</td>
<td>(Unknown)</td>
<td>Routine Call Fails</td>
</tr>
<tr>
<td>Unexpected</td>
<td>Undetected</td>
<td>Resolvable</td>
</tr>
<tr>
<td>(Un= Resolvable)</td>
<td>Hardware or O/S Exception</td>
<td></td>
</tr>
<tr>
<td>False Positives</td>
<td>Detectable</td>
<td>Routine applied to Non-Existing Object</td>
</tr>
<tr>
<td>(Un= Detectable)</td>
<td>Internal Exceptions</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1: Comparison and Summary of Classifications**

Erroneous input from users is a known real problem and to be expected, it is detectable through validation checks and resolvable through, for example, displaying messages for the user as to the effect and prompting for the correct input. All real reported exceptions must be detectable in order to be reported and can be resolved, meaning, a corrective course of action can be established for the specific exception or class of exceptions. Unexpected exceptions will not be made provision for with, for example, validation checks, since it is not anticipated and might therefore go undetected. The same is true for false positives.

Undetected exceptions are unknown to the system at the time of occurrence and therefore not resolvable at the time. Any exception might pass undetected, the detection of an exception relies on the explicit detection measures provided.

### 5 Integration

The classifications mentioned earlier need to be integrated into a single classification to be of use. Although each classification has its merits and is very different from the others it is not of much use if a correlation between these cannot be found in order to establish an
An integrated classification may be useful to assist in creating general exception-handling policies. These policies may then in turn be integrated in the design process and address the need for an integrated exception-handling policy spanning the whole software development life-cycle to counteract the problems experienced with ad-hoc methods used in practice.

An integrated classification of exception has been derived and is shown in Figure 3. The graph depicts the relationship between each different type of exception and three important criteria – the predictability, detectability and resolvability of exceptions. Predictability referring to the ability to establish if an exception can occur in a system and detectability and resolvability as defined in Section 3.3. These are the most important criteria for the classification of exceptions since the life cycle of exceptions revolves around these criteria. If an exception is detectable it can be predictable as well as resolvable whereas an unpredictable exception is difficult to detect and resolve.

The origin of the axis system depicts the value (resolvable, predictable, detectable) whereas the extreme ends of the x, y and z axes depict the unresolvability, undetectability and unpredictability of exceptions, respectively. "Silent" and "Reported" exceptions are opposites: "Reported Friendly" and "Reported Unfriendly" being distinguished since the user must always be considered in reporting exceptions. The former implies that a useful exception message will be displayed such that the user will have a clear idea as to what the cause was of the exception and what measures should be taken in order to resolve the problem whilst in the case of the latter the message will contain less helpful information due to the circumstances surrounding the detection of the exception. Silent exceptions are real exceptions passing through the system unnoticed. They pass undetected and are thus unresolvable and unpredictable.

A user-friendly message will only be displayed if the exception is detectable but this is not dependent on the resolvability of the exception. To prepare a handler for the display of the message implies that the exception must be detectable through certain conditions. The message can be adapted to the exception being resolvable or not. In the case of the unpredictable exception a handler can display a standard message. This means that although it was not expected general provision has been made for unexpected, detected exceptions. The message displayed will be a general one and thus less informative.

The values “Real” and “False Positives” are also opposites. “Real” implies a true exception and “False Positives” exceptions are reported as exceptions but result from a misdiagnosis by the application. False positives are not at all predictable as indicated by their name and

<table>
<thead>
<tr>
<th>Predictable</th>
<th>Detectable</th>
<th>Resolvable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reported Friendly</td>
<td>Reported Unfriendly</td>
<td>False Positives</td>
</tr>
<tr>
<td>Real</td>
<td>Silent</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3 - Integration**
nature. Their existence is only established through them being reported and thus they are detectable and also resolvable – not necessarily eliminated, but in the sense that any future occurrences will be labelled “false positive” without the need for investigation and a corrective course of action will be set in motion according to the circumstances.

Real exceptions are detectable, resolvable and predictable in most cases. There might be situations where they are unpredictable, unresolvable or undetectable. In the case where they are unresolvable or unpredictable an appropriate message can be displayed but in the case where they go undetected no message can be displayed and the situation will also not be resolvable since it is undetectable.

Figure 3 can thus be of use to classify a variety of exceptions conceptually and thus to handle these exceptions uniformly. The integration caters for all types of exceptions.

To show the actual functionality of Figure 3 as related to all the previous classifications each of the classifications can be positioned on the diagram. The legend of Figure 3 is used to code the position of each of the classifications in Table 2.

<table>
<thead>
<tr>
<th>General Classification</th>
<th>Causes</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dickson et al. [4]</td>
<td>Reported</td>
<td>Software Defects</td>
</tr>
<tr>
<td>Expected</td>
<td>Known</td>
<td>Garcia et al. [9]</td>
</tr>
<tr>
<td>Reported</td>
<td>Mandrioli and Meyer Assertion Violation</td>
<td></td>
</tr>
<tr>
<td>Luo et al. [7]</td>
<td>Failure Exceptions</td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>Routine Call Fails</td>
<td></td>
</tr>
<tr>
<td>Known</td>
<td>Failure Exceptions</td>
<td></td>
</tr>
<tr>
<td>False Positives</td>
<td>Routine applied to Non-Existing Object</td>
<td></td>
</tr>
<tr>
<td>Detectable</td>
<td>Internal Exceptions</td>
<td></td>
</tr>
<tr>
<td>Hardware or O/S Exception</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Application of Classification

Conclusion
Although each of the classifications was different and considered exceptions from a different perspective it was possible to integrate them into one classification. It is by no means a final or set integration and it will evolve over time as more research is done and a better understanding is gained of this problem area. We believe that such a classification will be helpful to establish general policies that can be successfully integrated in the overall software development process. The main criterion for the integration of such policies, to address of the current problems surrounding exception handling, is to incorporate it in a practical and executable manner in the development process.

Future Work
This paper reports on preliminary work into an exception-classification mechanism. It should be seen as work-in-progress. We intend refining this model which will offer appropriate action for each classification.

8. References